СЕМЕНОВ А.С.

ПОРОЖДАЮЩИЕ И РАСПОЗНАЮЩИЕ ПРОГРАММНЫЕ СИСТЕМЫ

Москва 2023

СОДЕРЖАНИЕ

Введение

Системы программирования - это компьютерные программы предназначены для разработки программного обеспечения. К ним относят: ассемблеры осуществляющие преобразование программы в форме исходного текста на языке ассемблера в машинные команды в виде объектного кода, трансляторы программ, компиляторы переводящие текст программы на языке высокого уровня, в эквивалентную программу на машинном языке, интерпретаторы анализирующие команды или операторы программы и тут же выполняющие их;

Пособие направлено на освоение и применение моделей формальных языков для разработки систем программирования. При этом решаются следующие задачи: определение типа модели формального языка, доказательство правильности сделанного выбора модели, построение цепочек вывода и дерева разбора для модели, нахождения эквивалентных моделей, описание языка и построения модели распознавателя, программная реализация модели, перевод языка из одной модели в другую формальную модель. Отличительной особенностью пособия является практическое содержание и решение задач с применением программной платформы, реализующей основополагающие алгоритмы формальных моделей, которые используются для проведения практических и курсовых работ. Пособие служит документацией к базовым алгоритмам платформы. Алгоритмы представлены в виде теоретико-множественной нотации.

Пособие состоит из семи глав и приложения, в котором приведены основные шаги выполнения практических работ. Теоретический материал и практические задания организованы по темам, в соответствии с классификацией моделей формальных языков, приведенной в главе 1. Основной задачей классификации является изучение порождающих и распознающих моделей формальных языков [1-5].

Пособие организовано в порядке возрастания сложности: предшествующий материал используется в последующих главах. В процессе выполнения практических заданий развивается умение разрабатывать модели формальных языков, проводить сравнительный анализ моделей, определять их вид в соответствии с классификацией, а также применять объектно-ориентированный подход для реализации моделей на языке программирования C#.

Во второй главе рассматривается класс автоматных грамматик и конечных автоматов, которые широко применяются в различных программных приложениях объектно-ориентированного подхода и искусственного интеллекта.

В третьей главе рассмотрен класс контекстно-свободных грамматик и МП-автоматов**,** позволяющий строить оптимальным образом синтаксические анализаторы для языков программирования.

В четвертой главе рассмотрены грамматики общего вида и машины Тьюринга, приведены соотношения между языками и грамматикам.

В пятой и шестой главах рассматриваются методы перевода между различными формальными моделями.

Книга полезна для студентов, аспирантов, ученых и специалистов, занимающихся разработкой формальных моделей языков, формальными лингвистическими моделями и математическим моделированием программных систем.

Глава 1. Классификация моделей формальных языков

Языки программирования являются формальными языками, которые были разработаны для вычислений на компьютере. Формальный язык состоит из цепочек символов, которые берутся из алфавита и объединяются в языковые строки. в соответствии с выбранной моделью, которая описывает правила формирования формального языка. Модели задают строго синтаксические характеристики языков посредством структурных паттернов.

1.1. Основные понятия и определения

Определим теоретико-множественные операции и понятия, используемые для формирования моделей формальных языков.

Определение 1. *Алфавит* (или словарный состав) – конечное множество символов. *Символ* - объект, описывается атрибутами и методами, например, a_p, где p - атрибут.

Фигурные скобки используются для обозначения множеств, Например, *алфавит* $V_1 = \{1,0\}$ – обозначает множество, содержащее два символа, *алфавит* $V_2 = \{a,b,c,d\}$ – множество, содержащее четыре символа, a, b, c и d.

Пустое множество V_3 обозначается как $V_3 = \varnothing$.

Определение 2. Результат операций *объединения* $\cup$ и *пересечения* $\cap$ *множеств*, например, $A \cup B = C$, $A \cap B = D$, если $A = \{a,b,c\}$ и $B = \{c,d,e\}$ равен новому множеству $C = \{a,b,c,d,e\}$ и $D = \{c\}$:

$$\{a,b,c\} \cup \{c,d,e\} = \{a,b,c,d,e\}$$
$$\{a,b,c\} \cap \{c,d,e\} = \{c\}$$

Определение 3. Выражение $A \supseteq B$, читается как множество A включает $\supseteq$ множество B, то есть, каждый элемент B является элементом A, например $\{c,d,e\} \supseteq \{c\}$. B называют подмножеством множества A.

Определение 4. Пусть $Z = \{1,2,3,...\}$ бесконечное (обозначается ...) множество положительных целых чисел. Множество B задается с помощью предиката | - читается как "так что" или "при условии":

$B = \{x \mid x \in Z$, x есть четное число$\}$ читается, как B есть множество символов x, таких что $x \in Z$ и (, - запятая читается как и) x есть четное число. Тогда $B = \{2,4,6,...\}$ бесконечное множество четных чисел. Предикат | от условия "$x \in Z$, x есть четное число" принимает значение *истина* или *ложь*.

Модель формального языка L можно сформировать, пользуясь заданием множества с помощью предиката, например:

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

Язык L включает цепочки символов, состоящие из нуля или нулей и того же числа последующих единиц n. Пустая цепочка символов включается в язык, что намного проще синтаксисов большинства моделей формальных языков.

Определение 5. *Цепочкой символов α в алфавите* V называется любая конечная последовательность символов этого алфавита. Например, $\alpha = 01$. $V=\{0,1\}$

Определение 6. *Пустой цепочкой символов* ε называется цепочка, не содержащая ни одного символа.

Определение 7. *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.

Определение 8. *Декартовым произведением* $A \times B$ множеств A и B называется множество пар (a, b) записывается, как $A \times B = \{ (a, b) \mid a \in A, b \in B \}$, где пара (a, b) элемент множества.

Определение 9. Операция итерации – обозначается парой круглых скобок со звездочкой $(a)^*$. Результатом является множество, включающее цепочки всевозможной длины, построенные с использованием символа **а**. Например: $(a)^*$ = (a), (aa), ..., (aaa...)

Определение 10. Унарный оператор, **замыка́ние Кли́ни** или звезда́ Кли́ни, обозначается V* определяется над множеством символов V, замкнут относительно операции конкатенации символов из V. Результатом замыка́ния Кли́ни является бесконечное надмножество множества V или множество цепочек символов, включая пустую цепочку ε. Например, если конечное множество V={0,1}, то V^* = {ε, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...} есть бесконечное множество цепочек символов.

Замыка́ние Кли́ни без пустой цепочки ε, обозначается как V^+ – бесконечное множество всех цепочек алфавита V, исключая пустую цепочку ε. Следовательно, $V^* = V^+ \cup \{\varepsilon\}$.

Определение 11. Оператор импликации → (от лат. implicatio - «связь») A → B - бинарная логическая связка, читается "если A, то B". Применяется в правилах продукций (порождений) грамматик A → B, читается "символ A порождает символ B".

Определение 12. Бинарный оператор вывода ⇒ означает, что символы слева от оператора заменяются или подставляются (принцип подстановки) символами справа, в соответствии с правилом продукции грамматики. Оператор применяется в порождающих системах. Например, A → Bb, B → c, тогда A ⇒ Bb ⇒ bc. Полученная цепочка символов называется цепочкой вывода или выводом.

Определение 13. Операция (также правило) перехода δ или функция перехода автомата.

Определение 14. Бинарный оператор ⊢ означает такт срабатывания модели автомата или алгоритма в соответствии с правилом перехода δ (применяется в распознающих системах).

Замыка́ние Кли́ни, оператор вывода ⇒ и такт срабатывания ⊢ являются основополагающими операторами для формирования порождающих и распознающих моделей формальных языков.

1.2. Платформа моделей формальных языков

Появление моделей формальных языков было обусловлено стремлением формализовать естественные языки и языки программирования. Рассмотрим основополагающие порождающие и распознающие модели описания формальных языков см. рис. 1.1.

Рис. 1.1. Платформа моделей формальных и естественных языков

Обработка естественного языка (Natural Language Processing– NLP) текстов и речи с привлечением формальных моделей и алгоритмов машинного обучения. Например, можно использовать NLP, чтобы создавать системы распознавания речи, обобщения документов, машинного перевода, выявления спама, распознавания именованных сущностей, ответов на вопросы, автокомплита, предиктивного ввода текста и т.д. [10-15]

Порождающие модели

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

В порождающих моделях правила 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

В рассматриваемых правилах порядок элементов фиксирован. Для определения грамматического формализма, эквивалентного синтаксическому, используем правило продукций →. Тогда порождающая модель, для рассматриваемого предложения, примет вид:

Предложение → подлежащее, группа сказуемого

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

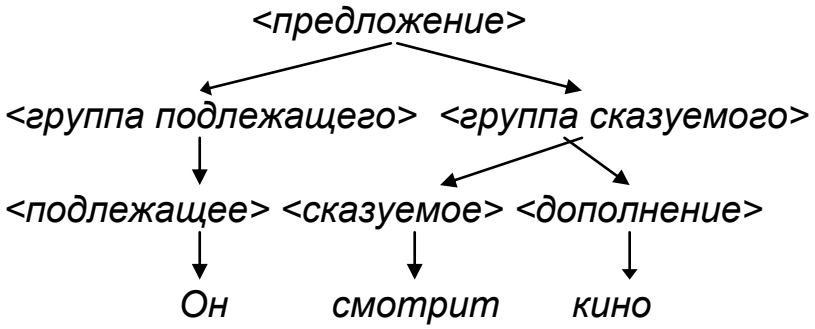


Рис. 1.2. Древовидная структура предложения

Если шаги подстановки интерпретировать как шаги построения графа дерева то получим дерево синтаксического разбора см. рис. 1.2., спускаясь по которому из вершины <предложение> получим предложение “он смотрит кино”.

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

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

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

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

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

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

Рассматриваемый пример в БНФ:

<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> ⇒ <подлежащее> <группа сказуемого>

Заменим синтаксическое понятие на одну из цепочек, из которых оно может состоять. Повторим процесс. Возьмем один из метасимволов в цепочке <подлежащее> <группа сказуемого>, например <подлежащее>; найдем

правило, где <подлежащее> находится слева от ::=, и заменим <подлежащее> в исходной цепочке на соответствующую цепочку, которая находится справа от ::=. Получим вывод:

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

Полный вывод одного предложения будет таким:

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

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

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

Последовательность цепочек α_1, α_2, …, α_n в этом случае называется выводом цепочки α_n из α_1**:** $\alpha_1 \Rightarrow^+ \alpha_n$.

Две последовательности связаны отношением $\Rightarrow^*$, когда вторая получается из первой применением некоторой последовательности продукции $\alpha_1 \Rightarrow^* \alpha_n$ тогда, и только тогда, когда $\alpha \Rightarrow^+ \alpha_n$. $\alpha_1 \Rightarrow^* \alpha_n$ означает, что цепочка α_n выводится из α_1 за **ноль** или более шагов.

На каждом шаге можно заменить любую метапеременную. В приведенном выше выводе всегда заменялся самый левый из них.

Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен *левый вывод* цепочки. Аналогично определяется правый вывод.

Рассматриваемая модель языка 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**)

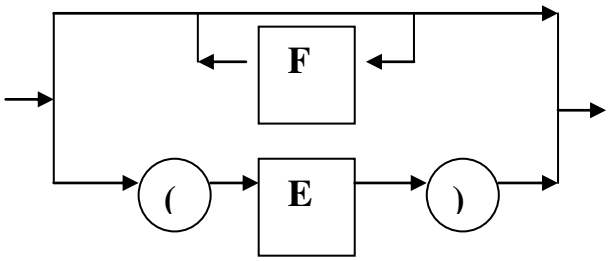


Рис. 1.3. Диаграммы Вирта

Грамматики. 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.

Синтаксис и семантика. Для того чтобы убрать семантику и перейти к формальному описанию модели языков закодируем заглавными (нетерминальные) символами S,A,B,C грамматику русского языка, а символами (терминальные) a,b,c слова рассматриваемого предложения:

S A B

Предложение → *подлежащее, группа сказуемого*

A a

подлежащее → он

B C D

группа сказуемого → *сказуемое, дополнение*

C b

сказуемое → смотрит

D c

дополнение → кино

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

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

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

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

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

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

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

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

Для обозначения терминальных символов 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 = (T={c,d}, V={B,S_0}, P, S_0), где P состоит из правил

S_0 → cBd

cB → ccBd

B → ε

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 $\Rightarrow$ cccddd. 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 над множеством алфавита T, которые выводимы из S_0 подстановкой правил из P.

Цепочка $\alpha \subset (T \cup V)^*$, для которой $S_0 \Rightarrow^* \alpha$ (то есть цепочка, 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 = (T= {0,1}, V = {A, S_0}, P_1, S_0), состоящая из правил P_1: S_0 → 0A1

0A → 00A1

A → ε

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 → 0S1 | 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 → 0S1 | ε

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.

Распознающие модели

Формального язык может быть распознан моделью отвечающей на вопрос, принадлежит ли данная цепочка символов языку. Например, принадлежит ли предложение “он смотрит кино” языку заданному рассмотренными грамматическими правилами? Для этого необходимо построить распознаватель.

Правила перехода δ описывают те последовательности символов, которые являются корректными, т.е. допустимыми предложениями языка.

Утверждение 1. Язык распознающей системы R_s эквивалентен языку порождающей системы G_s: $L(R_s) = L(G_s)$.

1.3. Классификация Хомского

Классификация Хомского вводит порождающие модели формальных языков. В книге классификация Хомского расширена эквивалентными распознающими моделями Таб. 1.1. Это дает представление об общности грамматик, и об эквивалентных им формальных автоматах. Приводятся примеры языков и грамматик.

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) $\alpha \to \beta, \mid \alpha \mid \leq \mid \beta \mid$ $\alpha \subset (T \cup V)^*$ $\beta \subset (T \cup V)^*$	$S_0 \to 0B0$ $B \to 1 \mid 0BC$ $C0 \to 100$ $C1 \to 1C$	$\{0^n1^n0^n \mid n \geq 1\}$	Машина Тьюринга с конечной лентой
2	Контекстно-свободная (КС) $A \to \alpha$ $\alpha \subset (T \cup V)^*$	$S_0 \to AS_0B \mid AB$ $A \to 0$ $B \to 1$ (нисходящий разбор) LL(k) грамматика (восходящий разбор) LR(k) грамматика $k = 0,1$	$\{0^n1^n \mid n \geq 1\}$	МП-автомат с магазинной памятью и РМП-автомат: Распознаватели на основе управляющих таблиц: LL(1) LR(0), LR(1) грамматики предшествования
3	Регулярная $A \to a$ $A \to aB$ $A \to \varepsilon$	$S_0 \to 0 \mid 0B$ $A \to 0B$ $B \to 1 \mid 1A$	$\{0(01)^n \mid n \geq 0\}$	Конечный автомат (КА) $\delta(A,a) = \{q_f\}$ $\delta(A,a) = \{B\}$ $\delta(A, \varepsilon) = \{A \mid B\}$

Классификация Хомского дает представление о 4 типах грамматик и об эквивалентных формальных автоматах, располагающихся по иерархии от 0 до 3 в порядке убывания их общности. Грамматики классифицируются по ограничениям накладываемым на правила вывода P, представляющие собой структурные паттерны формальной модели. В правилах вывода P большими латинскими буквами обозначают нетерминальные символы, маленькими латинскими буквами - терминальные.

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, например, $\Sigma = \{0,1\}$;

δ - конечное множество функций (правил) переходов. Функция перехода задается отображением

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

где Q - конечное множество подмножеств множества Q;

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

$F \subseteq Q$ - множество заключительных состояний.

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

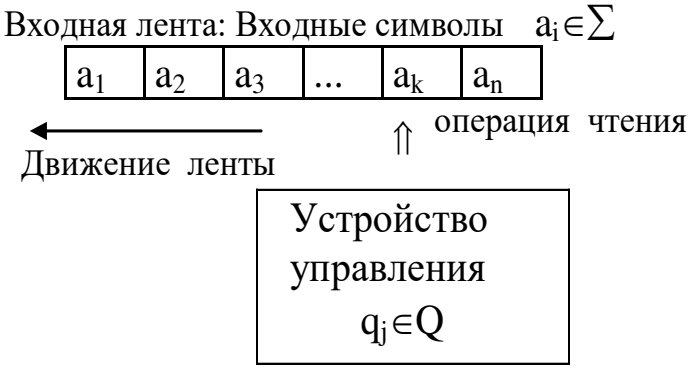


Рис. 2.1. Схема конечного автомата

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

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

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

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

Пусть {C} – множество конфигураций.

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

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

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

$L(KA) = \{\omega \in \sum^* \mid (q_0,\omega) \vdash^k (q_f, \varepsilon)$, где $q_f \in F\}$

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

1. Входные алфавиты конечного автомата $\sum$ и автоматной грамматики T совпадают, $\sum = T$.
2. $Q = V \cup \{q_f\}$, где q_f – заключительное состояние конечного автомата.
3. $q_0 = S_0$.

4. Функция переходов δ КА определяется следующим образом:

1. для каждого правила порождения вида $A \to a \in P$, $A \in V$, $a \in T$, построить функцию перехода $\delta(A,a) = \{q_f\}$, $A \in Q$, $a \in \Sigma$;
2. для каждого правила порождения вида $A \to aB \in P$, построить функцию перехода $\delta(A,a) = \{B\}$, $A,B \in Q$, $a \in \Sigma$;
3. для каждого правила порождения вида $A \to \varepsilon \in P$, $F = \{A, q_f\}$, $A \in Q$, в противном случае $F = \{q_f\}$;

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= $(\Sigma,Q,\delta, q_0, F)$ 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= $(\{0, 1\}, \{S_0, A, B, q_f\}, \delta, S_0, q_f)$

1. $\delta(S_0, 0) = \{A, q_f\}$
2. $\delta(A, 1) = \{B\}$
3. $\delta(B, 0) = \{A, q_f\}$

Пример конфигурации КА: $(S_0,01010) \vdash^1 (A,1010) \vdash^2 (B,010) \vdash^3 (A,10) \vdash^2 (B,0) \vdash^3 (q_f,\varepsilon)$

2.2. Способы задания конечных автоматов

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

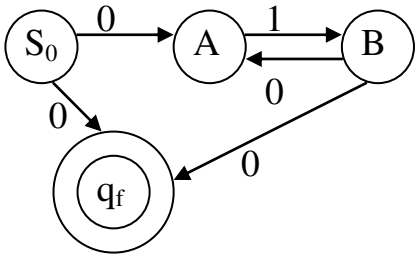


Рис. 2.2. Диаграмма переходов КА

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

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

Таб. 2.1. Таблица переходов для КА

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

Диаграммой переходов КА называется неупорядоченный граф, удовлетворяющий условиям:

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

2.3. Свойства регулярных языков: лемма о накачке

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

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

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

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

$\forall L \subseteq \Sigma^*$(regular(L) **=>** 1. л**юбой** язык $L \subseteq \Sigma^*$ регулярный **если**
($\exists p \geq 1$ 2. **существует** целое $p \geq 1$ такое что
($\forall w \in L$ (($|w| \geq p$) => 3. для **любой цепочки языка** $|w| \geq p$
($\exists x,y,z \in \Sigma^*$ ($w = xyz$ => 4. **найдется** $w = xyz$ такое что
($|y| \geq 1 \wedge |xy| \leq p \wedge i \geq 0 \wedge (xy^iz \in L)$) 5. y **повторяется** i раз
))
))
)
), на x и z ограничений не накладывается.

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

Доказательство леммы о накачке

Пусть автоматный язык L содержит бесконечное число цепочек и предположим, что L распознаётся детерминированным конечным автоматом

КА с n состояниями. Для проверки заключения леммы выберем произвольную цепочку α этого языка, которая имеет длину n.

Если конечный автомат КА распознаёт L, то цепочка α допускается этим автоматом, то есть в автомате КА существует путь длины n из начального в одно из заключительных состояний, помеченный символами цепочки α. Путь этот не может быть простым, он должен проходить ровно через n+1 состояние, в то время как автомат КА имеет n состояний. Это значит, что этот путь проходит по меньшей мере два раза через одно и то же состояние автомата КА, то есть на этом пути есть цикл с повторяющимся состоянием. Пусть это повторяющееся состояние q_k.

Разделим цепочку α на три части, так что α = xyz, где y — подцепочка, переводящая КА из состояния q_k опять в состояние q_k, и z — подцепочка, переводящая КА из состояния q_k в заключительное состояние. Заметим, что как x, так и z могут быть пустыми, но подцепочка y не может быть пустой. Но тогда очевидно, что автомат КА должен допускать также и цепочку xyyz, поскольку повторяющаяся подцепочка y снова проходит по циклическому пути из q_k в q_k, а также и цепочку xyyyz, и любую вида xyy...yz.

Это рассуждение и составляет доказательство леммы о накачке.

Для доказательства регулярности языка:

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

Свойство замкнутости регулярных языков, позволяет минимизировать построение автомата:

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

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

Утверждается, что язык L_{01} нерегулярен.

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

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

Пример 3. Нерегулярность языка $L = \{a^nb^n \mid n \geq 0\}$ 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 $w = a^p b^p$. По лемме о накачке, существует разбиение $w = xyz$, где $|xy| \leq p$, $|y| \geq 1$, такое что xy^iz принадлежит L для любого $i \geq 0$. Если допустить, что $|xy|=p$, а $|z|=p$, то xy — это первая часть w, состоящая из p последовательных экземпляров символа a.

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

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

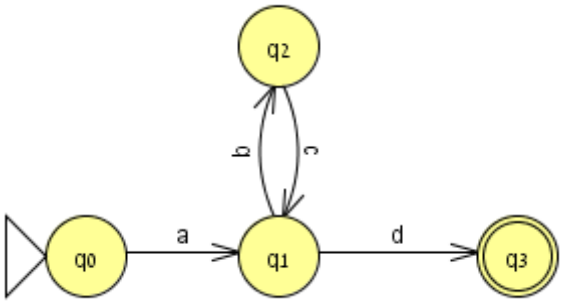


Рис. 2.3. Диаграмма переходов КА

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

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

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

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

Конфигурация автомата изменяется при такте срабавтвания и выполнении функции перехода δ, при этом считывается символ с входной ленты. По *эпсилон-переходу* символ не считывается с входной ленты, но конфигурация изменяется.

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

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

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

ε-переход позволяет объединить нескольких автоматов в один. Так,

например, два автомата, представленных на рис.2.5, можно соединить с помощью ε-переходов. В итоге получим автомат, представленный на рис.2.6.

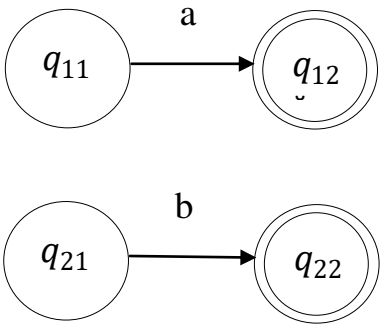


Рис. 2.5

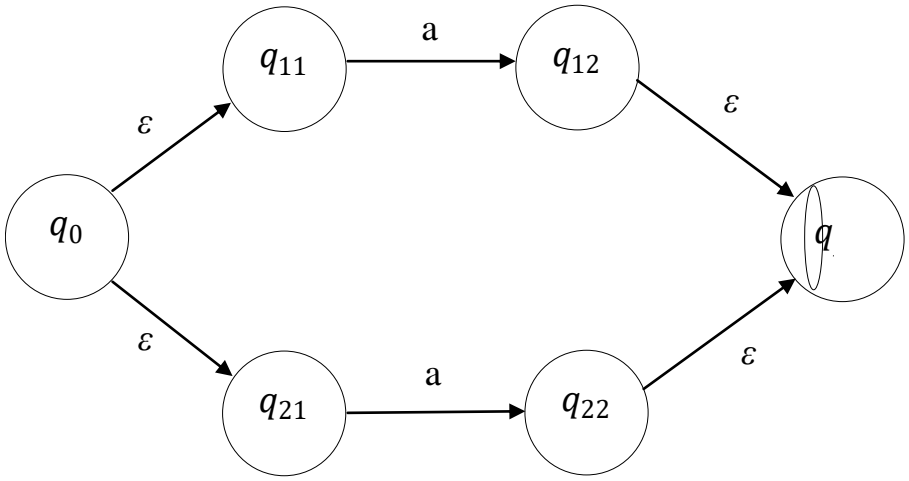


Рис. 2.6

Алгоритм. Построение ДКА из НКА: на основе таблицы переходов и очереди.
Таблица переходов:

В таблице переходов НКА каждая запись представляет собой множество состояний; в таблице переходов ДКА - единственное состояние.

Общая идея преобразования НКА в ДКА: каждому состоянию ДКА соответствует множество состояний НКА. ДКА использует свои состояния для отслеживания всех возможных состояний, в которых НКА может находиться после чтения очередного входного символа.

После чтения входной цепочки $a_1a_2..a$ ДКА находится в состоянии, подмножества состояний НКА, достижимых из начального состояния НКА по пути $a_1a_2..a$. Наихудший случай, встречающийся крайне редко, когда количество состояний ДКА экспоненциально зависит от количества состояний НКА.

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

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

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

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

Рис. 2.7. Операции над состояниями НКА

НКА может быть в любом из состояний множества *ε-closure(q_0),* где q_0 - начальное состояние НКА. Предположим, что состояния множества *Q*, и только они, достижимы из q_0 после считывания входных символов, *a* - следующий входной символ. НКА может перейти в любое состояние из множества *move(Q, a)*. Совершив из этих состояний все возможные *ε*-переходы, НКА после обработки *a* может быть в любом состоянии из *ε-closure(move(Q, a)).*

Алгоритм. Определить множество состояний *Dstates* ДКА и таблица переходов *Dtran* :

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

2. q_0 = *ε-closure(q_0).*

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

Вход: НКА = (Q, ∑, δ, q_0, F).
Выход: ДКА = (Q', ∑, δ', q_0', F').

Q′= ∅
δ′ = ∅
q_0' = q_0
newStates = ∅ – множество состояний, достижимых из текущих состояний по терминальному символу, ε-переходам.
currStates = ε-closure(q_0) – множество текущих состояний.
Q′= Q′ ∪ currStates
foreach (a ∈ ∑)
    newStates = ε-closure(move(currStates, a))
    if (newStates ∉ ∅)

δ′ = δ′ ∪ δ(currStates, a)
if (newStates ∉ Q′)
Q′= Q′ ∪ newStates
Dtran[currStates, a] = newStates
end
end
end foreach

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

Вычисление ε-closure(Q) является типичным процессом поиска графа для узлов, достижимых из данного множества узлов. В этом случае состояния Q представляют данное множество узлов, а граф состоит только из дуг НКА, помеченных ε.

Алгоритм вычисления ε-closure(Q)использует множество nextStates для хранения состояний, дуги которых не были проверены на наличие ε-переходов.

ReachableStates – множество достижимых состояний Q'.

Вход: НКА (Q, ∑, δ, q_0, F), currStates
Выход: ReachableStates
ReachableStates = ∅
ε-closure(currStates, ReachableStates)
nextStates = ∅
foreach (q ∈ currState)
foreach (δ'(q, ε)={q'} ∈ δ)
nextStates = nextStates ∪ q'
end foreach
if (q ∉ ReachableStates)
ReachableStates = ReachableStates ∪ q
end
if (nextStates ∉ ∅)
foreach (n ∈ nextStates)
ReachableStates = ReachableStates ∪ n
end foreach
end
end foreach
if (nextStates ∉ ∅)
ε-closure(nextStates, ReachableStates)
end

Пример. Дан конечный автомат с Q = {1, 2, 3, 4, 5, 6, 7, 8, 9}

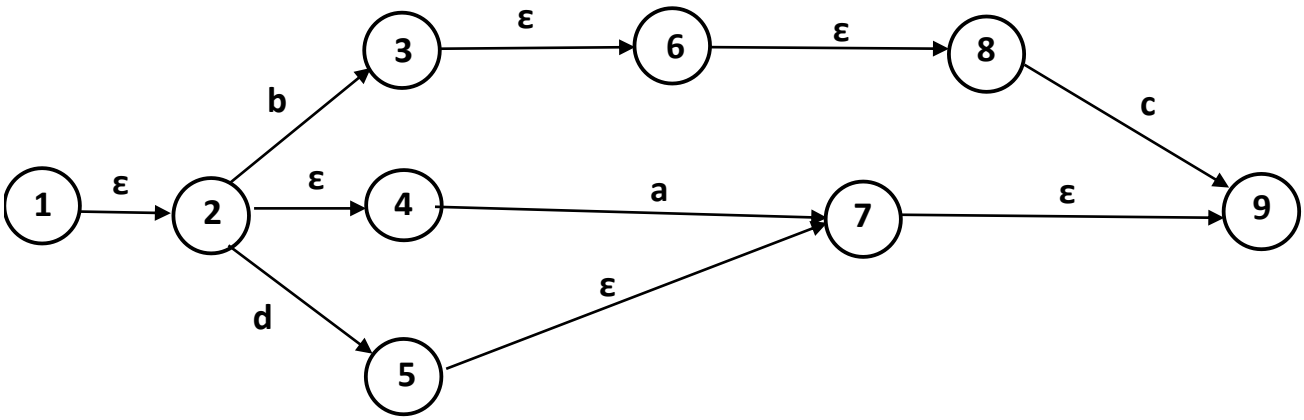


рис. 2.8

Получим ε-closure(Q) = {1, 2, 4, 6, 7, 8, 9}.

Определение заключительных состояний ДКА
Вход: Q′, F.
Выход: F′
F' = ∅
foreach (f ∈ F)
    foreach (q' ∈ Q')
        if (q' ∈ f)
            F' = F' ∪ f
        end
    end foreach
end foreach
Пример. Задан недетерминированный конечный автомат ∑ = {0, 1, 2, ,}:

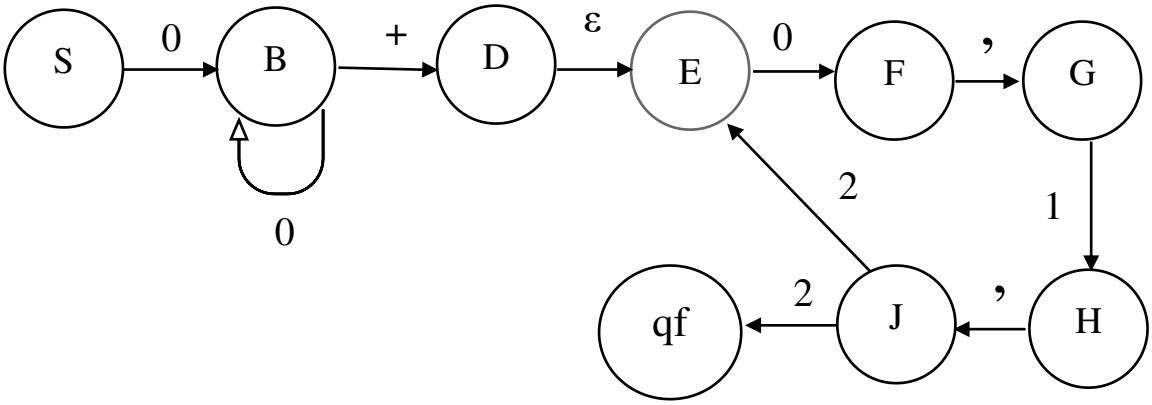


Рис. 2.9

q_0 эквивалентного ДКА = ε-closure(S), т.е. S' = {S}, именно эти состояния достижимы из состояния S путями, в которых каждая дуга помечена ε. Заметим, что путь может и не иметь дуги, так что состояние S также достигается из состояния S, а значит, входит в искомое множество.

Вычислим ε-closure(move(S', 0)). move(S', 0), множество состояний, имеющих переходы по символу 0 для элементов множества S'. Из состояний S только S имеет такой переход к состоянию B, ε-closure(move({S},0)) = ε-closure({B}) = {B}, Dtran[S', 0] = B'.

Для множества B: ε-closure(move(B', 0)). Вычислим move(B', 0), множество состояний, имеющих переходы по символу 0 для элементов множества B'. Из состояний B только B имеет такой переход к состоянию B,

ε-closure(move({B},0)) = ε-closure({B}) = {B}, Dtran[B', 0] = B'

ε-closure(move(B', +))

Вычислим move(B', +), множество состояний, имеющих переходы по символу + для элементов множества B'. Из состояний B только B имеет такой переход к состоянию D,

ε-closure(move({B},+)) = ε-closure({D}) = {D, E}, Dtran[B', +] = D' E'.

ε-closure(move(D'E', 0)). Вычислим move(D'E', 0), множество состояний, имеющих переходы по символу 0 для элементов множества D'E'. ε-closure(move({D, E},0)) = ε-closure({F}) = {F}. Таким образом, Dtran[D'E', 0] = F

Продолжая этот процесс, в конечном итоге все множества-состояния ДКА окажутся помеченными. Имеем восемь множеств состояний:

S' = {S} B' = {B} D'E' = {D,E} F' = {F} G' = {G} H' = {H} J' = {J} E'qf ' = {qf, E}, где S' начальное состояние, qf ' - единственное заключительное состояние.

Таблица переходов Dtran

Состояние	Входной символ				
	0	1	2	,	+
S'	B'				
B'	B'				D'E'
D'E'	F'				
F'				G'	
G'		H'			
H'				J'	
J'			qf '		
E'qf '	F'				

На основе очереди:

Шаг 1. Помещаем в очередь *Q* множество, состоящее только из стартовой вершины.

Шаг 2. Пока очередь не пуста выполняем следующие действия:

- Берем из очереди множество *q*
- Для всех $c \in \Sigma$ посмотрим в какое состояние ведет переход по символу ***c*** из каждого состояния в *q*. Полученное множество состояний положим в очередь *Q* если его не бвло ранее в очереди. Каждое такое множество в итоговом ДКА будет отдельной вершиной, в которую будут вести переходы по соответствующим символам.
- Если в множестве *q* хотя бы одна из вершин была терминальной в НКА, то соответствующая данному множеству вершина в ДКА также будет терминальной.

Вход: НКА = (Q, Σ, δ, q_0, F).
Выход: ДКА = (Q', Σ, δ', q_0', F').

```
        Q′= ∅
        δ′ = ∅
        q0' = q0
        Queue = ∅ // Очередь для обхода уникальных состояний
        currStates = ∅ // Мн-во состояний из очереди
        newStates = ∅ // промежуточное мн-во состояний из текущего по символу
        Queue = Queue ∪ q0
while (Queue ∉ ∅)
        currStates = Queue.pop()
        currStates = currStates ∪ ε-closure(currStates)
        foreach (a ∈ Σ)
                newStates = move (currStates, a)
                if (ε-closure(newStates) ∉ Q')
                        Queue = Queue ∪ newStates
                        Q' = Q' ∪ ε-closure(newStates)
                end
                if (newStates ∉ ∅)
                        δ′ = δ′ ∪ δ(currStates, a)
                end
        end foreach
end while
```

Пример. Задан недетерминированный конечный автомат Σ = {0, 1, 2, ,} (рис.2.9).
Шаги алгоритма для данного примера:
delta δ’ – добавленный переход
Шаг итерации: 0
Q’: ∅
Queue: S
Cur: S
delta δ’: δ(S, 0)

Шаг итерации: 1
Q’: S
Queue: B
Cur: B
delta δ’: δ(B, +), δ(B, 0)

Шаг итерации: 2
Q’: S, B
Queue: D
Cur: DE
delta δ’: δ(DE, 0)

Шаг итерации: 3
Q’: S, B, DE
Queue: F
Cur: F
delta δ’: δ(F, ,)

Шаг итерации: 4
Q’: S, B, DE, F
Queue: G
Cur: G
delta δ’: δ(G, 1)

Шаг итерации: 5
Q’: S, B, DE, F, G
Queue: H
Cur: H
delta δ’: δ(H, ,)

Шаг итерации: 6
Q’: S, B, DE, F, G, H
Queue: J
Cur: J
delta δ’: δ(J, 2) – переход в состояние qfE

Шаг итерации: 7
Q’: S, B, DE, F, G, H, J, qfE
Queue: qfE
Cur: F
delta δ’: δ(qfE, 0) – переход в состояние F

Шаг итерации: 8 // F не добавляем, так как оно уже есть в мно-ве
Q’: S, B, DE, F, G, H, J, qfE
Queue: ∅
Cur: ∅
delta δ’: -

Очередь пуста. Алгоритм завершён.
Результат:
Q’: S, B, DE, F, G, H, J, qfE
δ’:
δ(S, 0)
δ(B, 0)
δ(B, +)
δ(DE, 0)
δ(F, ,)
δ(G, 1)

δ(H, ,)
δ(J, 2)
δ(qfE, 0)
F': qfE
Оба алгоритма выдали одинаковый результат.
Граф переходов ДКА имеет вид:

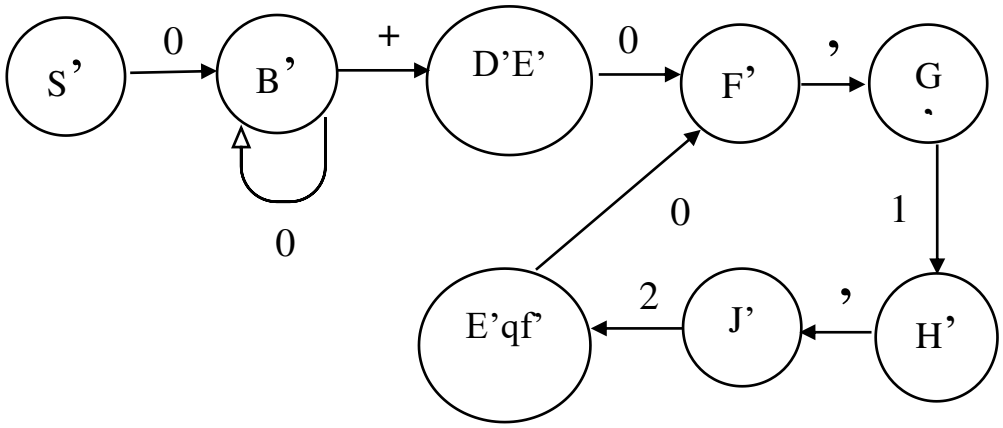


рис. 2.10

2.5. Составные конечные автоматы и регулярные языки

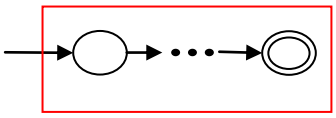
Для двух регулярных языков L_1 и L_2 справедливо следующее:

Операция	**Доказательство замкнутости относительно операции**
Объединение: $L_1 \cup L_2$	$L_1 \cup L_2$ является регулярным по определению регулярных языков.
Конкатенация: L_1L_2	L1L2 является регулярным по определению регулярных языков.
Замыкание Клини: L_1^*	L_1^* является регулярным по определению регулярных языков
Реверс: L_1^R	Рассмотрим НКА с ε-переходами $A=\langle \Sigma,Q1,s',\{s1\},\delta'1\rangle$, где $\delta'1(v,c)=\{u\|\delta1(u,c)=v\}$; $\delta'1(s',\varepsilon)=\{Ti\}$. Если в исходном автомате путь по α из s1 приводил в терминальное состояние, то в новом автомате существует путь по α из этого терминального состояния в s1 (и наоборот). Следовательно, этот автомат распознает в точности развернутые слова языка L_1. Тогда язык L_1^R — регулярный.
Дополнение: $\underline{L_1}$	Рассмотрим автомат $A=\langle\Sigma,Q1,s1,Q1\setminus T1,\delta1\rangle$, то есть

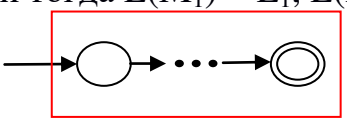
	автомат А, в котором терминальные и нетерминальные состояния инвертированы (при таком построении следует помнить, что если в исходном автомате было опущено состояние, его нужно явно добавить и сделать допускающим.) Очевидно, А допускает те и только те слова, которые не допускает автомат А, а значит, задаёт язык $\underline{L}_1$. Таким образом, $\underline{L}_1$— регулярный.
Пересечение: $L_1 \cap L_2$	L1 ∩ L2= дополнение($\underline{L}_1 \cup \underline{L}_1$). Тогда L1∩ L2 — регулярный. Автомат для пересечения языков строится явно произведением автоматов.

Результат операций есть регулярный язык. Регулярные языки замкнуты над этими операциями.

Пусть L_1 и L_2 регулярные языки тогда $L(M_1) = L_1$, $L(M_2) = L_2$

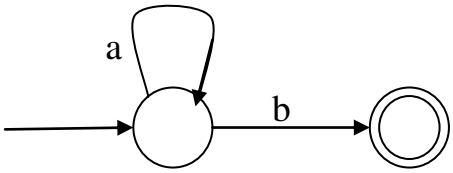


НКА
M_1



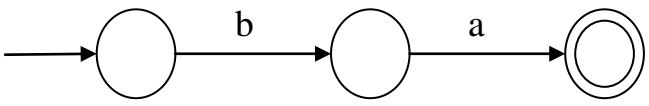
НКА
M_2

Пример: $L_1=\{a^nb\}$



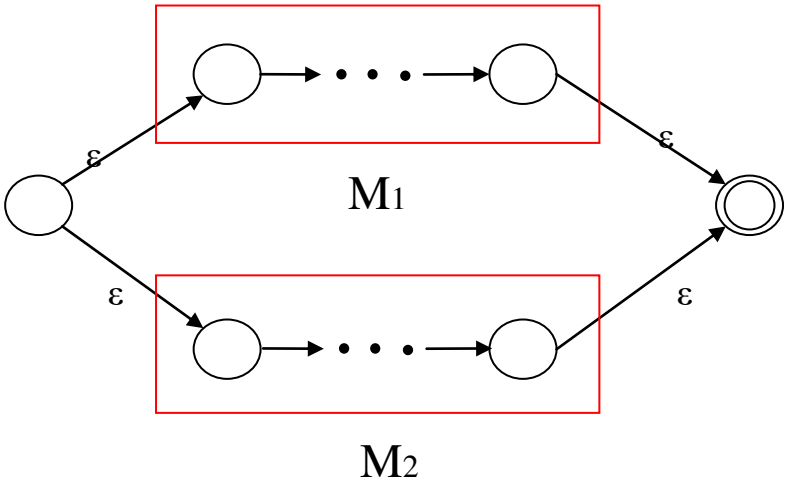
M_1

$L_2=\{ba\}$

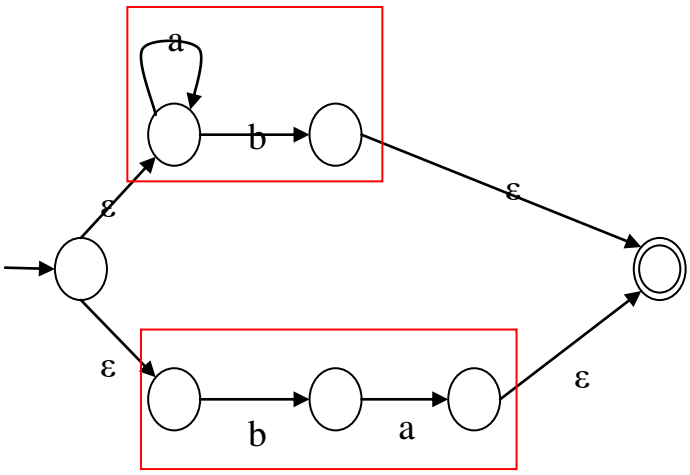


M_2

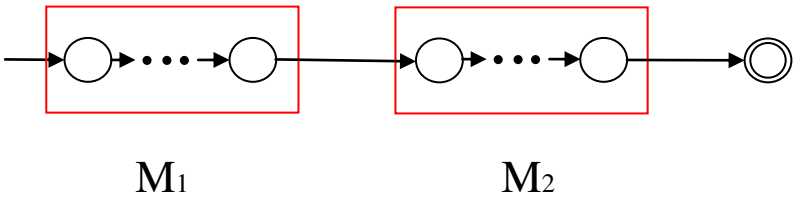
1. Объединение Недетерминированный конечный автомат L_1 U L_2



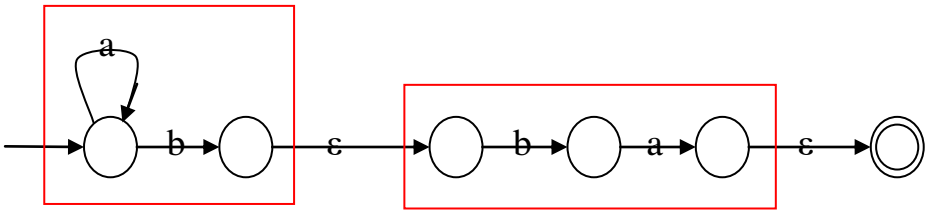
Недетерминированный конечный автомат L_1 U L_2 = {a^nb} U {ba}



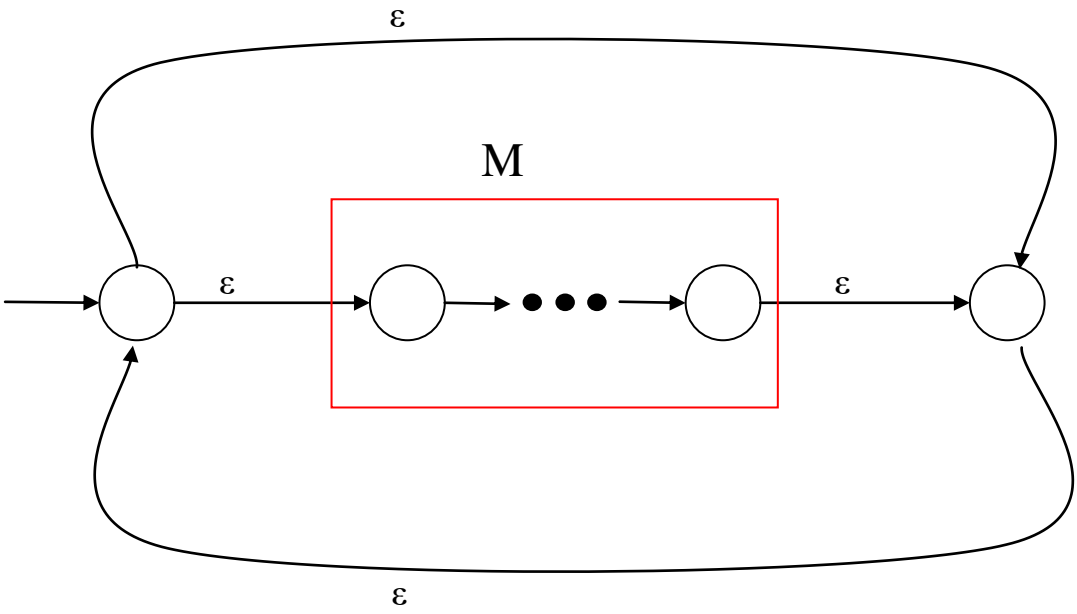
2. Конкатенация НКА для L_1L_2



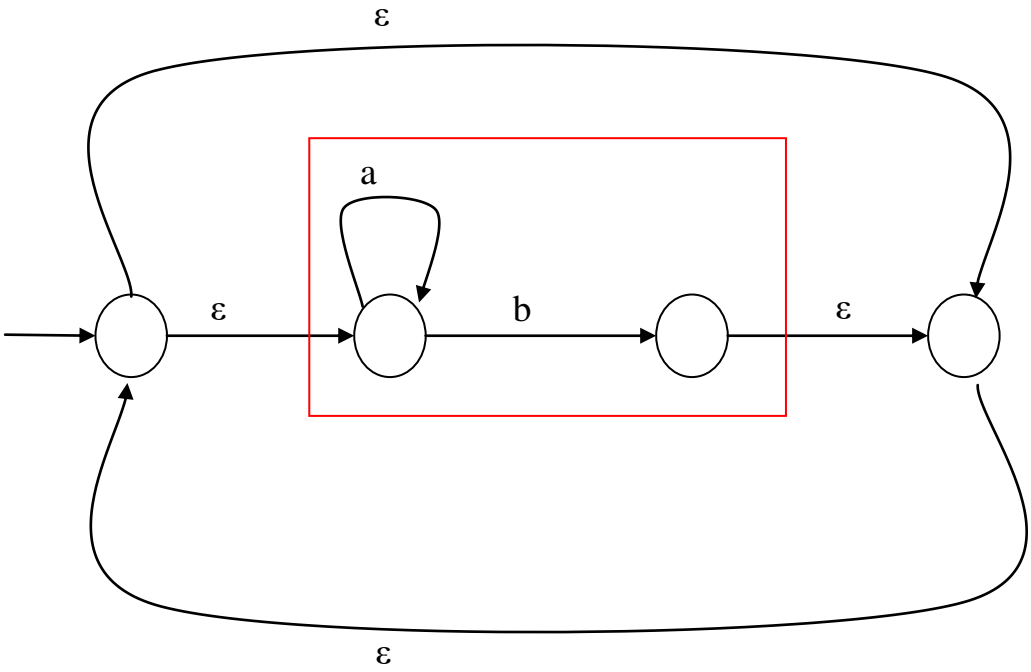
L_1L_2 = {a^nb}{ba} = { a^nbba}



3. Замыкание Клини НКА для L_1*



НКА $L_1^* = \{a^n b\}^*$



4. Реверс НКА для L_1^R
1. Отменить все переходы
2. Сделать начальное состояние конечным и наоборот

$L_1 = \{a^n b\}$, $L_1^R = \{b a^n\}$

5. Дополнение:
   1. Пусть КА допускает L_1
   2. Сделать конечные состояния не конечными и наоборот

$\bar{L}_1 = \{a,b\}^* - \{a^n b\}$

6. Пересечение $L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$

Алгоритм 1. Конкатенация двух автоматов.
Вход: $КА_1 = (Q_1, \Sigma_1, \Delta_1, q_{01}, F_1 = \{qf_{1.1}, \ldots, qf_{1.n}\})$,
$КА_2 = (Q_2, \Sigma_2, \Delta_2, q_{02}, F_2 = \{qf_{2.1}, \ldots, qf_{2.n}\})$
Выход: $КА = КА_1 + КА_1 = (Q, \Sigma, \Delta, q_0, F)$

```
Q = Q1 ∪ Q2
Σ = Σ1 ∪ Σ2
F = F2
q0 = q01
Δ = Ø
foreach (σ ∈ Δ 1)
  Δ = Δ ∪ σ
end foreach

foreach (σ(q,a) = A ∈ Δ 2)
  Δ = Δ ∪  σ(q,a) = A
  if (q = q02)
     foreach (qf1 ∈ F1)
       Δ = Δ ∪  σ(qf1, a) = A  //
     end foreach
   end
end foreach
```

Алгоритм 2. Конкатенация n конечных автоматов
Вход: $FS = \{КА_1, КА_2, \ldots, КА_n\}$
Выход: $КА = КА_1КА_2 \ldots КА_n$

```
КА = КА1
foreach( КАi ∈ FS; i > 1 )
   КА = КАКАi // применяем алгоритм конкатенации 2х автоматов
end foreach
```

Алгоритм 3. Объединение 2х автоматов
Вход: $КА_1 = (Q_1, \Sigma_1, \Delta_1, q_{01}, F_1)$, $КА_2 = (Q_2, \Sigma_2, \Delta_2, q_{02}, F_2)$
Выход: $КА = КА_1КА_2 = (Q, \Sigma, \Delta, q_0, F)$

```
q0 = S  // S - новый начальный нетерминал
```

$Q = Q_1 \cup Q_2 \cup q_0$
$\Sigma = \Sigma_1 \cup \Sigma_2$
$F = F_1 \cup F_2$
$\Delta = \emptyset$
$\Delta = \Delta \cup \{\sigma(q_0, \varepsilon) = q_{01}, \sigma(q_0, \varepsilon) = q_{02}\}$
$\Delta = \Delta \cup \Delta_1 \cup \Delta_2$
$КА = (Q, \Sigma, \Delta, q_0, F)$

Алгоритм 4. Объединение n конечных автоматов
Вход: $FS = \{КА_1, КА_2, \ldots, КА_n\}$
Выход: $КА = КА_1 \cup КА_2 \cup \ldots \cup КА_n$

$КА = КА_1$
foreach($КА_i \in FS; i > 1$)
  $КА = КА \cup' КА_i$ // применяем алгоритм объединения 2х автоматов
end foreach

Примеры использования составных автоматов

$L = \{0(000)^*(0+1)\ \omega_1 \mid \omega_1 \in \{0,1\}^*\}$
$L_1 = \{0(000)^*\}$
$(000)^* = \{000\ 000000\ 000000000\ \ldots\}$

$L_{1.1} \in L_1 = \{0000, 0000000, 0000000000, 000000000000\}$
$L_2 = \{(0+1)\}$
$L_3 = \{\ \omega_1 \in \{0,1\}^*\}$

$\{0,1\}^* = \{\{\varepsilon\}, \{0\}, \{1\}, \{00\}, \{10\}, \{01\}, \{11\}, \ldots\}$
$L_{3.1} \in L_3 = \{\ \varepsilon, 0, 1, 01\}$

Правила грамматики языка L_1:

1. $S_{01} \rightarrow 0A_1$
2. $A_1 \rightarrow 0B_1$
3. $B_1 \rightarrow 0C_1$
4. $C_1 \rightarrow 0A_1$
5. $C_1 \rightarrow 0$

$G_1 = (\{0\},\{S_{01}, A_1, B_1, C_1\}, P = \{p_1, p_2, p_3, p_4, p_5\}, S_{01})$

Правила грамматики языка L_2:

1. $S_{02} \rightarrow (A_2$
2. $A_2 \rightarrow 0B_2$
3. $B_2 \rightarrow +C_2$
4. $C_2 \rightarrow 1D_2$
5. $D_2 \rightarrow 0$

$G_2 = (\{0,1,+,(,)\},\{S_{02}, A_2, B_2, C_2\}, P = \{p_1, p_2, p_3, p_4, p_5\}, S_{02})$

Правила грамматики языка L_3 (замыкание Клини):

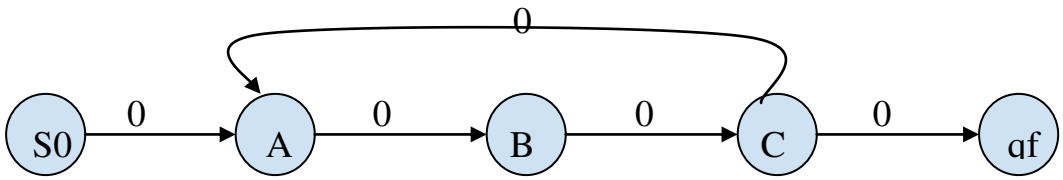
1. $S_{03} \rightarrow \varepsilon$

2. $S_{03} \rightarrow 0$
3. $S_{03} \rightarrow 1$
4. $S_{03} \rightarrow 0S_{03}$
5. $S_{03} \rightarrow 1S_{03}$

$G_2 = (\{0,1\},\{S_{03}\}, P = \{p_1, p_2, p_3, p_4, p_5\}, S_{03})$
$KA_1 = (\{S_{01}, A_1, B_1, C_1\}, \{0\}, \sigma, S_{01}, \{q_{f1}\})$

1. $\sigma_1(S_{01}, 0) = \{ A_1\}$
2. $\sigma_1(A_1, 0) = \{ B_1\}$
3. $\sigma_1(B_1, 0) = \{ C_1\}$
4. $\sigma_1(C_1, 0) = \{ A_1, q_{f1}\}$

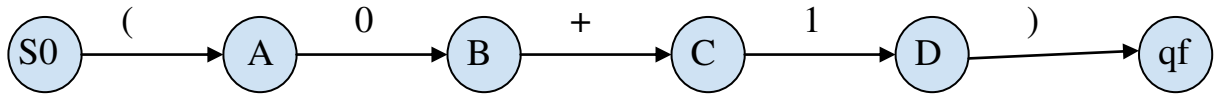
σ_1	0
S_{01}	$\{A_1\}$
A_1	$\{B_1\}$
B_1	$\{C_1\}$
C_1	$\{B_1, q_{f1}\}$
q_{f1}	



$KA_2 = (\{S_{02}, A_2, B_2, C_2, D_2\}, \{0,1,(,),+\}, \sigma, S_{02}, \{q_{f2}\})$

1. $\sigma_2(S_{02}, () = \{ A_2\}$
2. $\sigma_2(A_2, 0) = \{ B_2\}$
3. $\sigma_2(B_2, +) = \{ C_2\}$
4. $\sigma_2(C_2, 1) = \{ D_2\}$
5. $\sigma_2(D_2,)) = \{ q_{f2}\}$

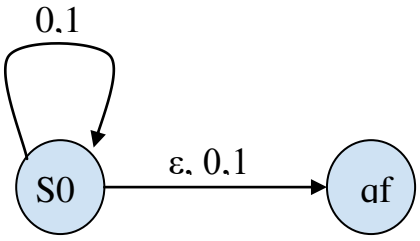
σ_2	0	1	(	)	+
S_{02}			$\{A_2\}$		
A_2	$\{B_2\}$				
B_2					$\{C_2\}$
C_2		$\{D_2\}$			
D_2				$\{q_{f2}\}$	
q_{f2}					



$KA_3 = (\{S_{03}\}, \{0,1\}, \sigma, S_{03}, \{q_{f3}\})$

1. $\sigma_3(S_{03}, \varepsilon) = \{ q_{f3}\}$
2. $\sigma_3(S_{03}, 0) = \{ S_{03}, q_{f3}\}$
3. $\sigma_3(S_{03}, 1) = \{ S_{03}, q_{f3}\}$

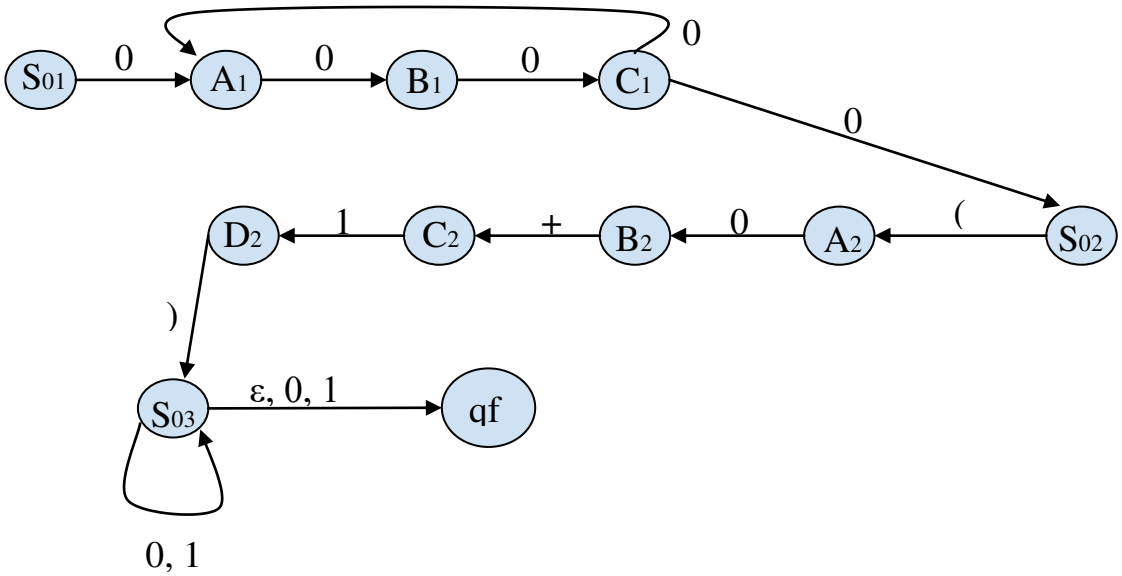
σ_2	0	1	ε
S_{03}	$\{S_{03}, q_{f3}\}$	$\{S_{03}, q_{f3}\}$	$\{ q_{f3}\}$
q_{f3}			



1. $L_1L_2L_3$

$KA_{L1L2L3} = KA_L = (\{S_{01}, A_1, B_1, C_1, S_{02}, A_2, B_2, C_2, D_2, S_{03}\}, \{0,1,(,),+\}, \sigma, S_{01}, \{q_{f3}\})$

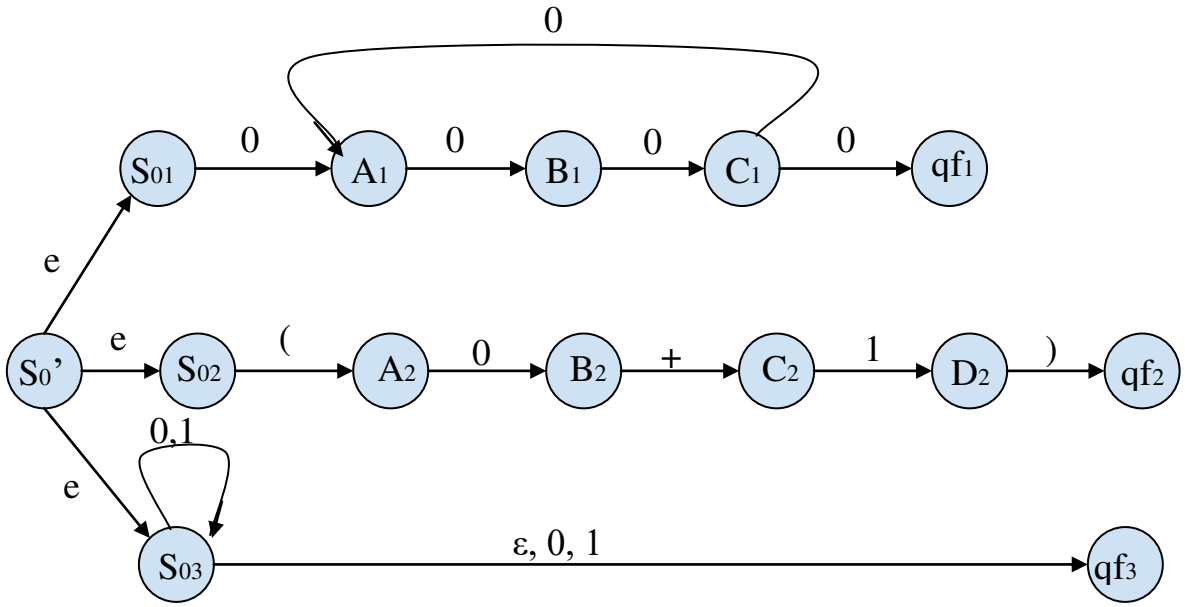
σ	*0*	*1*	*(*	*)*	*+*	ε
S_{01}	$\{ A_1\}$					
A_1	$\{ B_1\}$					
B_1	$\{ C_1\}$					
C_1	$\{ A_1, S_{02}\}$					
S_{02}			$\{ A_2\}$			
A_2	$\{B_2\}$					
B_2					$\{C_2\}$	
C_2		$\{D_2\}$				
D_2				$\{S_{03}\}$		
S_{03}	$\{S_{03}, q_{f3}\}$	$\{S_{03}, q_{f3}\}$				$\{q_{f3}\}$
q_{f3}						



$L_1 \cup L_2 \cup L_3 = (\{S_0, S_{01}, A_1, B_1, C_1, S_{02}, A_2, B_2, C_2, D_2, S_{03}\}, \{0,1,(,),+\}, \sigma, S_0, \{ q_{f1}, q_{f2}, q_{f3}\})$

2.

Σ	*0*	*1*	*(*	*)*	*+*	ε
S_0						$\{ S_{01}, S_{02}, S_{03}\}$
S_{01}	$\{ A_1\}$					
A_1	$\{ B_1\}$					
B_1	$\{ C_1\}$					
C_1	$\{ A_1, q_{f1}\}$					
q_{f1}						
S_{02}			$\{ A_2\}$			
A_2	$\{ B_2\}$					
B_2					$\{ C_2\}$	
C_2		$\{ D_2\}$				
D_2				$\{ q_{f2}\}$		
q_{f2}						
S_{03}	$\{ S_{03}, q_{f3}\}$	$\{ S_{03}, q_{f3}\}$				$\{ q_{f3}\}$
q_{f3}						



2.6. Язык регулярных выражений на основе паттернов

Язык регулярных выражений на основе паттернов - это узкоспециализированный язык программирования, например для обработки текста. Примерами такого языка являются язык паттернов встроенный в C# .NET [1], Python [2] и другие языки программирования. Паттерны регулярных выражений определяются специальным синтаксисом или языком, который совместим с регулярными выражениями Perl 5 [3] и добавляет некоторые дополнительные функции. Этот язык относительно невелик и ограничен, поэтому не все возможные задачи обработки строк могут быть выполнены с использованием регулярных выражений.

Паттерн язык предоставляет мощный, гибкий и эффективный метод обработки текста. Нотация сопоставления текста с паттерном позволяет быстро анализировать большие объемы текста, чтобы:

- Найти определенные паттерны символов.
- Проверить текст, чтобы убедиться, что он соответствует предварительно определенному паттерну (например, адресу электронной почты).
- Извлечь, редактировать, заменить или удалить текстовые подстроки.
- Добавить извлеченные строки в коллекцию, чтобы сгенерировать отчет.

Примеры регулярных выражений

Поддерживается синтаксис немного отличный от PCRE (Perl Compatible Regular Expressions) [3]. Регулярные выражения по умолчанию чувствительны к регистру.

Следующие примеры иллюстрируют использование и построение простых регулярных выражений. Каждый пример включает тип текста для сопоставления, одно или несколько регулярных выражений, соответствующих этому тексту, и примечания, поясняющие использование **специальных** символов и форматирование.

1. Точное совпадение с фразой.
2. Сопоставьте слово или фразу в списке

3. Искажение слова.
4. Совпадение с адресом электронной почты из определенного домена
5. Совпадение с IP-адресом в диапазоне
6. Соответствие буквенно-цифровому формату
7. Действительный номер мобильного телефона..
8. Проверка пользовательского ввода
9. Разбивка строки на основе патерна

1. Точное совпадение с фразой

Пример	Совпадение с фразой "stocks tips"
1 pattern	**(\W|^)stock\stips(\W|$)**
2 pattern	**(\W|^)stock\s{0,3}tips(\W|$)**
3 pattern	**(\W|^)stock\s{0,3}tip(s){0,1}(\W|$)**
Примечания	• \W соответствует любому символу, который не является буквой, цифрой или символом подчеркивания. Это предотвращает регулярное выражение из совпадающих символов до или после фразы. • В примере 2 \s соответствует символу пробела, а {0,3} указывает, что от 0 до 3 • между словами stock и tip могут быть пробелы. • ^ соответствует звездочке новой строки. Позволяет регулярному выражению соответствовать фразе, если она появляется в начале строки без символов перед ним. • $ соответствует концу строки. Позволяет регулярному выражению соответствовать фразе, если она появляется в конце строки без символов после него. • В примере 3 (s) соответствует букве s, а {0,1} указывает, что буква может встречаются 0 или 1 раз после слова tip. Следовательно, регулярное выражение соответствует stock tip и stock tips. • Кроме того, вы можете использовать символ ? вместо {0,1}
	Создайте шаблон для слова, которое начинается с буквы «M».

pattern 4	`string pattern = @"\b[M]\w+";`
	`string authors = "Mahesh Chand, Raj Kumar, Mike Gold, Allen O'Neill, Ma";`
	`Regex rg = new Regex(pattern)`
	`// Получить все совпадения`
	`MatchCollection matchedAuthors = rg.Matches(authors);`
	`// Распечатать всех подходящих авторов`
	`for (int count = 0; count < matchedAuthors.Count;count++)`
	`Console.WriteLine(matchedAuthors[count].Value);`

2. Сопоставьте слово или фразу в списке

пример	**Сопоставьте слово или фразу из следующего списка:**

	• baloney • darn • drat • fooey • gosh darnit • heck
pattern	**(?i)(\W|^)(baloney|darn|drat|fooey|gosh\sdarnit|heck)(\W|$)**
Примечания	• (...) группирует все слова таким образом, что класс символов \W применяется ко всем словav в скобках. • (?i) делает соответствие содержимого нечувствительным к регистру. • \W соответствует любому символу, который не является буквой, цифрой или символом подчеркивания. Он не позволяет регулярному выражению сопоставлять символы до или после слов или фраз в списке. • ^ соответствует новой строкt. Позволяет регулярному выражению сопоставлять слово, если оно появляется в начале строки без символов перед ним. • $ соответствует концу строки. Позволяет регулярному выражению сопоставлять слово, если оно находится в конце строки, без символов после него • | указывает «или», поэтому регулярное выражение соответствует любому слову в списке. • \s соответствует пробелу. Используйте этот символ для разделения слов во фразе.

3. Искажение слова: сопоставление слов с разными вариантами написания, включая специальные символы

Пример	Слово "гравий" искажено. Искажение используют спамеры, например:
	gra@via **grav1ia** **gr@av1ia** **gra@via**
pattern	g[i!1][r@]a[vi@a]
	• \W не включен, так что другие символы могут стоять до или после любого из вариантов "гравий". Например, регулярное выражение по-прежнему соответствует "гравий" в следующем тексте: **gravia!! or ***gravia***

4. Совпадение с адресом электронной почты из определенного домена. Проверка нескольких писем

Пример	Совпадение с адресом электронной почты из доменов

	yahoo.com, hotmail.com и gmail.com
1 pattern	**(\W|^)[\w.\-]{0,25}@(yahoo|hotmail|gmail)\.com(\W|$)**
Примечания	• \W соответствует любому символу, который не является буквой, цифрой или символом подчеркивания. Это предотвращает регулярное выражение из совпадающих символов до или после адреса электронной почты. • ^ соответствует новой строке. Позволяет регулярному выражению сопоставляться с адресом, если он появляется в начале строки без символов перед ним. • $ соответствует концу строки. Позволяет регулярному выражению сопоставляться с адресом, если он находится в конце строки без символов после него. • [\w.\-] соответствует любому символу слова (a-z, A-Z, 0-9 или знак подчеркивания), точке или дефису. Это наиболее часто используемые допустимые символы в первой части адреса электронной почты. Символ \- (обозначающий дефис) должен стоять последним в списке символов в квадратных скобках. • Знак \ перед тире и точкой «экранирует» эти символы, т. е. указывает, что тире и точка сами по себе не являются специальными символами регулярных выражений. Нет необходимости экранировать точку в квадратных скобках. • {0,25} указывает, что от 0 до 25 символов в предыдущем наборе символов могут стоять перед символом @. Настройка электронной почты соответствия содержания поддерживает сопоставление до 25 символов для каждого набора символов в регулярном выражении. • Форматирование (...) группирует домены, а | символ, который их разделяет, указывает на «или».
	Проверка нескольких писем
2 pattern	**^((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[;]{0,1}\s*)+$**
	разделение писем с помощью разделителя ';'
3 pattern	**^((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+$**
	Если вы хотите использовать разделитель ',' используйте этот
4 pattern	**^((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[;,.]{0,1}\s*)+$**
	и если вы хотите использовать оба разделителя ',' и ';'

5.Совпадение с IP-адресом в диапазоне

пример	Соответствует IP-адресу в диапазоне от 192.168.1.0 до 192.168.1.255.

1 pattern	**192\.168\.1\.**
2 pattern	**192\.168\.1\.\d{1,3}**
Примечания	• Знак \ перед каждой точкой «убегает» от точки, т. е. указывает, что период сам по себе не является специальным символом регулярного выражения. • В примере 1 за последней точкой не следует никаких символов, поэтому регулярное выражение соответствует любому IP-адресу, начинающемуся с 192.168.1., независимо от следующего за ним числа. • В примере 2 \d соответствует любой цифре от 0 до 9 после последней точки, а {1,3} указывает, что цифры от 1 до 3 могут появляться после этой последней точки. В этом случае регулярное выражение соответствует любому полному IP-адресу, начинающемуся с 192.168.1. Это регулярное выражение также соответствует недопустимым IP-адресам, таким как 192.168.1.999.

6. Соответствие буквенно-цифровому формату. Замена нескольких пробелов, специальные символы. Разделить строку по символу.

Пример	Сопоставьте номера заказов на покупку. Номер заказа имеет различные форматы, например:
	• PO nn-nnnnn • PO nn-nnnn • PO# nn nnnn • PO#nn-nnnn • PO nnnnnn
1 pattern	**(\W|^)PO[#\-]{0,1}\s{0,1}\d{2}[\s-]{0,1}\d{4}(\W|$)**
	Замена нескольких пробелов

2 pattern	`string badString = "a string with ton of white space.";`
	`string CleanedString = Regex.Replace(badString,"\\s+", " ");`
	`Console.WriteLine($"Cleaned String: {CleanedString}");`

	Заменить специальные символы
3 pattern	`Regex.Replace(your String, @"[^0-9a-zA-Z]+", "")`
	Этот код удаляет все специальные символы, но если вы не хотите удалять некоторые специальные символы, например. запятая","и двоеточие":" то: `Regex.Replace(Your String, @"[^0-9a-zA-Z:,]+", "")`
	Разделить строку па символу

4 pattern	`string azpattern = "[a-z]+";`
	`string str = "Asd2323b0900c1234Def5678Ghi9012Jklm";`
	`string[] result = Regex.Split(str, azpattern, RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(500));`
	`for (int i = 0; i < result.Length; i++){`
	`  Console.Write("'{0}'", result[i]);`

	`if (i < result.Length - 1)`
	`Console.Write(", ");`
	`}`

7. Допустимый номер мобильного телефона

Пример	Сопоставить номер **мобильного телефона шаблону**, например:
	`string[] str = { "+91-9678967101", "9678967101", "+91-9678-967101", "+91-96789-67101", "+919678967101"};`
1 pattern	`@"^\+?\d{0,2}\-?\d{4,5}\-?\d{5,6}"`

Примечания	`using System;`
	`using System.Collections.Generic;`
	`using System.Linq;`
	`using System.Text;`
	`using System.Text.RegularExpressions;`
	
	`namespace RegularExpression1 {`
	
	`class Program {`
	`static void Main(string[] args){`
	`Regex r = new Regex(@"^\+?\d{0,2}\-?\d{4,5}\-?\d{5,6}");`
	`string[] str = { "+91-9678967101", "9678967101", "+91-9678-967101", "+91-96789-67101", "+919678967101" };`
	`//Введите строки для совпадения с действительным номером`
	`// мобильного телефона.`
	`foreach (string s in str) {`
	`Console.WriteLine(" {0} {1} a valid mobile number.", s, r.IsMatch(s) ? "is" : "is not");`
	`//The IsMatch method is used to validate a string or`
	`//to ensure that a string conforms to a particular part`
	`} // foreach`
	`} // Main`
	`} // class`
	`} // namespace`

8. Проверка ввода символьных данных пользователя.

Пример	**Пользовательский ввод**
1 pattern	`If(!Regex.Match(firstNameTextBox.Text,"^[A-Z][a-zAZ]*$").Success) {...}`
2 pattern	`if (!Regex.Match(cityTextBox.Text, @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zAZ]+)$").Success) {...}`
3 pattern	`if (!Regex.Match(stateTextBox.Text, @"^([a-zA-Z]+|[a-zA-Z]+\s[azA-Z]+)$").Success) {...}`
4 pattern	`if (!Regex.Match(zipCodeTextBox.Text, @"^\d{5}$").Success) {...}`
5 pattern	`if (!Regex.Match(phoneTextBox.Text, @"^[1-9]\d{2}-[1-9]\d{2}-\d{4}$").Success) {...}`

9. Выбор слов из текста на основе патерна

Пример	**Отделить цифры от строк**
	`string Text = "1 One, 2 Two, 3 Three is good.";`
1 pattern	`@"\D+"`

Примечания	`string[] digits = Regex.Split(Text, @"\D+");`
	`foreach (string value in digits) {`
	`  int number;`
	`  if (int.TryParse(value, out number)) {`
	`  Console.WriteLine(value);`
	`  }`
	`}`

Регулярные выражения со следующими специальными символами не поддерживаются, так как они могут привести к задержкам в обработке электронной почты пользователя: * (звездочка) и + (знак плюс).

Пример 1. $L= (a)^* = \{a, aa, aaa, aaaa, \ldots\}$

$G(T=\{a\}, V=\{A,B\}, P = \{p_1, p_2,p_3\}, A)$

p_1. $A \rightarrow aB$

p_2 $B \rightarrow \varepsilon$

p_3. $B \rightarrow aC$

$A =>^1 aB =>^2 a$

$A =>^1 aB =>^3 aB =>^2 aa$

$A =>^1 aB =>^3 aB =>^3 aaaB$

$L = L(G) = \{ a, aa, aaa, aaaa, \ldots \}$

Пример 2. L(pattern = `D+`) $= \{0,1,2,3,4,5,6,7,8,9\}^+$

$G(T=\{0,1,2,3,4,5,6,7,8,9\}, V=\{A,B\}, P = \{p_1, p_2,p_3\}, A)$

p_1. $A \rightarrow 0B \mid 1B \mid 2B \mid 3B \mid 4B \mid 5B \mid 6B \mid 7B \mid 8B \mid 9B$

p_2 $B \rightarrow \varepsilon$

p_3 $B \rightarrow 0B \mid 1B \mid 2B \mid 3B \mid 4B \mid 5B \mid 6B \mid 7B \mid 8B \mid 9B$

$A =>^1 0$

$A =>^1 2B =>^3 25B =>^3 251B =>^3 2514B =>^2 2514$

L(pattern = `D+`) $= \{0,1,2,3,4,5,6,7,8,9\}^+ = L(G)$

[1] .NET Regular Expressions | Microsoft Docs

[2] HOWTO по регулярным выражениям | Python 3 (digitology.tech)

[3] Perl Compatible Regular Expressions - Wikipedia

2.7. Проектирование порождающих и распознающих систем

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

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

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

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

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

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

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

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

1.d. Используя грамматику G построить КА.

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

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

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

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

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

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

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

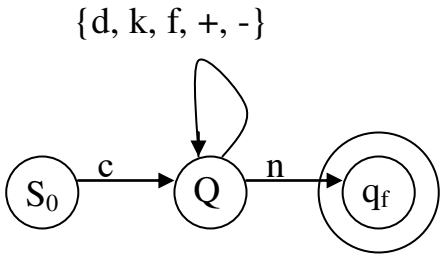


Рис. 2.6. Диаграмма переходов КА

Глава 3. Контекстно-свободные грамматики и МП-автоматы

3.1. Определение КС- грамматик и МП-автоматов

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

Определение 6. Контекстно-свободная грамматика типа 2:

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

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

Пример, КС-грамматики, порождающей скобочные арифметические выражения: $G_1 = (\{v,+,*,(,)\}, \{S_0,F,L\}, P, S_0)$, где $P = \{p_1: S_0 \to S_0 + F, p_2: S_0 \to F, p_3: F \to F * L, p_4: F \to L, p_5: L \to v, p_6: L \to (S_0)\}$.

Пример вывода, заменяя самый левый нетерминал (левосторонний вывод) сентенциальной формы : $S_0 \Rightarrow^1 S_0 + F \Rightarrow^2 F + F \Rightarrow^4 L + F \Rightarrow^5 v + F \Rightarrow^3 v + F * L \Rightarrow^4 v + L * L \Rightarrow^5 v + v * L \Rightarrow^5 v + v * v$.

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:

1). устранения бесполезных символов: *непроизводящих* и *недостижимых*;

2). устранения левой рекурсии;

3). преобразования в грамматику без ε–правил;

4). устранения цепных правил. то получим *приведенную* КС-грамматику.

Определение. Символ $x \in (T \cup V)$ называется *недостижимым* в КС-грамматике G, если он не может появиться ни в одной из ее сентенциальных форм. Т.е. символ недостижимый, если он не участвует ни в одной цепочке вывода из начального символа грамматики. Очевидно, что такой символ грамматике не нужен.

Определение. 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.

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

Рассмотрим алгоритмы и их применение.

Рис.3.1 Метод преобразования грамматики к *приведенной* форме

Алгоритм устранения бесполезных символов

Устранение *бесполезных* символов. Вначале исключить *непроизводящие* нетерминалы, а затем *недостижимые* символы. Выполнить:

1. Алгоритм устранения непроизводящих символов и

2. Алгоритм устранения недостижимых символов.

1. Алгоритм устранения непроизводящих символов

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

Вход: КС G = (T, V, P, S)

Выход: $V_p = \{A \mid A \Rightarrow^{+} \alpha, A \in V, \alpha \in T^{+}\}$, S'

1. Составить множество V_p нетерминалов, для которых найдется хотя бы одно правило, правая часть которого не содержит нетерминалов.
2. Если найдено такое правило, что все нетерминалы, стоящие в его правой части уже занесены в V_p, то добавить в V_p нетерминал, стоящий в его левой части.
3. Если на шаге 2, V_p больше не пополняется, то V_p содержит все производящие нетерминалы грамматики, а все нетерминалы не попавшие в него являются непроизводящими.

```
i = 0
V^i_p = Ø
do
   i = i + 1;
   foreach (A→ α ∈ P, α ∈(T ∪ V^(i-1)_p)^+ )
      if (A ∉ V^(i-1)_p )
         V^i_p = V^(i-1)_p ∪ {A}      // V^i_p = {A | (A→ α) ∈P, α ∈(T ∪ V^(i-1)_p)^+}
      end
   end foreach
while (V^i_p ≠ V^(i-1)_p)

if (S ∉ V^i_p)  S - непроизводящий символ, то определить S ∈ V^i_p
 n = 0
 foreach (A ∈ V^i_p)
    if | α | ≥ n, (A→ α ∈ P ))        при n = 0 если есть A→ ε
       n = | α |
       S = A
    end
 end foreach
end
```

1.Добавлять в P' только правила в правой части которых только производящие символы

Вход: КС $G = (T, V, P, S)$, V^i_p

Выход: $G_1 = (T, V^i_p, P', S')$

P' := Ø
foreach ($A \in V^i_p$, $(A \to \alpha) \in P$)
из цепочки символов α построить множество нетерминалов V_α
$V_\alpha = \{x \mid x \in \alpha, x \in V\}$ для $V_\alpha = Ø$, $V_\alpha - V^i_p = Ø$
if ($V_\alpha - V^i_p = Ø$) в V_α только производящие символы из V^i_p (- разность множеств)
$P' = P' \cup (A \to \alpha) \in P$
end
end foreach

2. Алгоритм устранения недостижимых символов

VT_r - *множество достижимых символов* (нетерминальных и терминальных).

Утверждение. Если нетерминал A в левой части правила грамматик G является достижимым, то достижимы и все символы α правой части этого правила $A \to \alpha$. $VT_r = \{x \mid A \to \alpha, x \in \alpha, \alpha \subseteq (V \cup T)^*\}$

Вход: КС грамматика $G_1 = (T, V^i_p, P', S')$

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

Шаг1. Построить множество VT^i_r - достижимых терминалов и не терминалов.

i = 0;
$VT^i_r = \{S'\}$
do
i = i + 1;
foreach $((A \to \alpha) \in P', A \in VT^{i-1}_r, \alpha \subseteq (V \cup T)^*)$
foreach $x \in \alpha$
if $x \notin VT^{i-1}_r$
$VT^i_r = VT^{i-1}_r \cup \{x\}$
end foreach
end foreach
while($VT^i_r \neq VT^{i-1}_r$)

Шаг 2. Построить P'', T', V'.

P'' = Ø
foreach $(A \to \{X_1,X_2,...,X_n\}, A, X_1,X_2,...,X_n \in VT^i_r) \in P$
$P'' := P'' \cup (A \to \{X_1,X_2,...,X_n\})$
end foreach

$T' := T \cap VT^i_r$
$V' := V \cap VT^i_r$

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

Алгоритм устранения непроизводящих символов.

множество $V^1_p = \{B, C\}$, $V^2_p = \{S, B, C\}$

$V - V_p = \{A\}$ – *непроизводящий символ*

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

Алгоритм устранения недостижимых символов.

$VT^1_r = \{S\}$

$VT^2_r = \{S, C, a\}$

$VT^3_r = \{S, C, a\}$, B – *недостижим*

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

Алгоритм устранения эпсилон-правил

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

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

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

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

Построение множества V_ε укорачивающих (обнуляемых) нетерминалов.

1. Алгоритм построение множества укорачивающих нетерминалов

Утверждение. Нетерминал A - укорачивающий (обнуляемый) если $A =>^* \varepsilon$.

1.A - укорачивающий для правила $A \to \varepsilon$ (см. Шаг 1.).

2.A - укорачивающий для правила $A \to B_1B_2 \ldots B_k$, если каждый нетерминал B_1 в правиле укорачивающий (см. Шаг 2.).

Вход: КС грамматика G = (T, V, P, S).

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

Шаг 1. Построить множество V^i_ε - укорачивающих нетерминалов для правил вида $A \to \varepsilon$.

```
i = 0
V^i_ε = Ø
foreach (A ∈ V)
   if (A → ε) ∈P // A → ε правило укорачивающиеся
      V^i_ε := V^i_ε ∪ {A}
   end
end foreach
```

Шаг 2. Построить множество V^i_ε - укорачивающих нетерминалов для правил вида $A \to \varepsilon$. $A \to B_1B_2 \ldots B_k$, если каждый нетерминал B_1 в правиле укорачивающий. $V^i{}_\varepsilon = V^{i-1}{}_\varepsilon \cup \{A \mid (A \to \alpha) \in P \text{ и } \alpha \in (V^{i-1}{}_\varepsilon)^+\}$. $\alpha \in (V^{i-1}{}_\varepsilon)^+$ - рассматриваются только правила, содержащие в правых частях только нетерминальные укорачивающие символы: $A \to \alpha$, например $A \to CDF$.

do
  i = i + 1
  $V^{i-1}{}_{\varepsilon} = V^{i}{}_{\varepsilon}$
  foreach $((A \rightarrow \alpha) \in P$ и $\alpha \in (V^{i-1}{}_{\varepsilon})^{+})$
    $V^{i}{}_{\varepsilon} := V^{i-1}{}_{\varepsilon} \cup A$
  end
while$(V^{i-1}{}_{\varepsilon} \neq V^{i}{}_{\varepsilon})$

2. Алгоритм устранения эпсилон-правил

Алгоритм устранения ε–правил в КС-грамматике основан на использовании множества укорачивающих нетерминалов. Алгоритм преобразует КС-грамматику с ε–правилами в эквивалентную КС-грамматику. Пусть $X_i \in \{X \mid (T \cup V) \cup \{\varepsilon\}, 0 < i \leq k\}$. Построить из $\{X_1, X_2, ..., X_k\}$ множество цепочек в которых, либо присутствует, либо устранён каждый из нетерминалов V_ε.

Вход: $G' = (T, V^{i}{}_{\varepsilon}, P, S)$
Выход: $G'' = (T, V', P', S')$

P' = Ø
foreach $(A \rightarrow X_1, X_2, ..., X_k \in P, X_1 \neq \varepsilon)$
  $V'_{\varepsilon} = \{ [X_1, X_2, ..., X_k] \cap V_{\varepsilon} \mid Xi < Xm \}$ V'_{ε} - упорядоченное мн-во
  if $(V'_{\varepsilon} = Ø)$
    $P' = P' \cup A \rightarrow X_1, X_2, ..., X_k$
  else построить $A \rightarrow \{Y_1, Y_2, ..., Y_k\}$
    Y = Ø
    foreach $a \in \{\alpha = \{ Xi, ..., Xm \} \subset V'^{+}_{\varepsilon} \mid |\alpha| \leq |V'_{\varepsilon}|$ и $Xi < Xm \}$
      foreach $v \in a$
        foreach $(X_i \in \{X_1, X_2, ..., X_k\}, A \neq X_i)$ правило $A \rightarrow A$ бессмысленно
          if $X_i = v$
            $Y_i = \varepsilon$
          end
          $Y = Y \cup Y_i$
        end foreach
        $P' = P' \cup (A \rightarrow \{Y_1, Y_2, ..., Y_k\}$
      end foreach
    end foreach
  end
end foreach

if $(S \in V_{\varepsilon})$
  $V' = V' \cup \{S'\}$
  $P' = P' \cup (S' \rightarrow S \mid \varepsilon)$
end
Положить G'' = (T, V', P', S').

Пример 1. Устранить из грамматика G ε–правила. Преобразовать грамматику G = (T, V, P, S), где V = {A,S}, T = {b,c}, P = {S→cA, S→ ε, A→cA, A→ bA, A→ ε}, в эквивалентную НКС-грамматику.

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

Шаг 2. Положить P' = Ø.

Рассмотрим правило S→cA.

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

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

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

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

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

Пример 2. Устранить из грамматика G ε–правила. Преобразовать грамматику

G = ({a,b,c}, {A,B,C,S},S,P)

P: S → AaB | aB | cC

A → AB | a | b | B

B → Ba | ε

C → AB | c

Алгоритм построения множества неукорачивающих нетерминалов V^i_ε:

1. $V^0_\varepsilon = \{B\}$, i = 1 (шаг 1)
2. $V^1_\varepsilon = \{B, A\}$, $V^1_\varepsilon \neq V^0_\varepsilon$, i = 2 (шаги 2 и 3)
3. $V^2_\varepsilon = \{B, A, C\}$, $V^2_\varepsilon \neq V^1_\varepsilon$, i = 3 (шаги 2 и 3)
4. $V^3_\varepsilon = \{B, A, C\}$, $V^3_\varepsilon = V^2_\varepsilon$, i = 3 (шаги 2 и 3)

Алгоритм устранения эпсилон-правил:

V = {A, B, C, S}, T = {a,b,c}, , строим множество P'

Добавить в P' все правила P за исключением ε - правил

P' p_1: S → AaB | aB | cC

p_2: A → AB | a | b | B

p_3: B → Ba

p_4: C → AB | c

Рассмотрим все правила множества из P' (шаг. 5):

p_1: S → AaB | aB | cC исключаем все комбинации $V^3_\varepsilon = \{B, A, C\}$ получим

S → Aa | aB | a | a | c добавим в P' без дублирования

P' = P' ∪ (S → Aa | aB | a | a | c)

p_2: A → AB | a | b | B исключаем все комбинации $V^3_\varepsilon = \{B, A, C\}$ получим

A → A | B их добавлять не надо A → A безсмысленно а A → B уже есть

p_3: B → Ba исключаем все комбинации $V^3_\varepsilon = \{B, A, C\}$ получим

B → a добавим в P' без дублирования

P' = P' $\cup$ (B $\rightarrow$ a) получим B $\rightarrow$ Ba | a

p_4: C $\rightarrow$ AB | c исключаем все комбинации V^3_ε = {B, A, C} получим
C $\rightarrow$ A | B | c добавим в P' без дублирования
P' = P' $\cup$ (C $\rightarrow$ A | B | c) получим C $\rightarrow$ AB | A | B | c

S $\notin V^3_\varepsilon$, поэтому в грамматику G не надо добавлять S'. В итоге получиим грамматику.

G' = ({a,b,c}, {A,B,C,S},S,P')
P': p_1: S $\rightarrow$ AaB | aB | cC | Aa | a | c
p_2: A $\rightarrow$ AB | a | b | B
p_3: B $\rightarrow$ Ba | a
p_4: C $\rightarrow$ AB | A | B | c

Алгоритм устранения цепных правил

Правило вида A$\rightarrow$B, где A и B - нетерминалы называется цепным.

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

Шаг 1. Для каждого правила X$\rightarrow$A$\in$P построить множество всех цепных символов V^i_X.

```
foreach (X∈V)   построить множество V_X = {A | X ⇒+ A}
   i = 0
   V^i_X = {X}
   do
      i++
      foreach (X ∈ V^(i-1)_X, X→ A∈P, A∈V)
         V^i_X = V^(i-1)_X ∪ {A}
      end
   while (V^i_X ≠V^(i-1)_X)
end
```

Шаг 2. Для каждого цепного правила A$\rightarrow$ R $\in$ P, Построить правила A$\rightarrow$ Yρ без цепных символов Y $\notin V^i_A$. Добавить в P' не цепные правила.

```
P' = Ø
foreach (A→ αRβ ∈ P)
   if (α = ε, β = ε, R ∈ V^i_A)   цепное правило
     foreach (R ∈ V^i_A)   найти правую не цепную часть правила
        if (R → Yρ ∈ P, Y ∉ V^i_A)
          P' = P' ∪ { A→ Yρ }
        end
     end
   else
      P' = P' ∪ { A→ αRβ}
   end
end
```

Пример. Устранить из КС грамматики G цепные правила, 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. Получаем $V_S = \{S, F, L\}$, $V_F = \{F, L\}$, $V_L = \{L\}$.

Шаг 2. Для каждого цепного правила A→ R ∈ P, **П**остроить правила A→ Yρ без цепных символов $Y \notin V^i_A$. Добавить в P' не цепные правила.

Для $V_S = \{S, F, L\}$ построим {S→ S + F | F*L | (S) | v }.

Включаем эти правила в P'={S→ S + F | F*L | (S) | v }.

Тоже для V_F, V_L.

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

Алгоритм устранения левой рекурсии

Определение. Нетерминал КС-грамматики называется *рекурсивным*, если A $\Rightarrow^+$ αAβ, 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**.

Непосредственной левой рекурсией считается такая рекурсия, при которой существует продукция вида A → Aa. Общим случаем устранения левой рекурсии обозначают устранение левой рекурсии, вызванной двумя и более шагами порождения.

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→ ε, и каждое правило из P отличное от S → ε, имеет вид A → aα, где a ∈ T, α ⊂ 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→ BC, где A, B, C ∈ V;
2. A→ a, где a ∈ T;
3. S→ ε, если ε ∈ L(G), причем S не встречается в правых частях правил.

Утверждение. Для любой КС-грамматики существует эквивалентная грамматика без левой рекурсии. Упорядочив нетерминалы по возрастанию индексов, и исключив правила вида $A_i \rightarrow A_j\alpha$, где j > i для всех A_i, получим что нет вывода $A_i \Rightarrow^* A_i$ и в грамматике нет левой рекурсии.

Алгоритм. Устранение непосредственной левой рекурсии **[1]** Ахо "Компиляторы: принципы,технологии и инструментарий":

Вход: КС грамматика G = (T, V, P, S_0), без ε-правил (вида A → ε).

Выход: Эквивалентная приведенная КС грамматика G' = (T, V, P', S'_0).

1. Упорядочить нетерминалы V по возрастанию V = [A_1, A_2 ,…, A_n]. Преобразоваить правила $A_i \rightarrow \alpha$ так, чтобы цепочка α начиналась либо с терминала, либо с такого A_j , что j > i. i = 1.

foreach i от 1 до n

foreach j от 1 до $i - 1$

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

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

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

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

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

4. Заменить каждое правило вида $A_i \rightarrow A_j\alpha$ правилами $A_i \rightarrow \beta_1\alpha |\ldots| \beta_m\alpha$, где $A_j \rightarrow \beta_1 |\ldots| \beta_m$ – 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.

Если ε присутствовал в языке исходной грамматики, добавим новый начальный символ S' и правила $S' \rightarrow S | \varepsilon$.

Корректность алгоритма на i итерации внешнего цикла:

1. В любой продукции вида $A_k \rightarrow A_l\alpha$, k < i, должно быть l > k. При следующей итерации внутреннего цикла растет нижний предел m всех продукций вида $A_i \rightarrow A_m\alpha$ до тех пор, пока не будет достигнуто $i \leqslant m$.

2. Будут только правила вида $A_i \rightarrow A_j\alpha$, где j > i. Неравенство становится строгим только после применения алгоритма устранения непосредственной левой рекурсии. При этом добавляются новые нетерминалы.

3. Созданный нетерминал A'_i имеет номер A_{-i} (номер меньший всех имеющихся на данный момент нетерминалов). Нет правила вида …→ A'_i, где A'_i самый левый нетерминал, а значит новые нетерминалы можно не рассматривать во внешнем цикле.

4. Все правила вида $A_i \rightarrow A_j\gamma$ где j < i заменяются на $A_i \rightarrow \alpha_1\gamma|\ldots|\alpha_k\gamma$ где $A_j \rightarrow \alpha_1|\ldots|\alpha_k$.

5. Итерация не меняет язык, а значит язык получившийся в итоге преобразования правил совпадает с исходным.

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

p_1: $S \rightarrow S + F | F$

p_2: $F \rightarrow F * L | L$

p_3:L → v | (S)

Шаг 1 . Пусть A_1 = S, A_2 = F, A_3 = L. V = [A_1, A_2, A_3].

Шаг 2. Для i = 1 преобразуем правила p_1: 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 = 1 и j = 2.

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

Шаг 2. Для i = 2 преобразуем правила p_2: 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 = 2, j = 2.

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

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

Обошли все нетерменалы

Получили правила новой грамматики G':

S→ F| FS', S'→ + F | + FS', F→L | LF', F'→ * L| LF', L → v |(S).

Алгоритм 2. Устранение косвенной левой рекурсии

Вход: приведенная КС-грамматика $G = (T, V, P, S)$, то есть грамматика не должна содержать цепных и ε − правил.

Выход: эквивалентная КС-грамматика без левой рекурсии.

$P' = \varnothing$ // множество конечных правил

$Pdel = \varnothing$ // хранение удаляемых правил

$V_1 = V$ // хранение дополняем нетерминалы со штрихами

foreach ($A_i \in V \mid 1 \leq i \leq n$)

//при начальном нетерминале следующие 3 вложенных цикла не выполняются

// но при i = 1 удаляется непосредственная левая рекурсия

foreach ($A_j \in V \mid 1 \leq j < i$)

foreach ($p \in P \mid A_i \rightarrow A_j\gamma$)

foreach ($p \in P \mid \{ A_j \rightarrow \alpha_1 \mid ... \mid \alpha_k \} \notin Pdel$)

//добавление правил вида $A_i \rightarrow x\gamma$

$P = P \cup A_i \rightarrow x\text{m}\gamma$

//добавление неиспользуемых правил

$Pdel = Pdel \cup A_i \rightarrow A_j\gamma$

end

end

end

$V_r = \varnothing$ //множество нетерминалов, имеющих левую рекурсию

//добавление в множество нетерминалов, имеющих левую рекурсию

foreach ($p \in P \mid A_i \rightarrow A_i\gamma$)

$V_r = V_r \cup A_i$

end

if ($A_i \in V_r$)

$V_1 = V_1 \cup A_i'$

end if

//добавление правил $A' \to \alpha_1 | \ldots | \alpha_k | \alpha_1 A' | \ldots | \alpha_k A'$ в конечное множество

foreach ($A_i \to A_i \alpha_1 | \ldots | A_1 \alpha_k \in P \mid A_i \to A_i \alpha_1 | \ldots | \alpha_k \notin Pdel$)

$P' = P' \cup A_i' \to \alpha_1 | \ldots | \alpha_k | \alpha_1 A_i' | \ldots | \alpha_k A_i'$

$Pdel = Pdel \cup A_i \to A_i \alpha_1 | \ldots | A_i \alpha_1$

end

//добавление правил $A \to \beta_1 | \ldots | \beta_m | \beta_1 A' | \ldots | \beta_m A'$ в конечное множество

foreach ($A_i \to \beta_1 | \ldots | \beta_m \in P \mid A_i \to \beta_1 | \ldots | \beta_m \notin Pdel$) {

$P' = P' \cup A_i \to \beta_1 | \ldots | \beta_m | \beta_1 A_i' | \ldots | \beta_m A_i'$

$P = P \cup A_i \to \beta_1 | \ldots | \beta_m | \beta_1 A_i' | \ldots | \beta_m A_i'$

end

//добавление правил, если не было рекурсии

if ($A_i \notin V_r$)

foreach ($R \to x_m \in \{ A_i \to \alpha_1 | \ldots | \alpha_k \} \mid R \to x_i \notin Pdel$)

$P' = P' \cup A_i \to x_m$

end

end if

end foreach

Пример 1.

$G = (\{a, b, d\}, \{S, C\}, P, S)$, где $P = \{S \to Sb, S \to Ca, S \to a, C \to d\}$

Расположить нетерминалы в некотором порядке:

$V = \{S, C\}$

1. $i = 1$

устранить непосредственную левую рекурсию среди Ai − продукции:

добавляем правила $A' \to \alpha_1 | \ldots | a_k | \alpha_1 A' | \ldots | a_k A'$:

$P' = \{S' \to bS', S' \to b\}$

добавляем правила $A \to \beta_1 | \ldots | \beta_m | \beta_1 A' | \ldots | \beta_m A'$ в P и P':

$P' = \{S \to Ca, S \to a, S \to CaS', S \to aS', S' \to bS', S' \to b\}$

$P = \{S \to Sb, S \to Ca, S \to a, C \to d, S \to Ca, S \to a, S \to CaS', S \to aS'\}$

2. i = 2, j = 1

Правила отсутствуют. Рекурсия отсутствует.

Добавляем правила для C в конечное множество:

$P' = \{S \to CaS', S \to aS', S \to bS', S \to b, S \to Ca, S \to a, C \to d\}$

Получим: $G' = (\{a, b, d\}, \{S, S', C\}, P', S)$, где

$P' = \{S \to CaS', S \to aS', S \to bS', S \to b, S \to Ca, S \to a, C \to d\}$

Пример 2 : $G = \{\{S, A\}, \{b, c, d\}, P, S\}$ $P = \{ S \to b \mid cA \mid c, A \to Ab \mid b \mid d \}$

Расположить нетерминалы в порядке : S, A

1. $i = 1$

V	S	A

i	1	
j		

$V_r = \varnothing$

рекурсия отсутствует

$P = \{ S \to b , S \to cA , S \to c, A \to Ab , A \to b , A \to d\}$

$P' = \{ S \to b , S \to cA , S \to c\}$

2. $i = 2, j = 1$

V	S	A
i		2
j	1	

Правила отсутствуют.

$V_r = \{A\}$

Устраняем рекурсию:

добавляем правила вида $A' \to \alpha_1|... |a_k|\alpha_1 A'|... |a_k A'$:

$P' = \{ S \to b , S \to cA , S \to c, A' \to bA', A' \to b\}$

$Pdel = \{A \to Ab\}$

добавляем правила вида $A \to \beta_1|... |\beta_m|\beta_1 A'|... |\beta_m A'$ в P и P':

$P' = \{ S \to b , S \to cA , S \to c, A' \to bA', A' \to b, A \to b, A \to d, A \to bA', A \to dA'\}$

$P = \{S \to b, S \to cA, S \to c, A \to Ab, A \to b, A \to d, A \to b, A \to d, A \to bA', A \to dA'\}$

Получим $G' = (\{a, b, d\}, \{S, S , C\}, P', S)$

где $P' = \{S \to b , S \to cA, S \to c, A \to b , A \to d, A \to bA' , A \to dA', A' \to bA', A' \to b\}$

Пример 3: $G = \{\{S, A\}, \{a, b, c, d\}, P, S\}$ $P = \{ S \to Aa \mid b, A \to Sd \mid Ac\}$

при $i = 1$, $V_r = \varnothing$ - рекурсии нет.

$P = \{ S \to Aa, S \to b, A \to Sd, A \to Ac\}$

$P' = \{ S \to Aa, S \to b\}$

при $i = 2, j = 1$

$P = \{ S \to Aa, S \to b, A \to Sd, A \to Ac, A \to Aad, A \to bd \}$

$Pdel = \{A \to Sd\}$

$V_r = \{A\}$

Устраняем рекурсию:

добавляем правила вида $A' \to \alpha_1|... |a_k|\alpha_1 A'|... |a_k A'$:

$P' = \{ S \to Aa, S \to b, A' \to cA', A' \to adA', A' \to c, A' \to ad \}$

$Pdel = \{A \to Sd, A \to Ac, A \to Aad\}$

добавляем правила вида $A \to \beta_1|... |\beta_m|\beta_1 A'|... |\beta_m A'$ в P и P':

$P' = \{ S \to Aa, S \to b, A' \to cA', A' \to adA', A' \to c, A' \to ad, A \to bd, A \to bdA'\}$

$P = \{ S \to Aa, S \to b, A \to Sd, A \to Ac, A \to Aad, A \to bd, A \to bdA'\}$

Пример 4: $G = \{\{S, A, B\}, \{a, b, c, d\}, P, S\}$

$P = \{ S \to Bc \mid Ad , A \to Sa \mid AbB \mid c , B \to Sc \mid b \}$

при $i = 1$

V	S	A	B

i	1		
j			

$Vr = \emptyset$, значит левой рекурсии нет, добавляем правила в конечное множество

$P' = \{S \to Bc,\ S \to cA\}$

$P = \{\ S \to Bc,\ S \to Ad\ ,\quad A \to Sa\ , A \to AbB, A \to c\ ,\ B \to Sc, AB \to b\ \}$

при $i = 2,\ j = 1$

V	S	A	B
i		2	
j	1		

такое правило есть: $A \to Sa$

добавляем в P правила $A \to Bca,\ A \to Ada$

$Pdel = \{A \to Sa\ \}$

$Vr = \{A\}$

Устраняем рекурсию:

добавляем правила вида $A' \to \alpha_1|... |a_k|\alpha_1 A'|... |a_k A'$:

$P' = \{S \to Bc,\ S \to cA,\ A' \to da, A' \to daA'\}$

$Pdel = \{\ A \to Ada, A \to AbB, A \to Sa\ \}$

добавляем правила вида $A \to \beta_1|... |\beta_m|\beta_1 A'|... |\beta_m A'$ в Pи P':

$P' = \{\ S \to Bc,\ S \to cA,\ A' \to da, A' \to daA',\ A \to BcaA',\ A \to Bca,\ A \to c,\ A \to cA'\}$

$P = \{S \to Bc, S \to Ad\ , A \to Sa\ , A \to AbB, A \to c\ ,\ B \to Sc, B \to b, A \to BcaA'\ ,\ A \to Bca, A \to c,\ A \to cA'\}$

$i = 3\ ,\ j = 1$

V	S	A	B
i			3
j	1		

такое правило есть: $B \to Sc$ добавляем в P правила: $B \to Bcc,\ B \to Adc$

$P = \{\ S \to Bc, S \to Ad,\ A \to Sa, A \to AbB,\ A \to c, B \to Sc, B \to b, A \to BcaA',$

$A \to Bca, A \to c, A \to cA', B \to Bcc\ , B \to Adc\}$

$Pdel = \{\ A \to Ada, A \to AbB, A \to Sa, B \to Sc\ \}$

$i = 3\ ,\ j = 2$

V	S	A	B
I			3
J		2	

Находим правила $B \to A\gamma$: $B \to Adc$

Добавляем новые правила:

$P = \{\ S \to Bc, S \to Ad\ , A \to Sa\ , A \to AbB, A \to c\ , B \to Sc, B \to b, A \to BcaA',$

$A \to Bca, A \to c, A \to cA', B \to Bcc\ , B \to Adc, B \to cdc, B \to cA'dc,$

$B \to BcaA'dc, B \to Bcadc\}$

$Pdel = \{ A \to Ada, A \to AbB, A \to Sa, B \to Sc, B \to Adc \}$
$Vr = \{B\}$
Устраняем рекурсию:
Добавляем правила вида $A' \to \alpha 1|... |ak|\alpha 1A'|... |akA'$:
$(B \to BcaA'dc, B \to Bcadc, B \to Bcc)$
$P' = \{S \to Bc, S \to cA, A' \to da, A' \to daA', A \to BcaA' , A \to Bca, A \to c, A \to cA', B' \to caA'dcB', B' \to caA'dc, B' \to cadcB', B' \to cadc, B' \to ccB', B \to cc \}$
$Pdel = \{ A \to Ada, A \to AbB, A \to Sa, B \to Sc, B \to Adc, B \to BcaA'dc, B \to Bcadc, B \to Bcc \}$
Добавляем правила вида $A \to \beta_1|... |\beta_m|\beta_1A'|... |\beta_mA'$ в P и P':
$(B \to b, B \to cdc, B \to cA'dc)$
$P' = \{S \to Bc, S \to cA, A' \to da, A' \to daA', A \to BcaA' , A \to Bca, A \to c, A \to cA', B' \to caA'dcB', B' \to caA'dc, B' \to cadcB', B' \to cadc, B' \to ccB', B \to cc, B \to bB', B \to b, B \to cdcB', B \to cdc, B \to cA'dcB', B \to cA'dc\}$

$P = \{ S \to Bc, S \to Ad , A \to Sa , A \to AbB, A \to c , B \to Sc, B \to b, A \to BcaA' , A \to Bca, A \to c, A \to cA', B \to Bcc , B \to Adc, B \to cdc, B \to cA'dc, B \to BcaA'dc, B \to Bcadc, B \to bB', B \to b, B \to cdcB', B \to cdc, B \to cA'dcB', B \to cA'dc\}$

Алгоритмы приведения грамматик к нормальной форме Хомского и Грейбах

Алгоритм 1. Приведение грамматики к нормальной форме Хомского

1) Удаление длинных правил

Для каждого длинного правила $A \to \alpha_1\alpha_2...\alpha_k$, $k > 2$, $\alpha_i \in V \cup T$:

1. Добавляем в грамматику k-2 новых нетерминала $A_1, \ldots, A_{k-2}$
2. Добавляем в грамматику k-1 новое правило:
   $A \to \alpha_1A_1$
   $A_1 \to \alpha_2A_2$
   $A_2 \to \alpha_3A_3$
   …
   $A_{k-2} \to \alpha_{k-1}\alpha_k$
3. Удаляем из грамматики правило $A \to \alpha_1\alpha_2...\alpha_k$

Вход: КС G = (T,V,P,S)
Выход: G'=(T, V', P', S)

V' = V
P' = ∅
foreach (A -> $\alpha \in$P)
  if ($A \to \alpha_1\alpha_2...\alpha_k$, $k > 2$)
    V' = V' ∪{$A_1, \ldots, A_{k-2}$} // создание k-2 нетерминалов

P’ = P’ ∪ A → $\alpha_1$$A_1$ // создание правила A → $\alpha_1$$A_1$
for (i = 2 .. k-2) // создание правил вида A_i → $\alpha_{i+1}$$A_{I+1}$
P’ = P’∪A_{i-1} → $\alpha_i$$A_i$
end for
P’ = P’∪A_{k-2} → $\alpha_{k-1}\alpha_k$ // создание правила A_{k-2} → $\alpha_{k-1}\alpha_k$
else
P’ = P’∪A → α
end if
end foreach

2) Удаление бесполезных символов
3) Удаление ε-правил
4) Если начальный нетерминал встречается в правых частях правил, нужно убрать правила такого вида. Заведем новый стартовый символ S’ и добавим правило S′ → S, а также S’ → ε вместо S → ε , если оно было.

Алгоритм:
V’ = V
P’ = P
foreach(A→ α ∈P)
if (S ∈ α)
V’ = V’∪{S’}
P’ = P’∪{S’→ S}
break
end if
end

5) Удаление цепных правил
6) Удаление ситуаций, в которых встречаются несколько терминалов

Алгоритм:
Для всех правил вида A → $\alpha_1\alpha_2$, α_i - терминал или нетерминал, заменяем все терминалы α_i на новые нетерминалы U_i и добавляем правила U_i → α_i.

Вход: КС G = (T,V,P,S)
Выход: G’=(T, V’, P’, S)

V’ = V
P’ = ∅
foreach (A → $\alpha_1\alpha_2$ ∈ P)

// если оба символа - терминалы
if ($A \to \alpha_1\alpha_2$, $\alpha_1, \alpha_2 \in T$)
/* выполняем проверку на существование правил $Y \to \alpha_2$ и $X \to \alpha_1$ в P и P', чтобы избежать дублирования правил */
// если правил вида $Y \to \alpha_2$ и $X \to \alpha_1$ нет
if ($X \to \alpha_1 \notin P'$ && $X \to \alpha_1 \notin P$ && $Y \to \alpha_2 \notin P'$ && $Y \to \alpha_2 \notin P$)
V' = V∪{A_1, A_2}
P' = P'∪{$A \to A_1A_2$, $A_1 \to \alpha_1$, $A_2 \to \alpha_2$}
// если нет только правила вида $X \to \alpha_1$
elif ($X \to \alpha_1 \notin P'$ && $X \to \alpha_1 \notin P$)
V' = V∪{A_1}
P' = P'∪{$A \to A_1Y$, $A_1 \to \alpha_1$}
// Y -> α_2, уже есть в P или P', поэтому в P' такое правило точно будет
// Аналогично дальше
elif ($Y \to \alpha_2 \notin P'$ && $Y \to \alpha_2 \notin P$)
V' = V∪{A_2}
P' = P'∪{$A \to XA_2$, $A_2 \to \alpha_2$}
else
P' = P'∪{$A \to XY$, $A_1 \to \alpha_1$}
end if
// если только первый символ - терминал
elif ($A \to \alpha_1\alpha_2$, $\alpha_1 \in T$)
if ($X \to \alpha_1 \notin P'$ && $X \to \alpha_1 \notin P$)
P' = P'∪{$A \to A_1\alpha_2$, $A_1 \to \alpha_1$}
V' = V'∪{A_1}
else
P' = P' ∪{$A \to X\alpha_2$}
end if
elif ($A \to \alpha_1\alpha_2$, $\alpha_2 \in T$)
V' = V∪{A_1, A_2}
P' = P' ∪{$A \to \alpha_1A_2$, $A_2 \to \alpha_2$}
if ($X \to \alpha_2 \notin P'$ && $X \to \alpha_2 \notin P$)
P' = P' ∪{$A \to \alpha_1A_2$, $A_2 \to \alpha_2$}
V' = V'∪{A_2}
else
P' = P' ∪{$A \to \alpha_1X$}
end if
else
P' = P'∪ {$A \to \alpha_1\alpha_2$,}
end if

end foreach

Пример 1.

Дано: P = { S → S + F, S -> F, F → F * L, F → L, L → (S), L →i, K -> i }

Шаг 1.

V’ = V = {S, F, L, K}

P’ = ∅

1. S → S+F, k=3 =>

Добавляем в V’ 1 новый нетерминал S_1.

Добавляем в P’ правила S → SS_1, S_1 → +F.

2. S → F, k=1 =>

Добавляем в P’ правило S → F.

Аналогично дальше:

3. F → F*L, k=3 => V’ = V’∪{F_1}, P’ = P’∪{F → FF_1, F_1 → *L}
4. F → L, k=1 => P’ = P’∪{F → L}
5. L → (S), k=3 => V’ = V’∪{L_1}, P’ = P’∪{L → (L_1, L_1 → S)}
6. L → i, k=1 => P’ = P’∪{L→ i}
7. K → i, k=2 => P’ = P’∪{K → i}

Результат:

V' = {S, S_1, F, F_1, L, L_1, K}

P' = {S → SS_1, S_1 → +F, S → F, F → FF_1, F_1 → *L, F → L, L → (L_1, L_1 → S), L→ i, K → i}

Шаг 2.

Удаление непроизводящих символов.

V^1_p = {K, L}, V^2_p = {F_1, F, L, K}, V^3_p = {S, S_1, F_1, F, L, K}, V^4_p = {S, S_1, F_1, F, L, L_1, K} => непроизводящих символов нет

Удаление недостижимых символов

VT^1_r = {S}

VT^2_r = {S, S_1, +, F}

VT^3_r = {S, S_1, +, F, F_1, *, L}

VT^4_r = {S, S_1, +, F, F_1, *, L, L_1, (}

VT^4_r = {S, S_1, +, F, F_1, *, L, L_1, (,), i}

=> K - недостижимый символ =>

V'' = {S, S_1, F, F_1, L, L_1}, P''= {S → SS_1, S_1 → +F, S → F, F → FF_1, F_1 → *L, F →L, L -> (L_1, L_1 -> S), L -> i}

Шаг 3.

P'' = {S → SS_1, S_1 → +F, S → F, F → FF_1, F_1 → *L, F → L, L→ (L_1, L_1 → S), L→ i}

ε-правил нет.

Шаг 4.
Вводим новый стартовый символ S’, добавляем новое правило S’ -> S
$V''' = \{S', S, S_1, F, F_1, L, L_1\}$, S' - начальный символ
$P''' = \{S' \to S, S \to SS_1, S_1 \to +F, S \to F, F \to FF_1, F_1 \to *L, F \to L, L \to (L_1, L_1 \to S), L \to i\}$

Шаг 5.
$V_{S'} = \{S', S, F, L\}$, $V_S = \{S, F, L\}$, $V_{S1} = \{S_1\}$, $V_F = \{F, L\}$, $V_{F1} = \{F_1\}$, $V_L = \{L\}$, $V_{L1} = \{L1\}$.
$V_{S'} = \{S', S, F, L\}$ построим $\{S' \to SS_1 \mid FF_1 \mid (L_1 \mid i\}$
$V_S = \{S, F, L\}$ построим $\{S \to SS_1 \mid FF_1 \mid (L_1 \mid i\}$.
$V_{S1} = \{S_1\} - \{S_1 \to +F\}$
$V_F = \{F, L\} - \{F \to FF_1 \mid (L_1 \mid i\}$
$V_{F1} = \{F_1\} - \{F_1 \to *L\}$
$V_L = \{L\} - \{L \to (L_1 \mid i\}$
$V_{L1} = \{L_1\} - \{L_1 \to S)\}$

$P'''' = \{S' \to SS_1 \mid FF_1 \mid (L_1 \mid i, S \to SS_1 \mid FF_1 \mid (L_1 \mid i, S_1 \to +F, F \to FF_1 \mid (L_1 \mid i, F_1 \to *L, L \to (L_1 \mid i, L_1 \to S)\}$

Шаг 6.

$P'''' = \{S' \to SS_1 \mid FF_1 \mid (L_1 \mid i, S \to SS_1 \mid FF_1 \mid (L_1 \mid i, S_1 \to +F, F \to FF_1 \mid (L_1 \mid i, F_1 \to *L, L \to (L_1 \mid i, L_1 \to S)\}$

1. $S' \to SS_1 \mid FF_1 \mid (L_1 \mid i \Rightarrow S' \to SS_1 \mid FF_1 \mid S'_2L_1 \mid i$ $S'_2 \to (S \to SS_1 \mid FF_1 \mid (L_1 \mid i \Rightarrow$

$S \to SS_1 \mid FF_1 \mid S'_2L_1 \mid iS'_2 \to ($

2. $S_1 \to +F \Rightarrow S_1 \to S_2F$ $S_2 \to +$
3. $F \to FF_1 \mid (L_1 \mid i \Rightarrow F \to FF_1 \mid S'_2L_1 \mid i$
4. $F_1 \to *L \Rightarrow F_1 \to F_2L$ $F_2 \to *$
5. $L \to (L_1 \mid i \Rightarrow L \to S'_2L_1 \mid i$ $L_1 \to S) \Rightarrow L_1 \to SL_2$ $L_2 \to)$

Результат:
$V = \{S', S, S_1, S'_2, S_2, F, F_1, F_2, L, L_1, L_2\}$
P:
$S' \to SS_1 \mid FF_1 \mid S'_2L_1 \mid i$
$S'_2 \to ($
$S \to SS_1 \mid FF_1 \mid S'_2L_1 \mid i$
$S_1 \to S_2F$, $S_2 \to +$
$F \to FF_1 \mid S'_2L_1 \mid i$
$F_1 \to F_2L$, $F_2 \to *$

$L \to S'_2L_1 \mid i$
$L_1 \to SL_2$
$L_2 \to)$

3.3. Свойства КС языков: лемма о накачке

Лемма о накачке КС языков. Пусть L - контекстно-свободный язык над алфавитом Σ, тогда существует такое p, что для любой цепочки $\omega \in L$, длины не меньше p найдется w = uxvyz такое что $|xy| \geqslant 1$ and $|xvy| \leqslant p$ и для любого i $\geqslant 0$, $j \geqslant 0$, $ux^ivy^jz \in L$.

Алгоритм. Определение накачки регулярных и КС языков

```
foreach( ω ∈ L = {a | S0 =>* a, a ∈ T*} )
  Λ = ∅
  foreach ( p ⩾ 1 and p ⩽ |ω|,  p++ )
      foreach( |xy| ⩾ 1 and |xvy| ⩽ p |  uxvyz = ω)
          foreach( i ⩾ 1 and i ⩽ |x|  and j ⩾ 1 and j ⩽ |y|, i++, j++)
              if ( x = a^i or y = b^j | a, b ∈ T)
                  Цепочка принадлежит регулярному языку
                  λ = {u x^i v y^j z}
                  Λ = Λ ∪ λ
              end if
              if ( x = a^i and y = b^j | a, b ∈ T )
                  Цепочка принадлежит КС-языку
                  λ = {u x^i v y^j z}
                  Λ = Λ ∪ λ
              end if
          end foreach
      end foreach
  end foreach
  if (Λ ≠ ∅)
      В Λ содержатся все повторения
  end if
end foreach
```

Рассмотрим КС язык в нормальной форме Хомского.

$S \to AB$ $A \to DE$ $B \to FD$ $E \to FG$
$G \to DB \mid g$ $F \to HG$ $D \to d$ $H \to h$

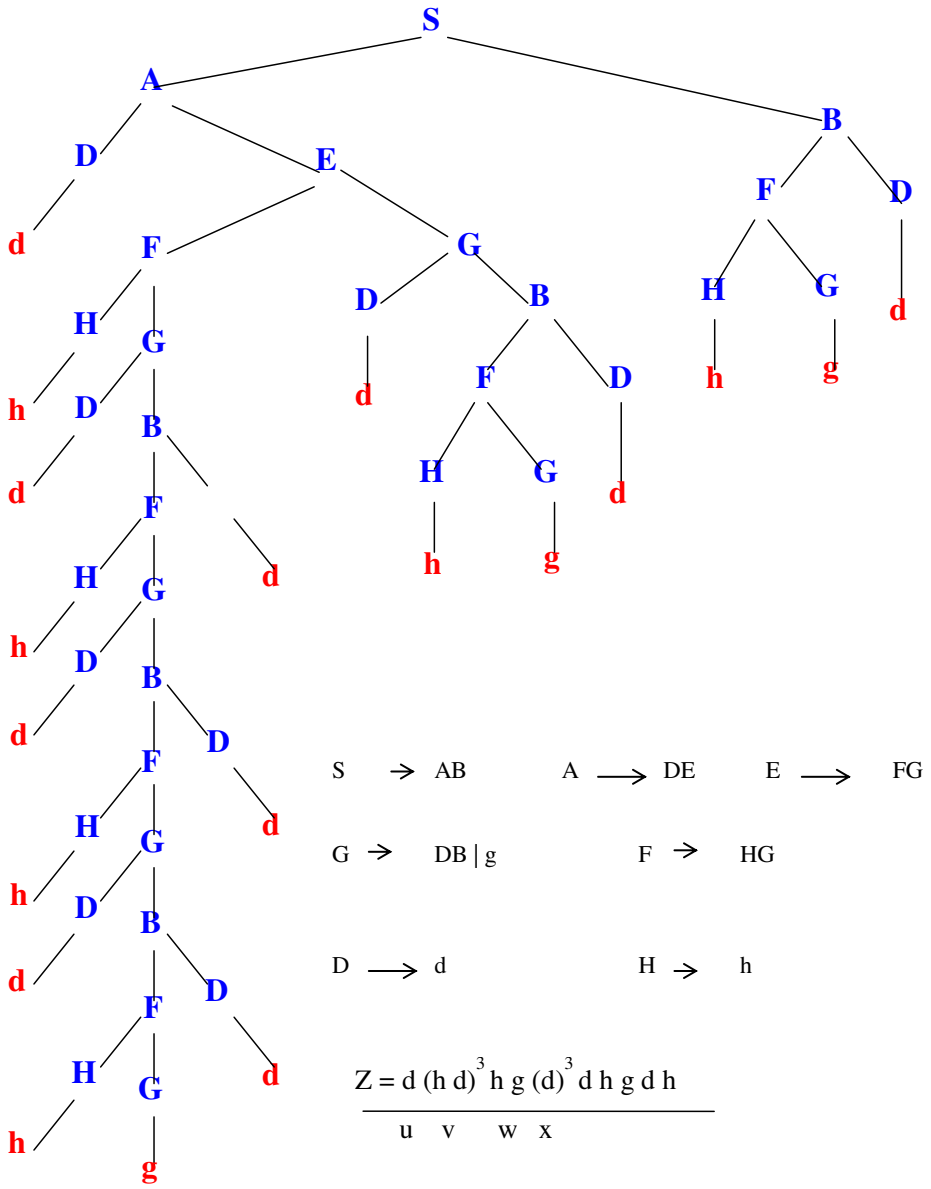


Рис. 2.6. Дерево разбора КС грамматики

Дерево вывода для строки z = d (h d)^3 h g (d)^3 d g h d g h d приведено на рис.2.6. Паттерн нетерминальной последовательности FGB-FGB-FGB встречается на самом длинном стволе дерева синтаксического анализа. Он имеет повторяющийся шаблон FGB, с листьями “hd” слева и “d” справа. Можно “вырастить” стебель, добавляя или удаляя повторяющиеся шаблоны F-G-B без ограничений.

Полученное дерево является деревом синтаксического анализа грамматики. Следовательно, грамматика имеет дерево разбора для z' = d (h d)^i h g (d)^i d g h d g h d, для всех i ≥ 0, что означает, что z' также принадлежит языку грамматики.

Пример. Z = UV^{i}WX^{i}Y с наибольшим возможным значением i.

!! Результат представить в виде таблицы

abcdefedcba
aaaaaaa ddddddd
a^7 d^7

aaabaaadaaa
aaabaaad a^3

aaab a^3 daaa
 a^3 baaadaaa
d^3

abbbbc
a b^4 c

abbbcccddde
a b^3 cccddde
abbb c^3 ddde
abbbccc d^3 e

abcdabcdabcdabcd
 abcd^4

aaabcdabcdabcdabcddaab
aa abcd^4 daab

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

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

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

3.4. Определение МП-автоматов

Автоматы с магазинной памятью (МП - автоматы) представляют собой модель распознавателей для языков, задаваемых КС-грамматиками. МП - автоматы имеют вспомогательную память, называемую магазином см. 1.10. В магазин можно поместить неограниченное количество символов. В каждый момент времени доступен только верхний символ магазина.

Верхний символом магазина будем считать самый левый символ цепочки.

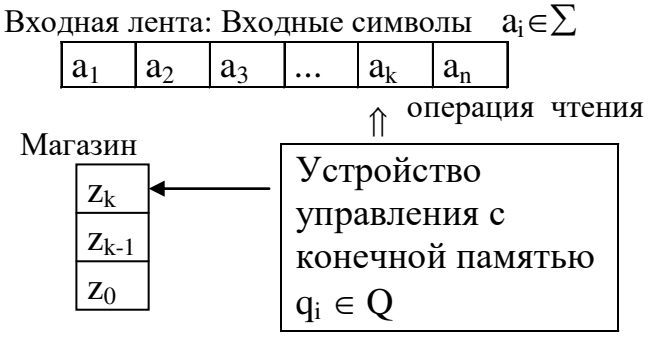


Рис. 1.10. Модель МП - автомата

Определение 9. МП автомат – это семерка объектов

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

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 $Q\times\Gamma^*$, δ: $Q\times(\Sigma \cup \{\varepsilon\})\times\Gamma \to Q\times\Gamma^*$

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 $z = \varepsilon$, то магазин считается пустым).

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

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

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

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

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

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

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

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

Есть два способа определить язык, допускаемый МП-автоматом:

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

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

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

Язык распознаваемый МП-автоматом, называется множество цепочек:

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

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

Цепочка символов языка L: aaabbb

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

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

2.$\delta(q ,a, a) = \{(q, aa)\}$

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

4.$\delta(q, \varepsilon, z_0) = \{(q_f, \varepsilon)\}$

Последовательность тактов:

$(q_0, aaabbb, z_0) \vdash^1 (q, aabbb, az_0) \vdash^2 (q, abbb, aaz_0) \vdash^2$
$(q, bbb, aaaz_0) \vdash^3 (q, bb, aaz_0) \vdash^3 (q, b, az_0) \vdash^3$
$(q, \varepsilon, z_0) \vdash^4 (q_f, \varepsilon)$

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

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

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

Пример проектирования МП-автомата. Постановка задачи. Построить МП-автомат и расширенный МП-автомат по КС-грамматики G = (T, V, P, S), без левой рекурсии, автомат распознает скобочные выражения. Написать последовательность тактов для выделенной цепочки. Определить свойства автомата.

1. А). Построить МП-автомат, распознающий скобочные арифметические выражения заданные КС-грамматикой G_1 = ({v,+,*,(,)}, {S_0,E,F,L}, P, S_0), где P состоит из правил: p_1: $S_0 \rightarrow$ (E)*L | F, p_2: E$\rightarrow$ E+F | w, p_3: F $\rightarrow$ F * L | L, p_4: L $\rightarrow$ v.

$S_0 =>^1$ (E)*L$=>^2$ (E+F)*L$=>^2$ (w+ F)*L$=>^3$ (w+ L)*L$=>^4$ (w+ v)*L$=>^4$ (w+ v)*v

Используя алгоритм 3.8. получим: МП = ({q}, {w,v,+,*,(,)}, {w,v,+,*,(,), S_0,E,F,L}, δ, S_0, z_0, {q}), в котором функция переходов δ определяется следующим образом:

1. $\delta(q_0, \varepsilon, S_0)$ = {(q,(E)*L), (q, F)};
2. $\delta(q, \varepsilon, E)$ = {(q, E+F), (q,w)};
3. $\delta(q, \varepsilon, F)$ = {(q, F*L), (q, L)};
4. $\delta(q, \varepsilon, L)$ = {(q, v)};
5. $\delta(q, a, a) = \{(q, \varepsilon)\}$ для всех $a \in \sum = \{w, v, +, *, (,)\}$.

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

$(q_0, (w+ v)*v, S_0) \vdash^1 (q, (w+ v)*v, (E)*L) \vdash^5 (q, w+ v)*v, E)*L) \vdash^2 (q, w+ v)*v, E+F)*L) \vdash^2 (q, (q, w+ v)*v, w+F)*L) \vdash^5 (q, + v)*v, +F)*L) \vdash^5 (q, v)*v, F)*L) \vdash^3 (q, v)*v, L)*L) \vdash^4 (q, v)*v ,v)*L) \vdash^5 (q,)*v,)*L) \vdash^5 (q, *v,*L) \vdash^5 (q, v, L) \vdash^4 (q, v, v)) \vdash^5 (q, \varepsilon)$

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, ∑, 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 ⊥ - конец цепочки. Пусть РМП = ({q,r}, ∑, $\sum \cup V \cup \perp$, δ, q, ⊥, {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 $A \to \alpha$ - 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_0,E,F,L,⊥}, δ, q, ⊥, {r}), где функция переходов δ определена следующим образом:

1. $\delta(q, a, \varepsilon) = \{(q, a\}$ для всех $a \in \sum = \{w, v, +, *, (,)\}$
2. $\delta(q, \varepsilon, (E)*L) = \{(q, S_0)\}$;
3. $\delta(q, \varepsilon, F) = \{(q, S_0)\}$;
4. $\delta(q, \varepsilon, E+F) = \{(q, E)\}$;
5. $\delta(q, \varepsilon, w) = \{(q,E)\}$;
6. $\delta(q, \varepsilon, F*L) = \{(q, F)\}$;
7. $\delta(q, \varepsilon, L) = \{(q, F)\}$;
8. $\delta(q, \varepsilon, v) = \{(q, L)\}$;
9. $\delta(q, \varepsilon, \perp S_0) = \{(r, \varepsilon)\}$;

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

(q, (w+ v)*v, ⊥) $\vdash^9$ (q, w+ v)*v, ⊥()
$\vdash^1$ (q, + v)*v, ⊥(w)
$\vdash^5$ (q, + v)*v, ⊥(E)
$\vdash^1$ (q, v)*v, ⊥(E+)
$\vdash^1$(q,)*v, ⊥ (E+v)
$\vdash^8$ (q,)*v, ⊥ (E+L)
$\vdash^7$ (q,)*v, ⊥ (E+F)
$\vdash^4$ (q,)*v, ⊥(E)
$\vdash^1$ (q, *v, ⊥ (E))
$\vdash^1$ (q, v, ⊥(E)*)
$\vdash^1$ (q, ε, ⊥(E)*v)
$\vdash^8$ (q, ε, ⊥(E)*L)
$\vdash^2$ (q, ε, ⊥S_0)
$\vdash^9$ (r, ε)

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

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

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

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

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

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

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

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

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

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

3.5. Способы реализации синтаксических анализаторов

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

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

2. Строятся диаграммы Вирта. Для каждого праивла грамматики строится диаграмма. Не терминальные символы реализуются вызовом подпрограмм. Таким образом создается неявный стек при их вызове и возможна косвенная рекурссия, что не требует преобразования грамматика в МП автомат.

3. Таблично-управляемый разбор (предварительно строится диаграмма Вирта), диаграмма сопоставляется с узлами и ребрами графа, что реализуется списочной структуры, над которой выполняется алгоритм обхода графа соответствующий распознаванию входной строки.

Диаграммы Вирта

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

2. Шаги реализации.

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

Привести вывод выделенной цепочки:

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

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

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

Составить синтаксический граф, диаграмму Вирта для детерминированного грамматического разбора с просмотром вперед на один символ. Никакие две ветви не должны начинаться с одного и того же символа, если какой-либо граф A можно пройти, не читая вообще никаких входных

символов, то такая “нулевая ветвь” должна помечаться всеми символами, которые могут следовать за A (это необходимо для принятия решения о переходе на эту ветвь).

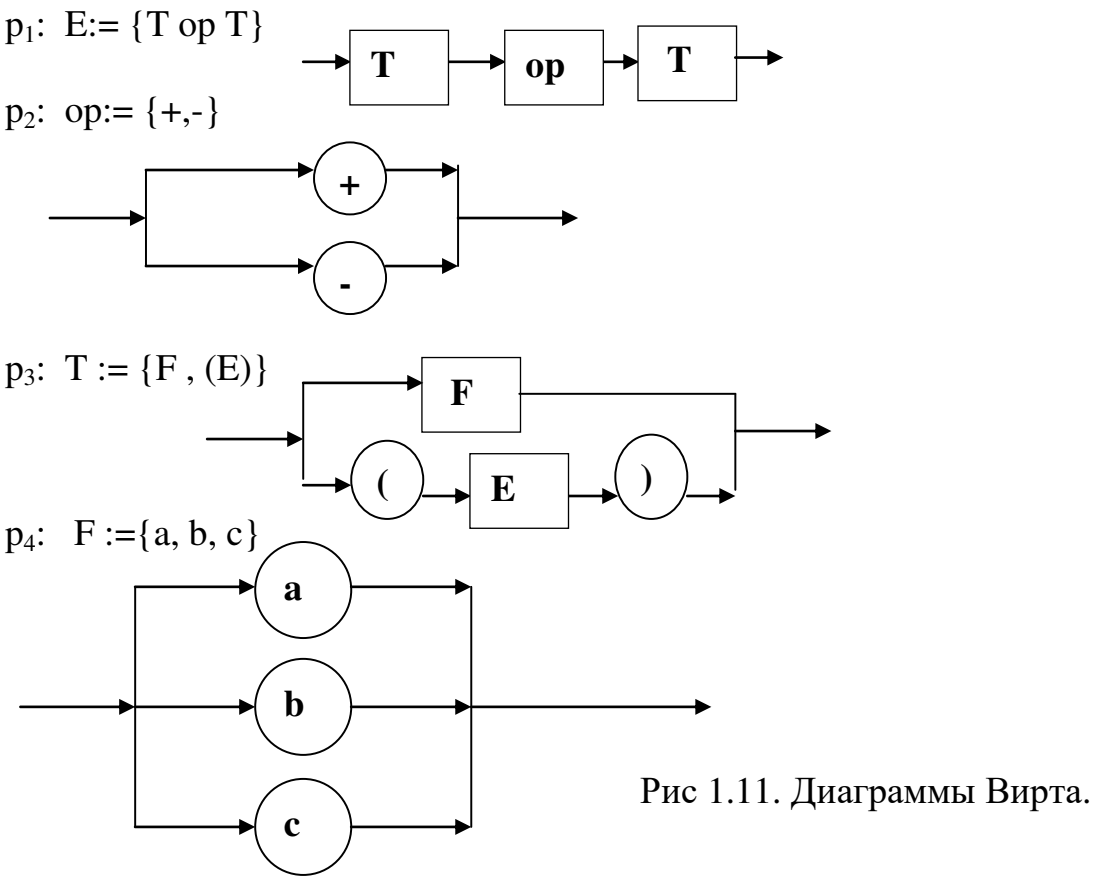


Рис 1.11. Диаграммы Вирта.

Правила преобразования графа в программу:

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

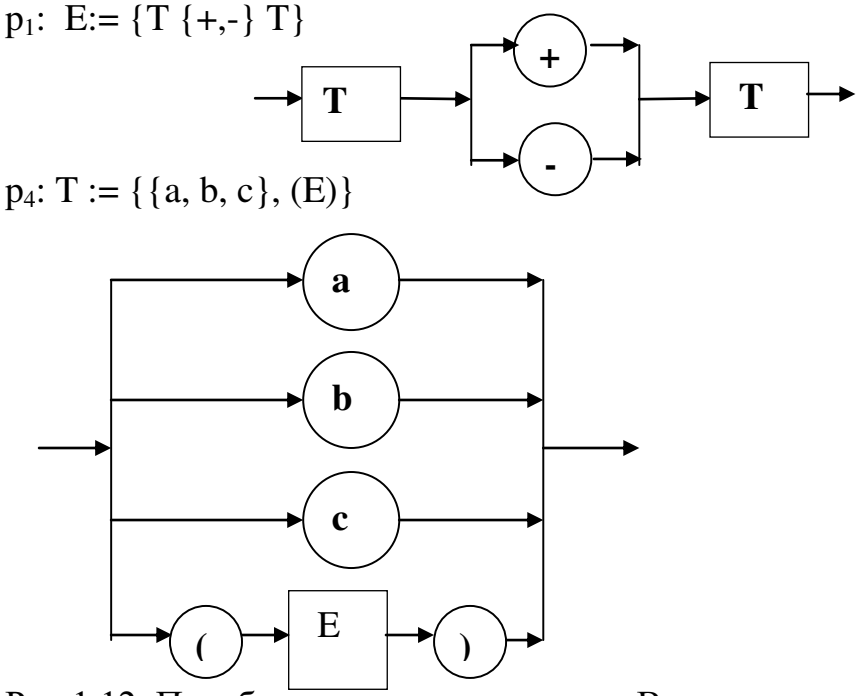


Рис 1.12. Преобразованные диаграммы Вирта.

Полученные правила на графе соответствуют нормальной форме Грейбах.

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

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

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

Таблично-управляемый разбор

1. Грамматика задается в виде данных, процедура разбора универсально для всех грамматик. Программа работает в строгом соответствии с методом простого нисходящего грамматического разбора и основывается на детерминированном синтаксическом графе (т.е. предложение должно анализироваться просмотром вперед на один символ без возврата).
2. Естественный способ представить граф в виде структуры данных, а не программ – это ввести узел для каждого символа и связать эти узлы с помощью ссылок. Выделим два типа узлов для терминальных (идентифицируется терминальным символом) и нетерминальных символов (ссылка на данные нетерминального символа). Тогда структура узла графически может быть представлена как:

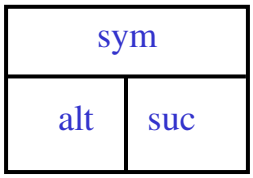


Рис. 1. 14. Структура узла Node

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

3. Правила преобразования графов, в структуру данных: получить как можно меньшее число графов с помощью подстановок.
4. Последовательность элементов преобразовать в список узлов (горизонтальный по “suc”), список альтернатив в список узлов (горизонтальный по “alt”), цикл в узел с указателем “suc” на себя, а “alt” на узел с “empty”, где “alt” = NULL, а “suc” выход из цикла.
5. Построим реализацию в виде структуры данных для рассмотренных в п.3 диаграмм.
6. Программа, реализующая структуру на рис. 1.15 приведена ниже. Булевская переменная b управляет алгоритмом разбора. Переменная state определяет состояние цепочки символов.

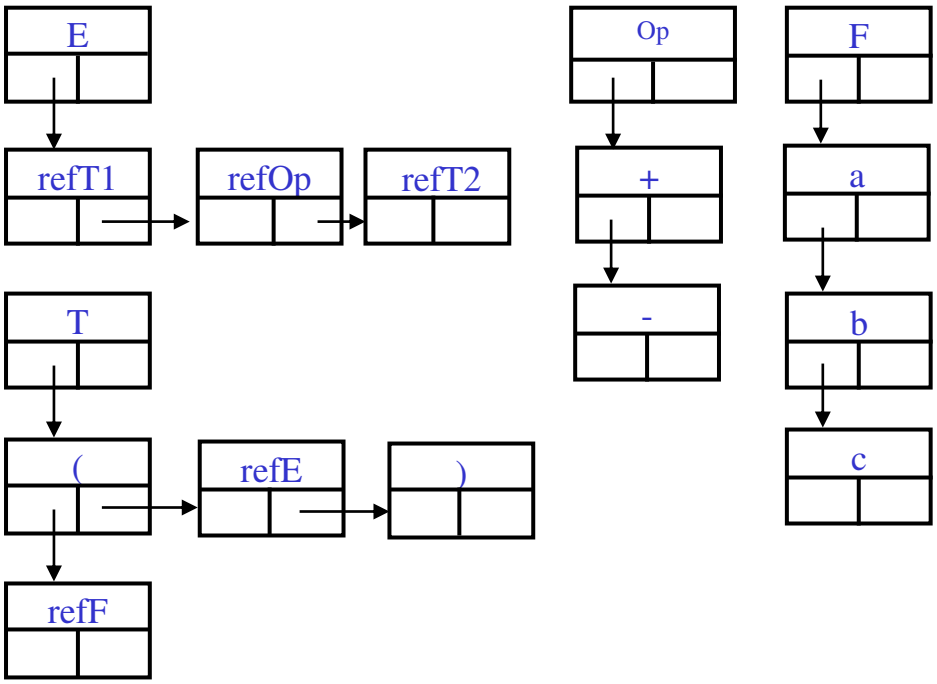


Рис. 1.15. Структура данных для диаграмм п.5

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

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 (простая от анг. simple)-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.

Если грамматика не удовлетворяет требованиям, то применяют факторизацию - представляет собой преобразование грамматики в пригодную для предиктивного анализа. Основная идея левой факторизации заключается в том, что когда не ясно, какая из двух альтернативных продукций должна использоваться для нетерминала A, A-продукции можно переписать так, чтобы отложить принятие решения до тех пор, пока из входного потока не будет прочитано достаточно символов для правильного выбора.

Алгоритм. Левая факторизация грамматики

Вход: грамматика G.

Выход: эквивалентная левофакторизованная грамматика.

Метод: для каждого нетерминала A находим самый длинный префикс α, общий для двух или большего числа альтернатив. Если $\alpha \neq \varepsilon$, т.е. имеется нетривиальный общий префикс, заменим все продукции $A \to \alpha\beta1 \mid \alpha\beta2 \mid \ldots \mid \alpha\beta n \mid \gamma$, где γ представляет все альтернативы, не начинающиеся с α, продукциями

$A \to \alpha A' \mid \gamma$

$A' \to \beta1 \mid \beta2 \mid \ldots \mid \beta n$

Здесь A′ — новый нетерминал. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс.

Рассмотрим частный случай функции FIRST:

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

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

Обобщим определение множества FIRST так, чтобы его можно было применить для правил произвольного вида. Множество FIRST(α) состоит из множества терминальных символов, которыми начинаются цепочки, выводимые из цепочки α.

$FIRST(\alpha) = \{a \in T \mid S \Rightarrow^{+} \alpha \Rightarrow^{+} a\beta, \text{ где } \alpha \in (T \cup V)^{+}, \ \beta \in (T \cup V)^{*}\}$

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

$FOLLOW(A) = \{a \in T \mid S \Rightarrow^{*} \alpha A \gamma \text{ и } a = FIRST(\gamma)\}$

Пример. G = (V = {S, F, L} , T = {i, =, *, (,)}, P, S)

P:

1. $S \to F{=}L$
2. $S \to L$
3. $F \to (*L)$
4. $F \to i$
5. $L \to F$

Грамматика не принадлежит классу LL(1), так как для правил $S \to F{=}L \mid L$:
FIRST(F=L) = FIRST(L) = FIRST(F) = {(, i}
Применив алгоритм факторизации правилам $S \to F{=}F \mid F$, заменяем их на правила $S \to FS'$, $S' \to {=}F \mid \varepsilon$. Получим грамматику без неопределённостей:
G = ({S, S', F}, {i, =, *, (,)}, P, S)
P:

1. $S \to FS'$
2. $S' \to {=}F$
3. $S' \to \varepsilon$
4. $F \to (*F)$
5. $F \to i$

1. Построение функции FIRST

FIRST(S) = FIRST(F) = {(, i}

FIRST(S') = {=, ε}

FIRST(F) = {(, i}

FIRST(FS') = FIRST(F) = {(, i}

FIRST((*F)) = {(}

1. Построение функции FOLLOW

FOLLOW(S) = {⊥}

FOLLOW(S') = {⊥}

FOLLOW(F) = FIRST(S') \ {ε} ∪ FOLLOW(S') ∪ {)} = {=, ⊥,)}

LL(k) анализатор. Пусть 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.

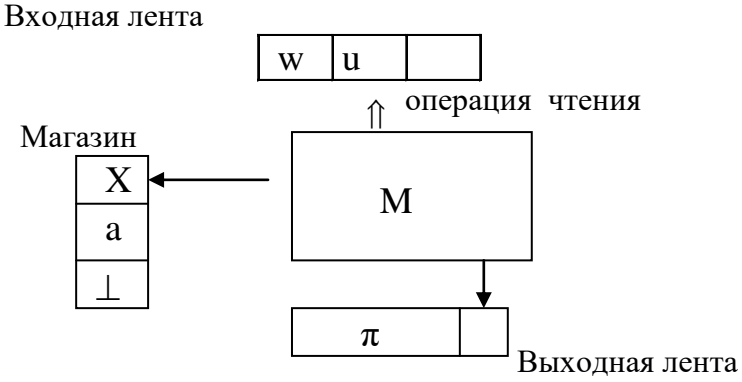


Рис. 1.18. LL(k)-анализатор

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,α)* = (β, i) следующая:

1. $(x, X\alpha, \pi) \vdash^{i} (x, \beta\alpha, \pi i)$, если *M(X,α)* = (β, 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 (ε, ⊥, π), что соответствует элементу управляющей таблицы *M(*⊥, ε*)* = ДОПУСК, то его работа прекращается, и выходная цепочка π является левым разбором входной цепочки.

4. Если алгоритм достигает конфигурации (x, Xα, π) и *M(X,*a*)* = ОШИБКА, то разбор прекращается и выдается сообщение об ошибке.

Конфигурация (ω, S⊥, ε), где $S \in V_p$ – начальный символ магазина (начальный символ грамматики), называется *начальной конфигурацией.*

Если (ω, S⊥, ε) $\vdash\!\!\!\rightarrow^{+}$ (ε, ⊥, π), 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. Для каждого терминала $a \neq \varepsilon$ из FIRST(α), $A \rightarrow \alpha$ – правило грамматики с номером i, добавляем в M(A, a) = (α, i).

Если $\varepsilon \in$ FIRST(α), то для каждого терминала b из FOLLOW(A) добавляем $A \rightarrow \alpha$ в M(A, b) = (α, i). Если $\varepsilon \in$ FIRST(α) и ε = FOLLOW(A), то добавляем $A \rightarrow \alpha$ в M(A, ε).

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

3. M(⊥, ε) = 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 $A \rightarrow \beta$ и $A \rightarrow \gamma$ 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.

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

А) Используем алгоритм 3.10. Управляющая таблица должна содержать 11 строк, помеченных символами из множества $(V \cup T \cup \{\perp\})$, и 6 столбцов, помеченных символами из множества $(T \cup \{\varepsilon\})$.

	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 по рядам. Последовательно рассмотрим все нетерминальные символы.

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

M(S, ( ) = M(S, i) = TE', 1

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

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

p_3: E'→ε имеет простую правую часть, вычислим множество символов, следующих за нетерминалом E' в сентенциальных формах. Построив левый вывод $S \Rightarrow TE' \Rightarrow PT'E' \Rightarrow (S)T'E' \Rightarrow (TE')TE' \ldots$, 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'	FIRST(PT') = {(,i}	M(T',() = M(T',i) = TE',4

p_5: T'→ *PT'	FIRST(*PT') = {*}	M(T',*) = *PT',5
p_6: T'→ ε	FOLLOW(T') = {+,), ε}	M(T',+)= M(T',)) = M(T',ε) = ε,6
p_7: P→ (S)	FIRST((S)) = {(}	M(P,() =(S),7
p_8: P→ i	FIRST(i) = {i}	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⊥

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

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

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

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

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

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

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

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

Выполняя шаги 1 и 2, алгоритм выполняет 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) ⊢ (*i, *PT'E'⊥, 14862485) ⊢ (i, PT'E'⊥, 14862485) ⊢ (i, iT'E'⊥, 148624858) ⊢	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) M(*, *) = ВЫБРОС M(P, i) = (i, 8) M(i,i) = ВЫБРОС

(ε, T'E'⊥, 148624858) ⊢ (ε, E'⊥, 1486248586) ⊢ (ε, ⊥, 14862485863)	M(T', ε) = (ε, 6) M(E', ε) = (ε, 3)

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

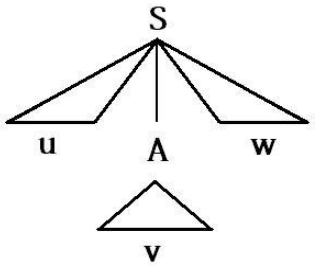
Так как значение M(⊥,ε) = ДОПУСК, то цепочка i+i*i принадлежит языку и последовательность номеров правил 14862485863 на выходной ленте – это ее разбор.

1.7. LR(k)-грамматики

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

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

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.

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

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

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

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

2. Пополненной грамматикой, полученной из КС-грамматики G = (T, V, P, S), называется грамматики G' = (V ϵ {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.

Утверждение. Если правило грамматики A→β не зависит от w, то во множестве $FIRST_k(w)$ содержится информация достаточная для определения основы LR(k)-грамматики, и применения операции Свертка.

Определение 15. КС-грамматика G = (T, V, P, S) называется LR(k)-грамматикой для $k \geq 0$, если существуют два правых вывода для *пополненной* грамматики G' = (T, V', P', S') полученной из G: (A→β (ab_ B_ c), B→σ, σx = βy

$$S' =>_r^* \alpha Aw =>_r \alpha\beta w,$$
$$S' =>_r^* \gamma Bx =>_r \gamma\sigma x =>_r \alpha\beta y,$$

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

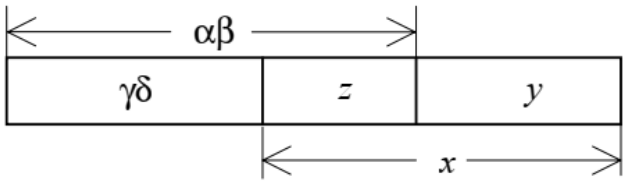


Рис. 3.1

Поскольку γσx = αβy, то | γσx | = | αβy |, и, учитывая, что | γσ | < | αβ |, заключаем: | x | > | y | , т.е. x = zy при некотором z V^+_T, | z | > 0. Заметим, что цепочка **z является префиксом** цепочки x, а не ее окончанием, так как именно цепочка y является окончанием всей сентенциональной формы αβy. Два разных разбиения одной и той же сентенциальной формы γσx = αβy представлены на рис. 3.1.

Условие γσx = αβy, можно переписать как γσzy = αβy, и потому γσz = αβ.

Если αβw и αβy – правовыводимые цепочки пополненной грамматики G', и $FIRST_k(w) = FIRST_k(y)$, 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.

Алгоритм "Перенос-Свертка"

При LR(k)-анализе применяется алгоритм перенос-свертка (англ. shift-reduce). Суть метода сводится к следующему:

1. Входная цепочка символов считывается до тех пор, пока не накопится в магазине цепочка, совпадающая с правой частью какого-нибудь из правил P. Операция Перенос переносит символ в магазин.

2. Далее все символы совпадающей цепочки извлекаются из стека и на их место помещается нетерминал, находящийся в левой части этого правила (операция Свертка).

Алгоритм “перенос-свертка” λ описывается в терминах конфигураций вида ($\perp X_1 \ldots X_m, a_1 \ldots a_n, p_1 \ldots p_r$), рис. 1.18. где :

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

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

$p_1 \ldots p_r$ – разбор правил, полученный к данному моменту времени на ленте π.

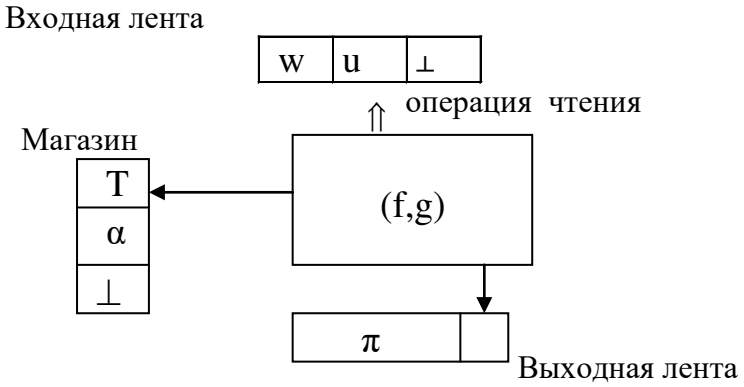


Рис. 1.18. LR(k)-анализатор

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

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

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

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

3. Если f(α, w) = ДОПУСК (Д), то (α, w, π) ⊢ ДОПУСК.

В остальных случаях (α, w, π) ⊢ ОШИБКА (пустое значение в таблице).

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

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.

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

Пример. 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

$S =>^1 bAb =>^2 bcBb =>^4 bcAadb =>^3 bcaadb$

Разберем с помощью алгоритма λ входную цепочку bcaadb.

Начальная конфигурация алгоритма (⊥, bcaadb, ε).

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:

$(\perp, bcaadb, \varepsilon) \vdash^s (\perp b,caadb, \varepsilon)$ и следующие шаги:

$(\perp b,caadb, \varepsilon) \vdash^s (\perp bc, aadb, \varepsilon) \vdash^s (\perp bca, adb, \varepsilon)$

Выполнение алгоритма λ

⊥	bcaadb	$\vdash^S$
⊥b	caadb	$\vdash^S$

⊥bc	aadb	$\vdash^{S}$
⊥bca	adb	$\vdash^{r}$ по p3
⊥ bcA	adb	$\vdash^{S}$
⊥ bcAa	db	$\vdash^{S}$
⊥ bcAad	**b**	$\vdash^{r}$ по p4
⊥ bcB	**b**	$\vdash^{r}$ по p2
⊥ bA	**b**	$\vdash^{S}$
⊥ bA**b**	**ε**	$\vdash^{r}$ по p1
⊥ S	**ε**	ДОПУСК

Очередной шаг выполнения алгоритма определяется значением f(⊥bca, adb, ε), котрое имеет значение СВЕРТКА. Правило, используемое при свертке, определяется значением функции f(⊥bca, adb) = 3 и переходим в следующую конфигурацию: (⊥bca, adb, ε) $\vdash^{s}$ (⊥bcA, adb, 3), следующие 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:

$S \Rightarrow^{1} bAb \Rightarrow^{2} bcBb \Rightarrow^{4} bcAadb \Rightarrow^{3} bcaadb$

Определение. ***Основой называется кодируемая цепочка символов в верхней части магазина.***

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

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

Пример. В приведённой грамматике с правилами:

p_1: S → F & L,

p_2: S → (S),

p_3: F → * L,

p_4: F → i,

p_5: L →F

грамматические вхождения $\&_{1,2}$, $)_{2,3}$, $L_{3,2}$ и т.д.

Каждый символ магазинного алфавита V_p (грамматическое вхождение символа грамматики в правую часть правила вывода) можно интерпретировать как кодированное представление подцепочки из правой части правила, последним символом которой он является. При такой интерпретации магазинный символ вталкивается только тогда, когда символ из верхушки магазина является *кодированным представлением префикса подцепочки, представляемой вталкиваемым в магазин символом*, или когда вталкиваемый символ представляет *новый префикс правой части какого-либо правила*. Перенос символов в магазин осуществляется до тех пор, пока в верхушке магазина не окажется символ, являющийся кодированным представлением

основы. В этом случае цепочка магазинных символов, соответствующих основе, выталкивается из магазина, и в магазин вталкивается символ, который исполняет роль подцепочки, совместимой с подцепочкой, представляемой магазинным символом, расположенным под основой.

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

Два способа построения анализаторов для LR(k)-грамматик без ε-правил на основе активных префиксов

Два способа построения анализаторов для LR(k)-грамматики без ε-правил на основе активных префиксов:

I. На основе кодирования активных префиксов и отношения OBLOW (be low входит под) для LR(0) и SLR(1)-грамматика без ε-правил. SLR(1)-грамматики (где S - simple) не принадлежат классу LR(0) грамматик.

Алгоритм построения LR(0) анализатора:

Шаг 1. Пронумеровать правила продукций. Кодировать активные префиксы.

Шаг В. При кодировании магазинному символу соответствует несколько вхождений, то это SLR(1)-грамматика и таблица действий f(a) задается алгоритмом.

Шаг 2. Построение управляющей таблицы M = [f(u), g(x)];

1. Построить пополненную грамматику G' для исходной грамматики G.
2. Построить отношение OBLOW (be low входит под) грамматических вхождений G.
3. Построить КА g(x) на основе отношения OBLOW. Недетерменированный КА преобразовать в детерменированный. Выполнить Шаг В.

Шаг 4. Применить алгоритм перенос-свёртка.

II. Построение SL(0) анализатора на основе LR(0)-ситуаций, функций замыкания CLOSURE и перехода GOTO.

Шаг 1. Построение управляющей таблицы M = [f(u), g(x)];

1.Пронумеровать правила продукций. Построить пополненную грамматику G' для исходной грамматики G.

2.Построить множества ситуаций и построение КА для множества ситуаций каждое множество ситуаций - состояние КА;

3.Построить отношение действий f(u) на основе КА.

Шаг 3. Применить алгоритм перенос-свёртка**.**

I. Построение анализаторов для LR(k)-грамматики без ε-правил на основе активных префиксов и отношения OBLOW

Пусть грамматические вхождения символов X и Z в правую часть i-го правила, а Yj - грамматическое вхождение символа Y в правую часть j-го правила. Определим множество OFIRST(Y_i) (входит первым), в которое включим Y_j и все грамматические вхождения, с которых могут начинаться цепочки, выводимые и Y:

OFIRST(Yi) = {Yj} U {Xi | Y =>* Aβ => Xαβ и Xi самое левое грамматическое вхождение в правую часть правила A → X}

Если в грамматике есть правила с пустой правой частью, то на последующем шаге вывода они не применяются.

Алгоритм нахождения множества OFIRST(Y_{ij}):
Вход: G'= (T, V', P', S'), Y_{ij} – грамматическое вхождение.
Выход: VT – множество грамматических вхождений, находящихся первыми после Y_{ij}.

VT := {Y_{ij}}
foreach($A_k \rightarrow X_{k1}\alpha \in P' \mid A_k$ = symbol (Y_{ij}) ∈ V') // symbol(Y_{ij}) = Y
VT := VT ∪ OFIRST(X_{k1})
end

Определим отношение OBLOW (входит под) используя множество OFIRST, следующим образом:

Xi OBLOW Yj - это множество {(Xi, Yj) | A→ αXiZjβ∈P и Yj ∈OFIRST(Zi)}
⊥ OBLOW Yj - это множество {(⊥, Yj) | Yj OFIRST(S_0)}, где S_0 - начальное вхождение.

Отношение Xi OBLOW Yj определяет множество Q грамматических вхождений Xi, для которых представляющие их магазинные символы могут встретиться в магазине непосредственно под символом, представляющим Yj.

Отношение OBLOW будем представлять в виде матрицы, содержащей n столбцов и (n+1) рядов, где n - число грамматических вхождений пополненной грамматики G. Первые n рядов матрицы отмечены грамматическими вхождениями, а последний ряд - маркером дна стека. Если Xi OBLOW Yj, то элемент матрицы, расположенный в ряде Xi и столбце Yj равен 1.

Алгоритм построения матрицы отношения OBLOW:
Вход: G '= (T, V', P', S').
Выход: Γ – множество всех грамматических вхождений (и маркер дна), OBLOW – матрица размера (n+1)×n, где $n=\sum^{|P'|}_{(i=1)} (|\alpha_i|, A_i \rightarrow \alpha_i \in P')$.

Γ {⊥}
foreach ($A_i \rightarrow \{X_{i1}, X_{i2},\ldots, X_{ik}\} \in P'$)
  foreach ($X_{ij} \in \{X_{i1}, X_{i2},\ldots, X_{ik}\}$)
    Γ:= Γ∪{X_{ij}}
    foreach (Y ∈ {OFIRST(X_{ij}) | A_i = S'}) // см. определение выше
      OBLOW[⊥,Y] := 1
    end
    foreach(Y ∈ {OFIRST($X_{i(j+1)}$) | j ≠ k}) // см. определение выше
      OBLOW[X_{ij}, Y] := 1
    end
  end
end

Пример. Построим матрицу отношения OBLOW для грамматики с правилами:

p_1: S → aAb p_2: S → c p_3: S → bS
p_4: A → Bb p_5: B → aA p_6: B → c
Непосредственно из правил вывода грамматики p_1 и p_4 получим:

A_1 OBLOW b_1, B_4 OBLOW b_4

Из определения отношения OBLOW следует, что ⊥ OBLOW Yj тогда и только тогда, когда Yj ∈ OFIRST(S_0). Из S можно вывести цепочки S => $a_1A_1b_1$ и S => c_2. Следовательно OFIRST(S_0) = {a_1,c_2, S_0} и ⊥ OBLOW a_1, ⊥ OBLOW c_2 и ⊥ OBLOW S_0.

Для правила p_1: a_1 OBLOW Yj, для всех Yj ∈ OFIRST(A_1) из A можно вывести цепочки A => b_3S_3 и A => b_4B_4 => b_6B_4 , A => b_4B_4 => $a_5A_5b_4$.

Следовательно, OFIRST(A_1) = {a_5,b_3,c_6,A_1,B_4} и a_1 OBLOW a_5, a_1 OBLOW b_3, a_1 OBLOW c_6, a_1 OBLOW A_1, a_1 OBLOW B_4

Вычислим OBLOW для правил p_3 и p_6, получим матрицу отношения OBLOW.

	S_0	a_1	A_1	b_1	c_2	b_3	S_3	B_4	b_4	a_5	A_5	c_6
S_0												
a_1			1			1		1		1		1
A_1				1								
b_1												
c_2												
b_3		1			1		1					
S_3												
B_4									1			
b_4												
a_5						1		1		1	1	1
A_5												
c_6												
⊥	1	1			1							

Шаг 1. Пример кодирования активных префиксов. Для каждого магазинного символа (за исключением S_0 и ⊥), кодируемая цепочка является *префиксом* правой части некоторого правила грамматики. Например, магазинный символ «$\&_1$», находящийся на вершине магазина, означает что:

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

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

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

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

P = { p_1: S→ F & L, p_2: S→ (S), p_3: F→ * L, p_4: F→ i, p_5: L→F }

Таблица 1. Закодированные символы для заданной грамматики.

Символ грамматики	Магазинный Символ	Кодируемая цепочка	Операции
S	S_0 S_2	⊥S (S	Д П
F	F_1 F_5	F F	П C_5
L	L_1 L_3	F&L *L	C_1 C_3
I	i_4	i	C_4
(	$(_2$	(	П
*	$*_3$	*	П
&	$&_1$	F&	П
)	$)_2$	(S)	C_2

Шаг В. Кодируемому магазинному символу соответствует единственное вхождение (детерменированный КА).

Определить функция действий F(u) на множестве $(V_p \cup \{\perp\})\times(T \cup \{\varepsilon\})$ по правилам:

1. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где T кодирует цепочку β, F(u) = C(i) = (СВЕРТКА, i).
   Если магазинному символу г соответствует только одно грамматическое вхождение Xi, являющееся самым правым вхождением в i-е правило вывода грамматики G, то все элементы ряда, помеченной г, имеют значение (СВЕРТКА, i).
2. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где г кодирует некий префикс цепочки β (но не саму основу), F(u) = Перенос.
3. Для конфигурации (S_0, ⊥, π), где S_0 кодирует цепочку ⊥S, F(u) = ДОПУСК.
4. В противном случае, F(u) = ОШИБКА.

Шаг 2. Алгоритм. Построение управляющей таблицы M для LR(0) грамматики. Таблица M состоит из матрицы действий f(u) и переходов КА g(X) и, которые строятся отдельно, а затем объединяются.

Вход: LR(0)-грамматика G = (T, V, P, S) не содержащая ε - правил, таблица 1 закодированных символов.

Выход: Управляющая таблица M для грамматики G.

1. Построить пополненную грамматику G' для исходной грамматики G.
2. Вычислить отношение OBLOW грамматических вхождений грамматики G.
3. Построить отношение переходов g(x).

Отношение переходов g(x) удобно представлять в виде таблицы, содержащей по одному столбцу для каждого символа T∪V и одной строке для каждого грамматического вхождения грамматики и маркера дна. Элемент в строке с помеченным грамматическим вхождением X_{ij} и столбце, отмеченном символом грамматики Y, должен содержать все грамматические вхождения, для которых справедливо отношение X_{ij} OBLOW Y_j.

Аргументами функции переходов g является символы $X \in (V \cup T)$ и цепочка $u \in (V_p \cup \{\perp\})^*$, её значениями – элементы множества {ОШИБКА} ∪ V_p, где V_p – множество специальных символов, соответствующих грамматическим вхождениям или их множествам.

Алгоритм построения матрицы **g(X) конечного автомата**

Вход: G'=(T, V', P', S'), Γ - множество грамматических вхождений, матрица OBLOW.

Выход: g(X) - матрица размера |Γ|×|T ∪ V'|, Ɲ - множество грамматических вхождений, приводящих к недетерминированности.

Ɲ :=∅

foreach($X_{ij} \in \Gamma$)

  foreach($Y_{kl} \in \Gamma \mid Y_{kl} \neq \perp$, OBLOW[$X_{ij}$,$Y_{kl}$] = 1)

    g[X_{ij},symbol(Y_{kl})] := g[X_{ij},Y]∪{Y_{kl}} // symbol(Y_{kl}) = Y

    Ɲ := Ɲ ∪ g[X_{ij},symbol(Y_{kl})] || g[X_{ij},symbol(Y_{kl})]| > 1

  end

end

Если некоторые грамматические вхождения содержатся исключительно во множествах с другими грамматическими вхождениями в ячейках, то их не нужно отдельно представлять, как магазинный символ, поскольку нет однозначных переходов к ним. Такие грамматические вхождения можно выделить в отдельное множество Ɲ, чтобы исключить из рассмотрения.

Матрица в каждой ячейке должна содержать не более одного грамматического вхождения, иначе проявляется недетерминированность и нужно определить новые состояния так, чтобы можно было задать магазинный алфавит V_p – каждому состоянию конечного автомата соответствует ровно один магазинный символ v_k.

Алгоритм построения матрицы переходов Ф(u) (примежуточной) на основе матрицы g(X), может быть построен недетерминированный КА:

Вход: G'=(V', T, P', S'), Г – множество грамматических вхождений, g – матрица, построенная на предыдущем этапе, Ɲ – множество грамматических вхождений, приводящих к недетерминированности.

Выход: F(**u**) **-** матрица переходов размера $|V_p| \times |T \cup V'|$, V_p - множество магазинных символов, $M: V_p \rightarrow \Gamma$ – словарь.

$V_p := \emptyset$
foreach ($X_{ij} \in \Gamma - Ɲ$)
$v_x = GET\_VP(\{X_{ij}\}, M)$
$V_p = V_p \cup \{v_x\}$
foreach ($s \in (T \cup V')$)
$v_d = GET\_VP(g[X_{ij},s], M) \mid g[X_{ij},s] \neq \emptyset$
$V_p = V_p \cup \{v_d\}$
$\Phi[v_x, s] = v_d$
// Разрешает недетерминированность
foreach($Y_{kl} \in g[X_{ij},s] \parallel g[X_{ij}, s] \mid > 1$)
foreach (s' ∈ (T ∪ V'))
$\Phi[v_d,s'] = GET\_VP(g[Y_{kl},s'], M) \mid g[Y_{kl},s'] \neq \emptyset$
end
end

end
end

Алгоритм. GET_VP(θ, M) – преобразовать множество грамматических вхождений θ (недетерминированность) в один магазинный символ v_k.

Вход: θ – множество грамматических вхождений, $M: V_p \rightarrow \Gamma$ – словарь.

Выход: v_θ – магазинный символ, соответствующий множеству θ.

if (θ ∉ M)
M[index(θ)] = θ // index(θ) – строковое представления множества
end
return index(θ)

Алгоритм. Построение матрицы функции действий F(u).

Вход: G'=(V', T, P', S'), V_p – множество магазинных символов, M– словарь отображения $V_p \rightarrow \Gamma$, Ф – матрица переходов размера $|V_p| \times |T \cup V'|$.

Выход: F - матрица действий размера $|V_p| \times \|T \cup \{\varepsilon\}|$

foreach ($v \in V_p$)
foreach(t ∈ {T U { ε }})
if (M[v] = {S_0 } и t = ε) // (0) S' → S
F[v,t] = ДОПУСК
else if (Ф[v,t] ≠ ∅ и t ≠ ε)
F[v,t] = ПЕРЕНОС
else if (t ∈ FOLLOW(v) и ($X_{ij} \in M[v] : j = |a|$ ($A_i \rightarrow \alpha X_{ij}$)) !!
F[v,t] = (СВЕРТКА, i)
else

exit // Если ни один из рассмотренных выше случаев не выполнен, то это значит, что построить функцию действий нельзя – грамматика не является LR(1)-грамматикой.

end

end

end

Шаг 4. Алгоритм перенос-свертка. Выполнение алгоритма описывается в терминах конфигураций, представляющих собой тройки вида ($\perp\alpha T$, ax, π), где αT – цепочка магазинных символов (T-верхний символ магазина), ax – необработанная часть входной цепочки, π выход (строка из номеров правил), построенный к настоящему моменту времени.

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

Управление алгоритмом осуществляется двумя функциями, задаваемых в таблице:

1. По верхнему символу магазина и входному символу, определяется функция действия: *ПЕРЕНОС или СВЕРТКА;*
2. *ПЕРЕНОС* определяет значение функции перехода, равное магазинному символу, который нужно втолкнуть в магазин;
3. СВЕРТКА(i), однозначно определяет правило i для свертки.

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 – ошибка.

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 r, к цепочке u:

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

f(u) = (СВЕРТКА, i) (C) 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' ϵ $V_p \cup \{\perp\}$, и выдать номер правила i на входную ленту. Определить символ T''= g(A) из строки таблицы M, отмеченной символом T', записать его в магазин и перейти к шагу 3.

f(u) = ОШИБКА (О). Выдать сообщение об ошибке и перейти к шагу 5.

f(u) = ДОПУСК (Д). Объявить цепочку, записанную на входной ленте, правым разбором входной цепочки z.

5. Останов.

Шаг В. При кодировании магазинному символу соответствует несколько вхождений, то это SLR(1)-грамматика и таблица действий f(a) задается алгоритмом**.**

Вход: LR(0)-грамматика G = (T,V,P, S), не содержащая ε-правил.

Выход: Множество TLR(0)-таблиц для грамматики G или сообщение о том, что грамматика G не является LR(0) грамматикой.

Для магазинного символа Г, представляющего множество грамматических вхождений Q, и входного символа *a* значение функции действий f(*a*) определяется следующим образом:

- если начальное вхождение $S_0 \in Q$ и *a* = ε, то f(ε) = ДОПУСК;
- если *a* ≠ ε и g(a) ≠ ОШИБКА, то f(a) = ПЕРЕНОС;
- если $X_j \in Q$ — самое правое грамматическое вхождение i-го правила A → αX_j и *a* ∈ FOLLOW(A), то f(a) = (СВЕРТКА, i). Значение остальных элементов для f(*a*) — ОШИБКА.

Если имеется множество грамматических вхождений, не удовлетворяющих перечисленным в шаге 4 условиям, то грамматика не принадлежит классу SLR(1).

Пример построения LR(0)-грамматики. Добавление продукций изображенных красным цветом вызывают неоднозначность, что требует построения анализатора для LR(1)-грамматики.

Построим управляющую таблицу LR(0) анализатора для следующей грамматики:

G = {T,V,P,S}, где T = {+,(,),I,*}, V = {S,T,P},

P: (1) S → S + T

(2) S → T

(3) T →T*P // для этих правил LR(0)-анализатор не справляется

(4) T → P

(5) P → (E)

(6) P→ i

Шаг 1. Определим пополненную грамматику G' для грамматики G':

G' = {T,V U {S'}, P U {S' → S}, S}

Шаг 2. Найдем множество грамматических вхождений Г и матрицу отношения OBLOW для грамматики G' согласно алгоритму 2:

Вход: G' = {T,V',P',S}

Выход: Г– множество грамматических вхождений, OBLOW – матрица отношения “входит под”.

Будем рассматривать правила пополненной грамматики G' как:

P': $S'_0 \rightarrow S_{01}$

$S_1 \rightarrow S_{11} +_{12} T_{13}$

$S_2 \rightarrow T_{21}$

$T_3 \rightarrow T_{31} *_{32} P_{33}$

$T_4 \rightarrow P_{41}$

$P_5 \rightarrow (_{51} S_{52})_{53}$

$P_6 \rightarrow i_{61}$

Шаг 0. Г := {⊥}

Шаг 1. для цепочки правила $S'_0 \rightarrow S_{01}$:

Г := Г U {S_{01}} = {⊥, S_{01}}

Это пополненное правило, поэтому в матрицу OBLOW необходимо записать, что под маркером дна ⊥ содержатся грамматические вхождения из $OFIRST(S_{01})$.

$OFIRST(S_{01})$ = { S_{01}, S_{11}, T_{21}, T_{31}, P_{41}, $(_{51}$, i_{61}}

$OBLOW[\perp,S_{01}] = 1$; $OBLOW[\perp,S_{11}] = 1$; $OBLOW[\perp,T_{21}] = 1$; $OBLOW[\perp,T_{31}] = 1$;

$OBLOW[\perp,P_{41}] = 1$; $OBLOW[\perp,(_{51}] = 1$; $OBLOW[\perp,i_{61}] = 1$;

Шаг 2. для цепочки правила $S_1 \rightarrow S_{11} +_{12} T_{13}$:

Г := Г U { S_{11}, $+_{12}$, T_{13}} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}}

1. $OFIRST(+_{12}) = \{+_{12}\}$

   $OBLOW[S_{11},+_{12}] = 1$;
2. $OFIRST(T_{13}) = \{ T_{13}, T_{31}, P_{41}, (_{51}, i_{61} \}$

   $OBLOW[+_{12}, T_{13}] = 1$; $OBLOW[+_{12}, T_{31}] = 1$; $OBLOW[+_{12}, P_{41}] = 1$; $OBLOW[+_{12}, (_{51}] = 1$;

   $OBLOW[+_{12}, i_{61}] = 1$;

Шаг 3. для цепочки правила $S_2 \rightarrow T_{21}$:

Г := Г U {T_{21}} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}}

Шаг 4. для цепочки правила $T_3 \rightarrow T_{31} *_{32} P_{33}$:

Г := Г U { $T_{31} *_{32} P_{33}$} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}, T_{31}, $*_{32}$, P_{33}}

1. $OFIRST(*_{32}) = \{*_{32}\}$

   $OBLOW[T_{31},*_{32}] = 1$
2. $OFIRST(P_{33}) = \{P_{33}, (_{51}, i_{61} \}$

   $OBLOW[*_{32}, P_{33}] = 1$; $OBLOW[*_{32}, (_{51}] = 1$; $OBLOW[*_{32}, i_{61}] = 1$;

Шаг 5. для цепочки правила $T_4 \rightarrow P_{41}$:

Г := Г U { P_{41}} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}, T_{31}, $*_{32}$, P_{33}, P_{41}}

Шаг 6. для цепочки правила $P_5 \rightarrow (_{51} E_{52})_{53}$:

Г := Г U { $(_{51}$, S_{52}, $)_{53}$} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}, T_{31}, $*_{32}$, P_{33}, P_{41}, $(_{51}$, S_{52}, $)_{53}$}

1. $OFIRST()_{53}) = \{)_{53}\}$

   $OBLOW[)_{53},)_{53}] = 1$
2. $OFIRST(S_{52}) = \{ S_{11}, T_{21}, T_{31}, P_{41}, (_{51}, S_{52}, i_{61}\}$

   OBLOW

Шаг 7. для цепочки правила $P_6 \rightarrow i_{61}$:

Г := Г U { i_{61}} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}, T_{31}, $*_{32}$, P_{33}, P_{41}, $(_{51}$, S_{52}, $)_{53}$, i_{61}}

Множество грамматических вхождений:

Г := Г U { i_{61}} = {⊥, S_{01}, S_{11}, $+_{12}$, T_{13}, T_{21}, T_{31}, $*_{32}$, P_{33}, P_{41}, $(_{51}$, S_{52}, $)_{53}$, i_{61}}

Матрица OBLOW.

Г	S_{01}	S_{11}	$+_{12}$	T_{13}	T_{21}	T_{31}	$*_{32}$	P_{33}	P_{41}	$(_{51}$	S_{52}	$)_{53}$	i_{61}
⊥	1	1			1	1			1	1			1
S_{01}													
S_{11}			1										
$+_{12}$				1		1			1	1			1
T_{13}													
T_{21}													
T_{31}							1						
$*_{32}$								1		1			1
P_{33}													
P_{41}													
$(_{51}$		1			1	1			1	1	1		1
S_{52}													
$)_{53}$												1	
i_{61}													

Алгоритм построения матрицы функции переходов g(x).

Вход: G'= (V', T, P', S'), Г - множество грамматических вхождений, матрица OBLOW.

Выход: g - матрица размера $|\Gamma| \times |T \cup V'|$. N – множество грамматических вхождений, приводящих к недетерминированности.

Шаг 0. N = 0

Шаг 1. для ⊥:

g [⊥,S] = { S_{01}, S_{11}}

g [⊥,T] = { T_{21}, T_{31}}

g [⊥,P] = { P_{41}}

g [⊥,(] = { $(_{51}$}

g [⊥,i] = { i_{61}}

Т.к. g [⊥,S] и g [⊥,T] содержит несколько грамматических вхождений, все их нужно добавить в множество N = N ∪ { S_{01}, S_{11}} ∪ { T_{21}, T_{31}} = { S_{01}, S_{11}, T_{21}, T_{31}}

Шаг 2. для S_{11}

g [S_{11}, +] = { $+_{12}$}

Шаг 3. для $+_{12}$

g [$+_{12}$, T] = { T_{13}, T_{31}}

g [$+_{12}$, P] = { P_{41}}

g [$+_{12}$, (] = { $(_{51}$}

g [$+_{12}$, i] = { i_{61}}

Т.к. g [$+_{12}$, T] содержит несколько грамматических вхождений, все их нужно добавить в множество N = N ∪ { T_{13}, T_{31}}= { S_{01}, S_{11}, T_{21}, T_{31}, T_{13}}

Шаг 4. для T_{31}

g [T_{31}, *] = {$*_{32}$ }

Шаг 5. для $*_{32}$

g [$*_{32}$, P] = { P_{33}}

g [$*_{32}$, (] = { $(_{51}$}

g [$*_{32}$, i] = { i_{61}}

Шаг 6. для $(_{51}$

g [$(_{51}$, S] = { S_{11}, S_{52}}

g [$(_{51}$, T] = { T_{21}, T_{31}}

g [$(_{51}$, P] = { P_{41}}

g [$(_{51}$, (] = { $(_{51}$}

g [$(_{51}$, i] = { i_{61}}

g [$(_{51}$, S] и g [$(_{51}$, T] содержит несколько грамматических вхождений, все их нужно добавить в множество N = N ∪ { S_{11}, S_{52}} ∪ { T_{21}, T_{31}} = { S_{01}, S_{11}, T_{21}, T_{31}, T_{13}, S_{52}}

Шаг 7. для $)_{53}$ g [$)_{53}$,)] = { $)_{53}$}

	S	+	T	*	P	(	)	i
⊥	{S_{01}, S_{11}}		{T_{21}, T_{31}}		{P_{41}}	$(_{51}$		i_{61}
S_{01}								
S_{11}		{$+_{12}$}						
$+_{12}$			{T_{13}, T_{31}}		{P_{41}}	$(_{51}$		i_{61}
T_{13}								
T_{21}								
T_{31}				{$*_{32}$}				
$*_{32}$					P_{33}	$(_{51}$		i_{61}
P_{33}								
P_{41}								
$(_{51}$	{S_{11}, S_{52}}		{T_{21}, T_{31}}		P_{41}	$(_{51}$		i_{61}
S_{52}								
$)_{53}$							$)_{53}$	
i_{61}								

Полученное множество грамматических вхождений Ɲ, приводящих к недетерминированности:

$$N = \{ S_{01}, S_{11}, T_{21}, T_{31}, T_{13}, S_{52}\}$$

Применим матрицу g для построения матрицы Φ функции переходов g(a), зададим множество магазинных символов V_p и словарь отображения M: $V_p \rightarrow \Gamma$.

Алгоритм построения матрицы переходов g(x):

Вход: G'= (V', T, P', S'), Γ – множество грамматических вхождений, g – матрица, построенная на предыдущем этапе, Ɲ - множество грамматических вхождений, приводящих к недетерминированности.

Выход: Φ – матрица переходов размера $|V_p| \times |T \cup V'|$, V_p - множество магазинных символов, M – словарь, представляющий отображение $V_p \rightarrow \Gamma$.

Шаг 1. Для ⊥: M[⊥]≔ {⊥}

Рассмотрим все s∈ (T∪V'), для которых заданы переходы из ⊥:

1. s=S: g[⊥,S] = {S_{01},S_{11} }, M[v_1] = {S_{01},S_{11}} Φ[⊥, S] = v_1

Поскольку магазинный символ v_1 задает несколько грамматических вхождений, необходимо составить таблицу для него:

1) Для S_{01} – нет грамматических вхождений для перехода.

2) Для S_{11}: g[S_{11},+] = {$+_{12}$}, M[$+_{12}$] = {$+_{12}$} Φ[v_1, +] = $+_{12}$

2. s=T: $g[\perp,T]=\{T_{21,}\ T_{31}\}$, $M[v2\ ]=\{T_{21,}\ T_{31}\}$ $\Phi[\perp, T] = v2$

Поскольку магазинный символ v_2 задает несколько грамматических вхождений, необходимо составить таблицу для него:

1) Для T_{21} – нет грамматических вхождений для перехода.

2) Для T_{31}: $g[T_{31},*] = \{*_{32}\}$, $M[*_{32}] = \{*_{32}\}$ $\Phi[v_2, *] = *_{32}$

3. s=P: $g[\perp, P\ ]=\{\ P_{41}\}$, $M[P_{41}] =\{\ P_{41}\}$ $\Phi[\perp, P\ ] = P_{41}$

4. s=(: $g[\perp, (\ ]=\{(_{51}\}$, $M[(_{51}] = \{(_{51}\}$ $\Phi[\perp, (\ ]= (_{51}$

5. s=i: $g[\perp, i] = \{\ i_{61}\ \}$, $M[i_{61}\ ] = \{\ i_{61}\}$ $\Phi[\perp, i]= i_{61}$

Шаг 2. Для $+_{12}$: $M[+_{12}\ ]=\{+_{12}\ \}$ (был задан ранее)

Рассмотрим все $s \in (T\cup V')$, для которых заданы переходы из $+_{12}$:

1. s=U: $g[+_{12},T]=\{T_{13,}\ T_{31}\}$, $M[v3\ ]:=\{T_{13,}\ T_{31}\}$ $\Phi\ [+_{12}, T] = v3$

Поскольку магазинный символ v_3 задает несколько грамматических вхождений, необходимо составить таблицу для него:

1) Для T_{13} – нет грамматических вхождений для перехода.

2) Для T_{31}: $g[T_{31},*]=\{*_{32}\}$, $M[*_{32}]:=\{*_{32}\}$ (задан ранее) $\Phi[v_3, *]= *_{32}$

2. s=(: $g[+_{12}, (\ ]=\{(_{31}\ \}$, $M[(_{31}\ ]=\{(_{31}\}$ (задан ранее) $\Phi\ [+_{12}, (\ ]= (_{31}$

3. s=i: $g[+_{12},i]=\{i_{61}\ \}$, $M[i_{61}\ ]=\{i_{61}\}$ (задан ранее) $\Phi\ [+_{12},i]= i_{61}$

4. s=P: $g[+_{12},P]=\{\ P_{41}\ \}$, $M[P_{41}\ ]=\{\ P_{41}\}$ (задан ранее) $\Phi\ [+_{12},i]= P_{41}$

Шаг 3. Для T_{31}: $M[T_{31}]=\{\ T_{31}\}$ Рассмотрим все $s \in (T\cup V')$, для которых заданы переходы из $(_{31}$:

1. s= *: $g[T_{31}, *]=\{*_{32}\}$, $M[*_{32}\ ]:=\{*_{32}\}$(задан ранее) $\Phi[T_{31}, *] = *_{32}$

Шаг 4. Для $*_{32}$: $M[*_{32}]=\{\ *_{32}\}$ (задан ранее) Рассмотрим все $s \in (T\cup V')$, для которых заданы переходы из $(_{31}$:

1. s= P: $g[*_{32}, P]=\{\ P_{33}\}$, $M[P_{33}]:=\{\ P_{33}\}$ $\Phi\ [*_{32}, P] = P_{33}$

2. s= (: $g[*_{32}, (]=\{\ (_{51}\}$, $M[(_{51}]:=\{\ (_{51}\}$(задан ранее) $\Phi\ [*_{32}, (] = (_{51}$

3. s= i: $g[*_{32}, i]=\{\ i_{61}\}$, $M[i_{61}]:=\{\ i_{61}\}$(задан ранее) $\Phi[*_{32}, i] = i_{61}$

Шаг 5. Для $(_{51}$: $M[(_{51}]=\{\ (_{51}\}$ (задан ранее). Рассмотрим все $s\in (T\cup V')$, для которых заданы переходы из $(_{31}$:

1. s= S: $g[(_{51}, S]=\{S_{11,}\ S_{52}\}$, $M[v4]:=\{S_{11,}\ S_{52}\}$ $\Phi\ [(_{51}, S] = v4$ Поскольку магазинный символ v_4 задает несколько грамматических вхождений, необходимо составить таблицу для него:

1) Для S_{52} – нет грамматических вхождений для перехода.

2) Для S_{11}: $g[S_{11},+]=\{+_{12}\}$, $M[+_{12}\ ]:=\{+_{12}\}$ (задан ранее) $\Phi\ [v_4, +] =+_{12}$

2. s= T: $g[(_{51}, T]=\{T_{21,}\ T_{31}\}$, $M[v2]:=\{T_{21,}\ T_{31}\}$ $\Phi\ [(_{51}, T] = v2$

3. s= P: $g[(_{51}, P]=\{\ P_{41}\}$, $M[P_{41}]:=\{\ P_{41}\}$ $\Phi\ [(_{51}, P] = P_{41}$

3. s= (: $g[(_{51}, (]=\{\ (_{51}\}$, $M[(_{51}]:=\{\ (_{51}\}$ $\Phi\ [(_{51}, (] = (_{51}$

3. s= i: $g[(_{51}, i]=\{\ i_{61}\}$, $M[i_{61}]:=\{\ i_{61}\}$ $\Phi\ [(_{51}, i] = i_{61}$

Шаг 6. Для $)_{53}$: $M[)_{53}]=\{\ )_{53}\}$. Рассмотрим все $s \in (\ T \cup V')$, для которых заданы переходы из $(_{31}$:

1. s =): $g[T_{31},)]=\{)_{53}\}$, $M[)_{53}]:=\{)_{53}\}$ (задан ранее) $\Phi\ [T_{31},)] =)_{53}$

Полученное множество магазинных символов V_p:

$$V_p = \{\perp, v_1, +_{12}, v2, *_{32,}\ P_{41}, (_{51},\ i_{61}, v3, T_{31}, P_{33}, v4,)_{53}\}$$

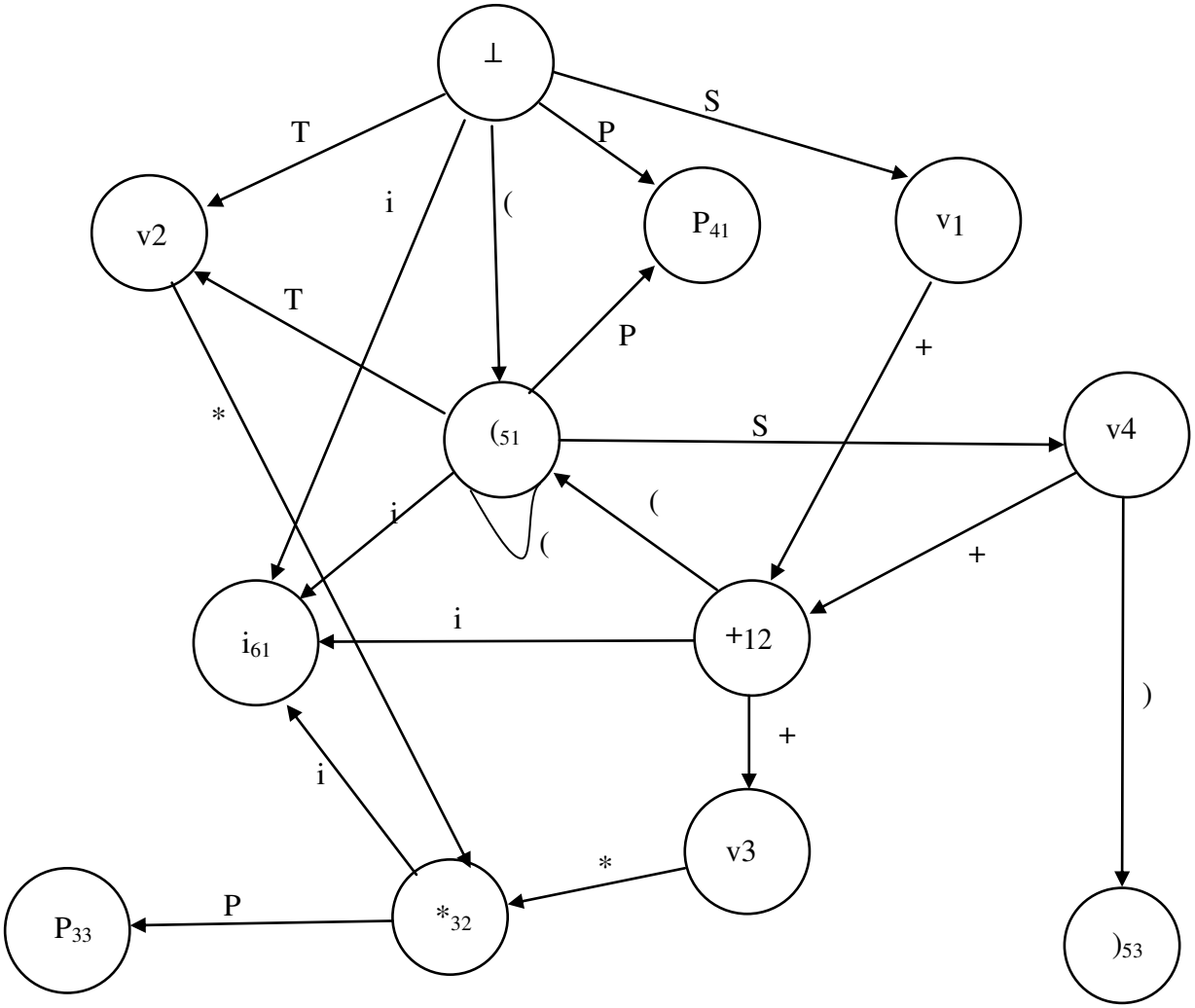
Полученный словарь отображения M: $V_p \rightarrow \Gamma$:

V_p	⊥	v_1	$+_{12}$	v2	$*_{32}$	P_{41}	$(_{51}$	i_{61}	v3	T_{31}	P_{33}	V4	$)_{53}$
Γ	{⊥}	$\{S_{01},S_{11}\}$	$\{+_{12}\}$	$\{T_{21}, T_{31}\}$	$\{*_{32}\}$	$\{P_{41}\}$	$\{(_{51}\}$	$\{i_{61}\}$	$\{T_{13}, T_{31}\}$	$\{T_{31}\}$	$\{P_{33}\}$	$\{S_{11}, S_{52}\}$	$\{)_{53}\}$

Матрица переходов Ф.

	S	+	T	*	P	(	)	i
⊥	v1		v2		P_{41}	$(_{51}$		i_{61}
v_1		$+_{12}$						
$+_{12}$			v3			$(_{51}$		i_{61}
v2				$*_{32}$				
$*_{32}$					P_{33}	$(_{51}$		i_{61}
P_{41}								
$(_{51}$	v4		v2		P_{41}	$(_{51}$		i_{61}
i_{61}								
v3				$*_{32}$				
P_{33}								
v4		$+_{12}$					$)_{53}$	
$)_{53}$								

Матрица переходов Ф задает конечный автомат, с диаграммой, где магазинные символы - вершины, ребра – операции переноса терминального или нетерминального символов:



Шаг 4. Построить матрицу действия F(u) для всех магазинных символов V_p, каждому из которых соответствует один ряд таблицы, согласно **алгоритму**:

Вход: G' = (T,V',P',S'), V_p – множество магазинных символов, M - словарь отображения $V_p \rightarrow$ Г,Z - матрица переходов размера $|V_p| \times |T \cup V'|$.

Выход: F - матрица действий размера $|V_p|$ x |T U {ε}|

Рассмотрим все v ∈ V_p = {⊥, v_1, $+_{12}$, v2, $*_{32}$, P_{41}, $(_{51}$, i_{61}, v3, T_{31}, P_{33},v4, $)_{53}$}

1. v=⊥, M[⊥] = {⊥} ⇒ ∀ t ∈ Ф [v,t], F[⊥,t] = ПЕРЕНОС
2. v=v1, M[v1] = {S_{01},S_{11}} ⇒ ∀ t ∈ Ф [v,t], F[v1,t] = ПЕРЕНОС
3. v = $+_{12}$, M[$+_{12}$] = {$+_{12}$} ⇒ ∀t ∈ Ф [v,t], F[$+_{12}$,t] = ПЕРЕНОС
4. v = v2, M[v2] = {T_{21},T_{31}} ⇒ ∀t ∈ Ф [v,t], F[v2,t] = ПЕРЕНОС
   FOLLOW(T_{21}) = {+,),ε} т.к. T_{21} является правым вхождением правила 2, то ∀ t ∈ FOLLOW(v2), F [v2,t] = СВЕРТКА, 2
5. v = $*_{32}$, M[$*_{32}$] = {$*_{32}$} ⇒ ∀t ∈ Ф [v,t], F [$*_{32}$,t] = ПЕРЕНОС
6. v = P_{41}, M[P_{41}] = {P_{41}}
   FOLLOW(P_{41}) = {+,*,),ε} т.к. P_{41} является правым вхождением правила 4, то ∀ t ∈ FOLLOW(P_{41}), F[P_{41},t] = СВЕРТКА, 4
7. v = $(_{51}$, M[$(_{51}$] = {$(_{51}$} ⇒ ∀t ∈ Ф [v,t], F[$(_{51}$,t] = ПЕРЕНОС
8. v = i_{61}, M[i_{61}] = {i_{61}}
   FOLLOW(i_{61}) = {+,*,),ε} т.к. i_{61} является правым вхождением правила 6, то ∀ t ∈ FOLLOW(i_{61}), F[P_{41},t] = СВЕРТКА, 6
9. v = v3, M[v3] = {T_{13},T_{31}} ⇒ ∀t ∈ Ф[v,t], H[v3,t] = ПЕРЕНОС
   FOLLOW(T_{13}) = {+,),ε} т.к. T_{13} является правым вхождением правила 1, то ∀ t ∈ FOLLOW(T_{13}), F[v3,t] = СВЕРТКА, 2
10. v = T_{31}, M[T_{31}] = {T_{31}} ⇒ ∀ t ∈ Ф[v,t], F[T_{31},t] = ПЕРЕНОС
11. v = P_{33}, M[P_{33}] = {P_{33}}
    FOLLOW(P_{33}) = {+,*,),ε} т.к. P_{33} является правым вхождением правила 3, то ∀ t ∈ FOLLOW(P_{33}), F[P_{33},t] = СВЕРТКА, 3
12. v = v4, M[v4] = {S_{11}, S_{52}}⇒ ∀t ∈ Ф[v,t], F[v4,t] = ПЕРЕНОС
    F[v4, ε] = ДОПУСК
13. v = $)_{53}$, M[$)_{53}$] = {$)_{53}$}
    FOLLOW($)_{53}$) = {+,*,), ε } т.к. $)_{53}$ является правым вхождением правила 5, то ∀ t ∈ FOLLOW($)_{53}$), F[$)_{53}$,t] = СВЕРТКА, 5

Матрица действий F(u):

	+	*	(	)	i	ε
⊥			П		П	
v1	П					Д
$+_{12}$			П		П	
v2	С,2	П		С,2		С,2
$*_{32}$			П		П	
P_{41}	С,4	С,4		С,4		С,4
$(_{51}$			П		П	
i_{61}	С,6	С,6		С,6		С,6
v3	С,1	П		С,1		С,1
P_{33}	С,3	С,3		С,3		С,3

V4	П			П		
$)_{53}$	С,5	С,5		С,5		С,5

Управляющая таблица М.

Г	F(u)						g(x)						
	+	*	(	)	i	ε	S	+	T	*	P	(	)
⊥			П		П		v1		v2		P_{41}	$(_{51}$	
v_1	П					Д		$+_{12}$					
12			П		П				v3			$(_{51}$	
v2	С,2	П		С,2		С,2				$*_{32}$			
$*_{32}$			П		П						P_{33}	$(_{51}$	
P_{41}	С,4	С,4		С,4		С,4							
$(_{51}$			П		П		v4		v2		P_{41}	$(_{51}$	
i_{61}	С,6	С,6		С,6		С,6							
v3	С,1	П		С,1		С,1				$*_{32}$			
P_{33}	С,3	С,3		С,3		С,3							
v4	П			П				$+_{12}$					$)_{53}$
$)_{53}$	С,5	С,5		С,5		С,5							

Шаг 4. Применение алгоритма перенос-свертка.

Разница между LR(0) и LR(1) анализаторами

LR(0) требует только канонических элементов для построения таблицы синтаксического анализа, LR(1) требует символ просмотра вперед.

LR(1) позволяет выбирать корректные операции свертки и использовать грамматику, которая не является однозначно обратимой, а LR(0) - нет.

LR(0) операции свертки для всех элементов, тогда как LR(1) только для символов просмотра вперед.

LR(0) = канонические элементы и LR(1) = LR(0) **+** символ просмотра вперед.

В канонической коллекции LR(0) состояние/элемент имеет вид:

Например. $I_2 = \{C \rightarrow AB\bullet\}$

тогда как в канонической форме LR(1) состояние/элемент имеет вид:

Например. $I_2 = \{C \rightarrow d\bullet, e/d\}$, где e и d - символ просмотра вперед.

Построение канонической формы LR(0) начинается с C = { CLOSURE ({S'→ S})}, тогда как построение канонической коллекции LR(1) начинается с C = { CLOSURE ({S'→S, $}) } где S — начальный символ.

В LR(0) может возникнуть конфликт в таблице синтаксического анализа, в LL(1) используется просмотр вперед, чтобы избежать ненужных конфликтов в таблице синтаксического анализа.

Алгоритм LR(0) проще, чем алгоритм LR(1).

II. Построение анализатора LR(0)-грамматик на основе LR(0)-ситуаций: функции замыкания CLOSURE и перехода GOTO

Определение. LR(0) ситуация - это правило грамматики с точкой в некоторой позиции правой части, например [A → w_1 • w_2], если A → w_1w_2 – правило КС-грамматики.

Пример. Для правила S → (S) можно получить 4 ситуации:

[S → • (S)]

[S → (•S)]

[S → (S•)]

[S → (S)•]

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

Утверждение. Для каждого активного префикса существует непустое множество ситуаций.

Для простого LR(0)-анализатора вначале на основе КС-грамматики строится детерминированный конечный автомат (КА) для распознавания активных префиксов: ситуации группируются в множества, изменяющие состояния анализатора.

Шаг 1. Определение ситуаций и построение конечного автомата

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.

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

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

Пример 1. Исходная КС - грамматика : G({ V = {S,L,F}, T = { i,),=,(,*} P, S)
P = { S→ F=L, S→ L, F→ (*L), F→ i, L→ F }

Пополненная грамматика G_0 содержит еще одно правило: S' → S

Замыкание множества ситуаций

Пусть вначале множество ситуаций CLOSURE(I_0) содержит одну ситуацию *S′* → • *S,* т.е. CLOSURE(I_0) ={ S' → • S }

Ситуация S'→ •S содержит точку перед символом S, поэтому по второму правилу в CLOSURE(I_0) необходимо включить *S*-правила с точкой слева. Окончательно получим:

I_0 = CLOSURE({S' → •S})

Шаг 1: I_0 = {S' → •S}

Шаг 2: $I_0 = I_0 \cup$ {S → •F=L, S → •L}

Шаг 3: $I_0 = I_0 \cup \{F \rightarrow \bullet\ (*L), F \rightarrow \bullet i\}$

Шаг 4: $I_0 = I_0 \cup \{L \rightarrow \bullet F\}$

$I_0 = \{S' \rightarrow \bullet S, S \rightarrow \bullet F{=}L, S \rightarrow \bullet L, F \rightarrow \bullet\ (*L), F \rightarrow \bullet i, L \rightarrow \bullet F\}$

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

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

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

Пусть для грамматики G_0 некоторое множество $I = \{[S' \rightarrow S\bullet], [S \rightarrow F\bullet{=}\&L]\}$.

Для вычисления значения $GOTO(I_0, =)$ необходимо рассмотреть ситуации, в которых непосредственно сразу за точкой следует символ =. Это ситуация [S → F•=L]. Перемещая точку за символ =, построим множество ситуаций [S → F=•L] и замыкание этого множества: {L → •F, F → •(*L), F → •i }. Тогда $GOTO(I_2, =) = \{S \rightarrow F{=}\bullet L, L \rightarrow \bullet F, F \rightarrow \bullet\ (*L), F \rightarrow \bullet i\} = I_6$

Функция переходов GOTO

$I_0 = CLOSURE(\{S' \rightarrow \bullet S\}) = \{S' \rightarrow \bullet S, S \rightarrow \bullet F{=}L, S \rightarrow \bullet L, F \rightarrow \bullet(*L), F \rightarrow \bullet i, L \rightarrow \bullet F\})$

$GOTO(I_0, S) = \{S' \rightarrow S\bullet\} = I_1$

$GOTO(I_0, F) = \{S \rightarrow F\bullet{=}L, L \rightarrow F\cdot\} = I_2$

$GOTO(I_0, L) = \{S \rightarrow L\bullet\} = I_3$

$GOTO(I_0, i) = \{F \rightarrow i\bullet\} = I_4$

$GOTO(I_0, (\ ) = \{F \rightarrow (\bullet *L)\} = I_5$

$GOTO(I_2, =) = \{S \rightarrow F{=}\bullet L, L \rightarrow \bullet F, F \rightarrow \bullet\ (*L), F \rightarrow \bullet i\} = I_6$

$GOTO(I_5, *) = \{F \rightarrow (*\bullet L), L \rightarrow \bullet F, F \rightarrow \bullet\ (*L), F \rightarrow \bullet i\} = I_7$

$GOTO(I_6, L) = \{S \rightarrow F{=}L\bullet\} = I_8$

$GOTO(I_6, F) = \{L \rightarrow F\bullet\} = I_9$

$GOTO(I_6, (\ ) = \{F \rightarrow (\bullet *L)\} = I_5$

$GOTO(I_6, i) = \{F \rightarrow i\bullet\} = I_4$

$GOTO(I_7, L) = \{F \rightarrow (*L\bullet)\} = I_{10}$

$GOTO(I_7, F) = \{L \rightarrow F\bullet\} = I_9$

$GOTO(I_7, (\ ) = \{F \rightarrow (\bullet *L)\} = I_5$

$GOTO(I_7, i) = \{F \rightarrow i\bullet\} = I_4$

$GOTO(I_{10},)\ ) = \{F \rightarrow (*L)\ \bullet\} = I_{11}$

Каноническая форма множества ситуаций

Построение канонической системы множеств *LR(0)*– ситуаций:

1 φ = ∅

2. Включить в φ множество $I_0 = CLOSURE([S' \rightarrow \bullet\ S])$, которое в начале «не отмечено».

3. Если множество ситуаций I, входящее в систему , «не отмечено», то:

- отметить множество I;
- вычислить для каждого символа $X \in (V \cup \Sigma)$ значение I' = GOTO(I,X);

- если множество I' ≠ ∅ и еще не включено в φ, то включить его в систему множеств как «неотмеченное» множество.

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

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

Определив множество I_0 = CLOUSURE ([S' → •S]) = {S'→ •S}) = {S' → •S, S → •F=L, S → •L, F → •(*L), F → •i, L→ •F}, включим его в систему φ в качестве «неотмеченного» множества.

φ = { I_0 = {S' → •S, S → •F=L, S → •L, F → • (*L), F → •i, L → •F}}

Отметим множество I_0 и определим множества GOTO(I_0, X) для всех символов X грамматики G_0:

I_1 = GOTO (I_0, S) = CLOUSURE ([S' → S•]) = { S' → S•}

Множество I_1 непустое и отсутствует в системе φ. Включим I_1 в φ как «неотмеченное» множество.

φ = {I_0 = {S' → •S, S → •F=L, S → •L, F → • (*L), F → •i, L → •F},
I_1 = {S' → S•}}

Продолжая строить GOTO(I_0, X), получаем:

I_2 = GOTO (I_0, F) = CLOUSURE ([S → •F=L]) = { S → F•=L, L → F• }

I_3 = GOTO (I_0, L) = {S → L•}

I_4 = GOTO (I_0, i) = {F → i•}

I_5 = GOTO (I_0, () ={F → (•*L)}

Остальные множества GOTO(I_0, X), где X ϵ {S,*,),=}, пусты, поэтому система φ принимает вид:

φ = {I_0 = {S' → •S, S → •F=L, S → •L, F → • (*L), F → •i, L → •F},
I_1 = {S' → S•}}
I_2 = GOTO (I_0, F) = CLOUSURE ([S → •F=L]) = { S → F•=L, L → F• }
I_3 = GOTO (I_0, L) = {S → L•}
I_4 = GOTO (I_0, i) = {F → i•}
I_5 = GOTO (I_0, () ={F → (•*L)}
}

Продолжая выполнять шаг 3 построения системы множеств, «отмечаем» множество I_1. Для всех X ∈ (V ∪ Σ) множества GOTO(I_1, X) пусты.

Теперь отмечаем множество I_2. GOTO(I_2, =) = {S → F= •L, L → •F, F → • (*L), F → •i} = I_6

отмечаем множество I_5. GOTO(I_5, *) = {F → (*•L), L → •F, F → • (*L), F → •i} = I_7

Добавляем множество I_6 в φ.

GOTO(I_6, L) = {S → F=L•} = I_8
GOTO(I_6, F) = {L → F•} = I_9
GOTO(I_6, ( ) = {F → (•*L)} = I_5
GOTO(I_6, i) = {F → i•} = I_4

Добавляем в φ только те множества, которые в φ отсутствуют. Продолжая аналогичным образом, включаем в систему φ множества:

GOTO(I_7, L) = {F → (*L•)} = I_{10}
GOTO(I_{10},)) = {F → (*L) •} = I_{11}

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

φ = {

I_0 = {S' → •S, S → •F=L, S → •L, F → • (*L), F → •i, L → •F},

I_1 = {S' → S•},

I_2 = {S → F•=L, L → F•},

I_3 = {S → L•},

I_4 = {F → i•},

I_5 = {F → (•*L)},

I_6 = {S → F=•L, L → •F, F → • (*L), F → •i},

I_7 = {F → (*•L), L → •F, F → • (*L), F → •i},

I_8 = {S → F=L•},

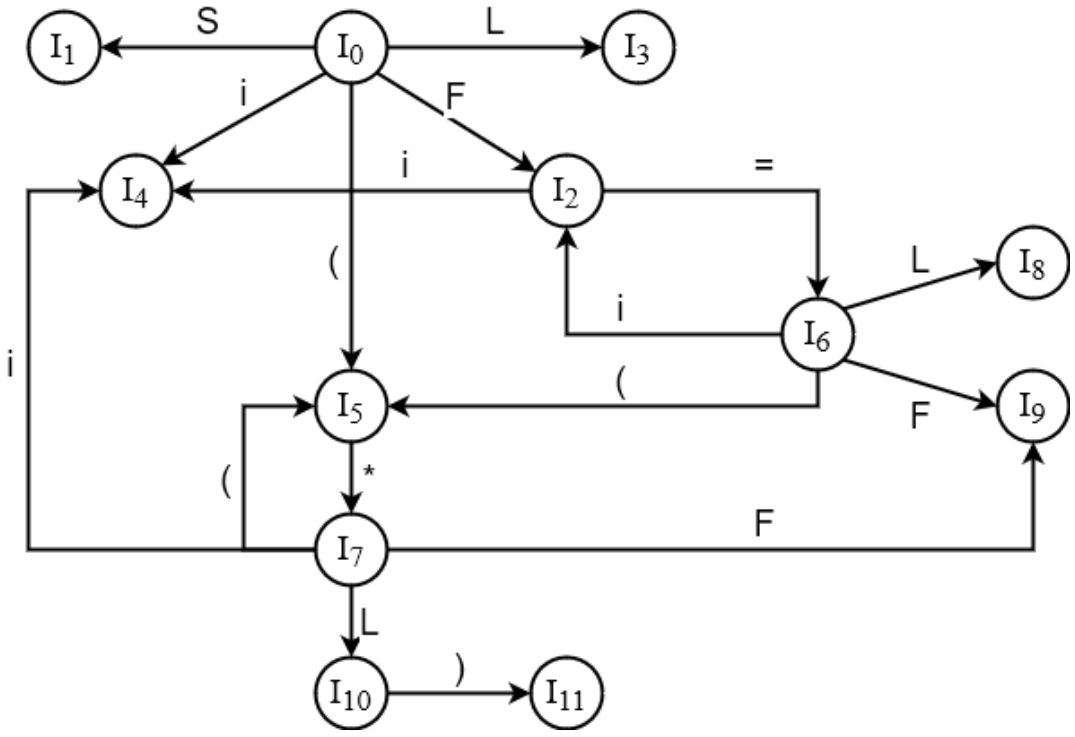
I_9 = {L → F•},

I_{10} = {F → (*L•)},

I_{11} = {F → (*L) •}

}

Используя каноническую систему LR(0)–множеств, можно представить функцию GOTO в виде диаграммы детерминированного конечного автомата. Диаграмма переходов ДКА для активных префиксов грамматики G_0:



Переход в состояние, которому на диаграмме соответствует лист (вершина, не имеющая исходящих дуг) однозначно определяет операцию Свёртки (i) по правилу i с переходом в новое состояние определяемое переходм на КА левым не териналом правила i, все остальные переходы операцию Переноса .

Шаг 2. Построение управляющей таблицы. Алгоритм построения управляющей таблицы M для LR(0)-грамматик основывается на рассмотрении пар грамматических вхождений, которые могут быть представлены соседними магазинными символами в процессе разбора допустимых цепочек.

1. Если операция[s_m, a_i] = Перенос(s), синтаксический анализатор выполняет перенос в стек очередного состояния s и его конфигурация становится

(s_0s_1... s_ms, a_{i+1}... a_n\$)

Символ a_i хранить в стеке не нужно (он может быть восстановлен из s). Текущим входным символом становится a_{i+1}.

2. Если операция[s_m, a_i] = Свертка(i) правила p_i: A→β, синтаксический анализатор выполняет свертку в два шага и его конфигурация становится

(s_0s_1... s_{m-r}s, a_ia_{i+1}... a_n\$)

здесь r - длина β, а s = GOTO[s_{m-r}, A].

2.1. Определяется правило для свертки i и левый нетерминал: p_i: A→ α.

2.2. Синтаксический анализатор снимает r символов состояний с вершины стека, что переносит на вершину стека состояние s_{m-r}. При свертке текущий входной символ не изменяется. (Удаляется из верхней части магазина |α| символов в соответствии с правилом С(i), где i - номер правила, A→ α, |α| - длина правой части правила).

2.3. После чего на вершину стека помещается s, запись из GOTO[s_{m-r}, A]. (По правилу для свертки i и левый нетерминал: p_i: A→ α , определяется по таблице переходов состояние, которое должно быть занесено в стек)

Последовательность символов грамматики X_{m-r+1}...X_m всегда соответствует α, правой части продукции свертки.

3. Если операция[s_m, a_i] = допуск синтаксический анализ завершается.

4.Если операция[s_m, a_i] = ошибка синтаксический анализатор вызывает программу восстановление после ошибки.

Управляющая таблица

I	f(u)						g(x)		
	i	=	*	(	)	⊥	S	F	L
0	П, 4			П, 5			1	2	3
1						Д			
2		П, 6			С, 5	С, 5			
3						С, 2			
4		С, 4			С, 4	С, 4			
5			П, 7						
6	П, 4			П, 5				9	8
7	П, 4			П, 5				9	10
8						С, 1			
9					С, 5	С, 5			
10					П, 11				
11		С, 3			С, 3	С, 3			

Шаг 3. Применение алгоритма «перенос-свёртка» для разбора цепочки символов на ленте.

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

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

Распознавание цепочек:

1. (0, (*(*i))=(*i)⊥, ε) $\vdash^{П}$ (0 5, *(*i))=(*i)⊥, ε) $\vdash^{П}$ (0 5 7, (*i))=(*i)⊥, ε) $\vdash^{П}$ (0 5 7 5, *i))=(*i)⊥, ε) $\vdash^{П}$ (0 5 7 5 7, i))=(*i)⊥, ε) $\vdash^{П}$ (0 5 7 5 7 4,))=(*i)⊥, ε) $\vdash^{С}$ (0 5 7 5 7 9,))=(*i)⊥, 4) $\vdash^{С}$ (0 5 7 5 7 10,))=(*i)⊥, 4 5) $\vdash^{П}$ (0 5 7 5 7 10 11,)=(*i)⊥, 4 5) $\vdash^{С}$ (0 5 7 9,)=(*i)⊥, 4 5 3) $\vdash^{С}$ (0 5 7 10,)=(*i)⊥, 4 5 3 5) $\vdash^{П}$ (0 5 7 10 11, =(*i)⊥, 4 5 3 5) $\vdash^{С}$ (0 2, =(*i)⊥, 4 5 3 5 3) $\vdash^{П}$ (0 2 6, (*i)⊥, 4 5 3 5 3) $\vdash^{П}$ (0 2 6 5, *i)⊥, 4 5 3 5 3) $\vdash^{П}$ (0 2 6 5 7, i)⊥, 4 5 3 5 3) $\vdash^{П}$ (0 2 6 5 7 4,)⊥, 4 5 3 5 3) $\vdash^{С}$ (0 2 6 5 7 9,)⊥, 4 5 3 5 3 4) $\vdash^{С}$ (0 2 6 5 7 10,)⊥, 4 5 3 5 3 4 5) $\vdash^{П}$ (0 2 6 5 7 10 11, ⊥, 4 5 3 5 3 4 5) $\vdash^{С}$ (0 2 6 9, ⊥, 4 5 3 5 3 4 5 3) $\vdash^{С}$ (0 2 6 8, ⊥, 4 5 3 5 3 4 5 3) $\vdash^{С}$ (0 1, ⊥, 4 5 3 5 3 4 5 3 1) $\vdash^{Д}$

2. (0, i=i⊥, ε) $\vdash^{П}$ (0 4, =i⊥, ε) $\vdash^{С}$ (0 2, =i⊥, 4) $\vdash^{П}$ (0 2 6, i⊥, 4) $\vdash^{П}$ (0 2 6 4, ⊥, 4) $\vdash^{С}$ (0 2 6 9, ⊥, 4 4) $\vdash^{С}$ (0 2 6 8, ⊥, 4 4 5) $\vdash^{С}$ (0 1, ⊥, 4 4 5 1) $\vdash^{Д}$

3. (0, (*(*(*i)))⊥, ε) $\vdash^{П}$ (0 5, *(*(*i)))⊥, ε) $\vdash^{П}$ (0 5 7, (*(*i)))⊥, ε) $\vdash^{П}$ (0 5 7 5, *(*i)))⊥, ε) $\vdash^{П}$ (0 5 7 5 7, (*i)))⊥, ε) $\vdash^{П}$ (0 5 7 5 7 5, *i)))⊥, ε) $\vdash^{П}$ (0 5 7 5 7 5 7, i)))⊥, ε) $\vdash^{П}$ (0 5 7 5 7 5 7 4,)))⊥, ε) $\vdash^{С}$ (0 5 7 5 7 5 7 9,)))⊥, 4) $\vdash^{С}$ (0 5 7 5 7 5 7 10,)))⊥, 4 5) $\vdash^{П}$ (0 5 7 5 7 5 7 10 11,))⊥, 4 5) $\vdash^{С}$ (0 5 7 5 7 9,))⊥, 4 5 3) $\vdash^{С}$ (0 5 7 5 7 10,))⊥, 4 5 3 5) $\vdash^{П}$ (0 5 7 5 7 10 11,)⊥, 4 5 3 5) $\vdash^{С}$ (0 5 7 9,)⊥, 4 5 3 5 3) $\vdash^{С}$ (0 5 7 10,)⊥, 4 5 3 5 3 5) $\vdash^{П}$ (0 5 7 10 11, ⊥, 4 5 3 5 3 5) $\vdash^{С}$ (0 4, ⊥, 4 5 3 5 3 5 3) $\vdash^{С}$ (0 2, ⊥, 4 5 3 5 3 5 3 4) $\vdash^{С}$ (0 3, ⊥, 4 5 3 5 3 5 3 4 5) $\vdash^{С}$ (0 1, ⊥, 4 5 3 5 3 5 3 4 5 2) $\vdash^{Д}$

3.8. Грамматики предшествования

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

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

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

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

Выделяют следующие виды грамматик предшествования:

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

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

Грамматики простого предшествования

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

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

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

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

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

Для грамматик простого предшествования известны следующие полезные свойства:

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

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

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

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

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

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

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

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

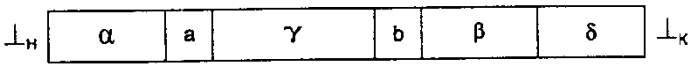


Рис. 12.5. Отношения между символами входной цепочки в грамматике предшествования

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

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

- $\mathbf{L}(A) = \{X \mid \exists\ A \Rightarrow^* Xz\}$, $A \in \mathbf{VN}$, $X \in \mathbf{V}$, $z \in \mathbf{V}^*$ — множество крайних левых символов относительно нетерминального символа A (цепочка z может быть и пустой цепочкой);
- $R(A) = \{X \mid \exists\ A \Rightarrow^* zX\}$, $A \in \mathbf{VN}$, $X \in \mathbf{V}$, $z \in \mathbf{V}^*$ — множество крайних правых символов относительно нетерминального символа A.

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

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

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

Шаг 1. $\forall A \in VN$:

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

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

Шаг 2. $\forall A \in VN$:

$R_i(A) = R_{i-1}(A) \cup R_{i-1}(B), \forall B \in (R_{i-1}(A) \cap VN)$,

$L_i(A) = L_{i-1}(A) \cup L_{i-1}(B), \forall B \in (L_{i-1}(A) \cap VN)$.

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

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

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

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

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

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

Здесь S — целевой символ грамматики.

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

Алгоритм «перенос-свертка» для грамматики простого предшествования

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

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

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

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

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

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

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

Ошибка в процессе выполнения алгоритма возникает, когда невозможно

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

Пример 1.

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

Дана грамматика $G(\{a, (,)\}, \{S, R\}, P, S)$, с правилами P:
1) $S \rightarrow (R|a$; 2)$R \rightarrow Sa)$. Построить распознаватель для строки $(((aa)a)a)\perp_к$.

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

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

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

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

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

Таблица 1.2 – Матрица предшествования символов грамматики

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

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

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

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

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

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

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

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

Шаг 4. Получили следующую цепочку вывода:
$S\Rightarrow(R\Rightarrow(Sa)\Rightarrow((Ra)\Rightarrow((Sa)a)\Rightarrow(((Ra)a)\Rightarrow(((Sa)a)a)\Rightarrow(((aa)a)a)$.

Пример 2.

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

Построим крайние левые и правые:

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

Дополняем к T входящие a,b:

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

11. R(R) = {T,R, a,b}
12. R(T) = {a,b}

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

Процесс распознавания работает как LR анализатор:

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

Пример 3.
I → TS;
T → int | char
S → *BR | BR | B
R → , S
B → i | j
Построим крайние левые и правые:
L(I) = { T, int ,char}
L(T) = {int, char}
L(S) = {*, B, i,j}
L(R) = {,}
L(B) = {i,j}
R(I) = {;}
R(T) = {int, char}
R(S) = {R,B, i, j,S}
R(R) = {S, R, B, i,j}
R(B) = {I,j}
Строим таблицы:

	T	S	;	int	char	*	B	R	,	i	j	Ik
T		=•				<•	<•			<•	<•	
S			=• •>									
;												
int		•>				•>	•>			•>	•>	
char		•>				•>	•>			•>	•>	
*							=•			<•	<•	
B			•>					=•	<•			
R			•>									
,												
I			•>					•>	•>			
J			•>					•>	•>			
In	<•			<•	<•							

В таблице имеет два предшествования. Это на пересечении ; и S.

Следовательно правило нужно переписать:

I → TS;

T → int | char

S → ,*BR | ,BR | ,B

B → i | j

Тогда буква S должна исчезнуть.

R(R) = {R, B, i,j}

Грамматики операторного предшествования

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

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

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

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

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

Отношения предшествования для грамматик операторного предшествования определены таким образом, что для них выполняется еще одна особенность —

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

Для грамматик операторного предшествования также известны следующие свойства:

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

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

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

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

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

$\mathbf{L}^t(A) = \{t \mid \exists\ A \Rightarrow^* tz$ или $\exists\ A \Rightarrow^* Ctz\ \}$, где $t \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $z \in \mathbf{V}^*$;

$\mathbf{R}^t(A) = \{t \mid \exists\ A \Rightarrow^* zt$ или $\exists\ A \Rightarrow^* ztC\ \}$, где $t \in \mathbf{VT}$, $A,C \in \mathbf{VN}$, $z \in \mathbf{V}^*$.

Тогда определения отношений операторного предшествования будут выглядеть так:

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

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

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

Шаг 1. ∀ A∈**VN**:

$\mathbf{R}^t_0(A) = \{t \mid A \to ytB$ или $A \to yt,\ t \in \mathbf{VT},\ B \in \mathbf{VN},\ y \in \mathbf{V}^*\}$,

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

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

Шаг 2. $\forall\ A \in \mathbf{VN}$:

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

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

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

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

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

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

$\perp_H \lessdot\bullet\ a,\ \forall\ a \in \mathbf{VT}$, если $\exists\ S \Rightarrow^* ax$ или $\exists\ S \Rightarrow^* Cax$, где $S,C \in \mathbf{VN}$, $x \in \mathbf{V}^*$ или если $a \in \mathbf{L}^t(S)$;

$\perp_K \bullet\gtrdot\ a,\ \forall\ a \in \mathbf{VT}$, если $\exists\ S \Rightarrow^* xa$ или $\exists\ S \Rightarrow^* xaC$, где $S,C \in \mathbf{VN}$, $x \in \mathbf{V}^*$ или если $a \in \mathbf{R}^t(S)$.

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой операторного предшествования. Поскольку она содержит только терминальные символы, то, следовательно, будет иметь меньший размер, чем аналогичная матрица для грамматики простого предшествования. Следует отметить, что напрямую сравнивать матрицы двух грамматик нельзя — не всякая грамматика простого предшествования является грамматикой операторного предшествования, и наоборот. Например, рассмотренная далее в примере грамматика операторного предшествования не является грамматикой простого предшествования (читатели могут это проверить самостоятельно). Однако если две грамматики эквивалентны и задают один и тот же язык, то их множества терминальных символов должны совпадать. Тогда можно утверждать, что размер матрицы для грамматики операторного предшествования всегда будет меньше, чем размер матрицы эквивалентной ей грамматики простого предшествования.

Все, что было сказано выше о способах хранения матриц для грамматик простого предшествования, в равной степени относится также и к грамматикам

операторного предшествования, с той только разницей, что объем хранимой матрицы будет меньше.

Алгоритм «перенос-свертка» для грамматики операторного предшествования

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

Алгоритм состоит из следующих шагов.

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

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

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

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

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и самым верхним терминальным символом в стеке, то надо прервать выполнение алгоритма и сообщить об ошибке.

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

Пример

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

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

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

$E \to \sim E \mid F$

$F \rightarrow (E) \mid a$

Построить: распознаватель для G.

Построение множеств $L(A)$ и $R(A)$:

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

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

i	$L^t(A)$	$R^t(A)$
0	$L^t_0(S)=\{\wedge\}$ $L^t_0(T)=\{\&\}$ $L^t_0(E)=\{\sim\}$ $L^t_0(F)=\{(, a\}$	$R^t_0(S)=\{\wedge\}$ $R^t_0(T)=\{\&\}$ $R^t_0(E)=\{\sim\}$ $R^t_0(F)=\{), a\}$
1	$L^t_1(S)=\{\wedge, \&, \sim, (, a\}$ $L^t_1(T)=\{\&, \sim, (, a\}$ $L^t_1(E)=\{\sim, (, a\}$ $L^t_1(F)=\{(, a\}$	$R^t_1(S)=\{\wedge, \&, \sim,), a\}$ $R^t_1(T)=\{\&, \sim,), a\}$ $R^t_1(E)=\{\sim,), a\}$ $R^t_1(F)=\{), a\}$
2	$L^t_2(S)=\{\wedge, \&, \sim, (, a\}$ $L^t_2(T)=\{\&, \sim, (, a\}$ $L^t_2(E)=\{\sim, (, a\}$ $L^t_2(F)=\{(, a\}$	$R^t_2(S)=\{\wedge, \&, \sim,), a\}$ $R^t_2(T)=\{\&, \sim,), a\}$ $R^t_2(E)=\{\sim,), a\}$ $R^t_2(F)=\{), a\}$

Матрица операторного предшествования символов грамматики :

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

&	•>	•>	<•	<•	•>	<•	•>
~	•>	•>	<•	<•	•>	<•	•>
(	<•	<•	<•	<•	=•	<•	
)	•>	•>			•>		•>
A	•>	•>			•>		•>
$\perp_i$	<•	<•	<•	<•		<•	

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

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. см. Алгоритм "Перенос-Свертка". В литературе описано несколько вариантов грамматик предшествования.

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, ∑, R, L, B, δ, q_0,)

Q - множество состояний

∑ - входной алфавит

R, L - правые и левые символы, не входящие в Q∪∑, B - пробел (B ∈ ∑)

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},

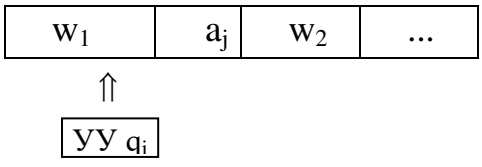
$\delta = Q \times \sum$ в $Q \times (\sum \cup \{\mathbf{B}, L, R\})$ $q_i a_j \rightarrow q_k \{\mathbf{B}, L, R\}$

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

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

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 $L=\{w \in \sum^* \mid q_0$ и $w \Rightarrow^* C_f$, где C_f - заключительная конфигурация.$\}$

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

4.2. Контекстно-зависимые грамматики и ленточные автоматы

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

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

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

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

5. Соотношения между классами грамматик и языками

Отношения между классами КС-грамматик

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

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

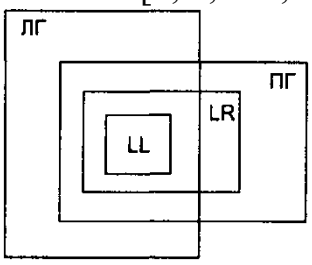


Рис. 12.10. Соотношение классов левоанализируемых и правоанализируемых КС-грамматик

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

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

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

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

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

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

Рис. 12.11. Схема взаимосвязи некоторых классов КС-грамматик

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

Отношения между классами КС-языков

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

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

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

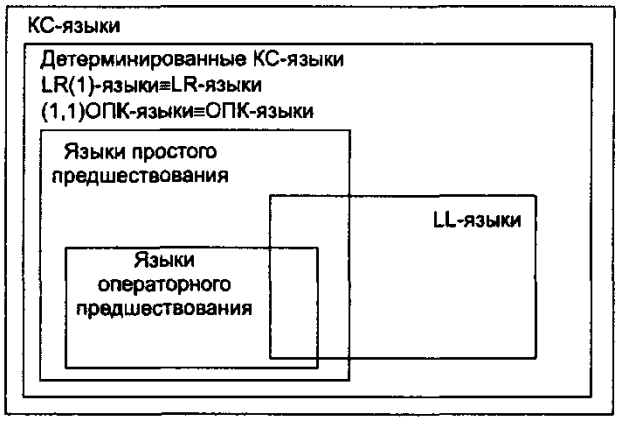


Рис. 12.12. Соотношение между различными классами КС-языков

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

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

Языки простого предшествования, в свою очередь, также являются собственным подмножеством LR-языков, а языки операторного предшествования — собственным подмножеством языков простого предшествования. Интересно, что языки операторного предшествования представляют собой более узкий класс, чем языки простого предшествования.

В то же время языки простого предшествования и LL-языки несопоставимы между собой: существуют языки простого предшествования, которые не являются LL-языками, и в то же время существуют LL-языки, которые не являются языками простого предшествования. Однако существуют языки, которые одновременно являются и языками простого предшествования, и LL-языками. Аналогичное замечание относится также к соотношению между собой языков операторного предшествования и LL-языков.

Можно еще отметить, что язык арифметических выражений над символами a и Ь, заданный грамматикой G({+.-,/,*,a,b},{S,T,E},**P**,S), **P** = {S→S+T|S-T|T, T→T*E|T/E|E, E→(S)|a|b), который многократно использовался в примерах в данном учебном пособии, подпадает под все указанные выше классы языков. Из приведенных ранее примеров можно заключить, что этот язык является и LL-языком, и языком операторного предшествования, а следовательно, и языком простого предшествования и, конечно, LR(1)-языком. В то же время этот язык по мере изложения материала пособия описывался различными грамматиками, не все из которых могут быть отнесены в указанные классы. Более того, он может быть задан с помощью грамматики, которая не являлась даже однозначной.

Таким образом, соотношение классов КС-языков не совпадает с соотношением задающих их классов КС-грамматик. Это связано с неразрешимостью проблем преобразования и эквивалентности грамматик, которые не имеют строго формализованного решения.

Соотношения между типами грамматик:

1) любая регулярная грамматика является КС-грамматикой;

2) любая регулярная грамматика является укорачивающей КС-грамматикой (УКС). Отметим, что УКС-грамматика, содержащая правила вида $A \to \varepsilon$, не является КЗ-грамматикой и не является неукорачивающей грамматикой;

3) любая (приведенная) КС-грамматика является КЗ-грамматикой;

4) любая (приведенная) КС-грамматика является неукорачивающей грамматикой;

5) любая КЗ-грамматика является грамматикой типа 0.

6) любая неукорачивающая грамматика является грамматикой типа 0.

Определение 19. Язык L(G) является *языком типа k*, если его можно описать грамматикой типа k.

Соотношения между типами языков:

1) каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными (например, $L = \{a^n b^n \mid n>0\}$);

2) каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками (например, $L = \{a^n b^n c^n \mid n>0\}$);

3) каждый КЗ-язык является языком типа 0. УКС-язык, содержащий пустую цепочку, не является КЗ-языком. Если язык задан грамматикой типа k, то это не значит, что не существует грамматики типа k' (k'>k), описывающей тот же язык. Поэтому, когда говорят о языке типа k, обычно имеют в виду максимально возможный номер k.

Например, КЗ-грамматика $G_1 = (\{0,1\}, \{A,S\}, P_1, S)$ и КС-грамматика $G_2 = (\{0,1\}, \{S\}, P_2, S)$, где $P_1 = \{S \to 0A1, 0A \to 00A1, A \to \varepsilon\}$ и $P_2 = \{S \to 0S1 \mid 01\}$

описывают один и тот же язык $L = L(G_1) = L(G_2) = \{ 0^n1^n \mid n>0\}$. 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. Промежуточные формы представления программ

Математически перевод определяется как множество пар, причём первый элемент принадлежит множеству объектов, которые требуется перевести, а второй элемент - множеству объектов, являющихся результатом перевода.
Компилятор в целом определяет перевод исходной программы, представленной в виде цепочки символов на языке, в объектный код. Если компиляцию рассматривать как многоэтапный процесс, то можно говорить о переводе, присущем каждому этапу компиляции. Например, в процессе лексического анализа исходная программа преобразуется из цепочки символов в цепочку лексических единиц (лексем), а на этапе синтаксического анализа цепочка лексем преобразуется в некоторую *промежуточную форму,* являющуюся линейной формой записи синтаксического дерева программы. Такой подход имеет определённые преимущества:

- перевод в промежуточную форму позволяет не учитывать особенности целевого языка, которые значительно усложняют перевод;
- упрощается перенос на другие целевые машины, для чего потребуется только реализовать преобразование промежуточной формы в язык новый машины;
- к программе в промежуточной форме можно применять алгоритмы машинно-независимой оптимизации.

Различные формы представления промежуточной программы отличаются друг от друга, в основном, способом соединения операторов и операндов. Общепринятыми промежуточными формами представления программы являются [2, 11, 28–30, 34, 45]:

- **польская инверсная запись;**
- **тетрады;**
- **триады;**
- **связные списочные структуры.**

В настоящее время ряд компиляторов в качестве промежуточной формы программы генерирует файлы, содержащие байт-код (Byte-code), представляющий собой инструкции для виртуальной машины Java (JVM – Java Virtual Machine). Поскольку ядро виртуальной машины Java реализовано практически для любого типа компьютеров, то файлы байт-кодов можно рассматривать как независимые от платформы приложения.

5.1. Польская запись

Одной из основных конструкций языков программирования является выражение, которое:

- может быть описано только рекурсивной самовложенной КС-грамматикой, т. к. оно включает в себя парные символы (скобки), т. е. по своей природе рекурсивно;

- содержит операции, имеющие различное число операндов и выполняющиеся в определенном порядке (приоритет операций);
- может содержать операнды различного типа, что требует дополнительного анализа при переводе.

Перечисленные особенности, с одной стороны, усложняют анализ и перевод выражений, требуют тщательного проектирования описания перевода, а с другой стороны, позволяют использовать выражения в качестве модели при рассмотрении проблем перевода.

5.1.1. Вычисление выражений

Порядок вычисления значения выражения определяется приоритетом операций и наличием в выражении скобок. Для простоты рассмотрим вычисление полноскобочного выражения, т. е. выражения, в котором при помощи дополнительных скобок явно определен порядок выполнения операций. Например выражение

$$A + B - C + D * E$$

в полноскобочной форме может быть записано как

$$(((A + B) - C) + (D * E)).$$

При выполнении вычислений необходимо найти самую внутреннюю пару скобок, выполнить операцию, удалить использованную пару скобок, заменив её вычисленным значением, и повторять этот процесс, пока есть скобки.

Замечание

Если каждой левой скобке придать вес +1, а каждой правой – вес -1, то процесс нахождения выполняемой на данном шаге операции упростится: нужно будет просмотреть выражение и найти максимум веса, который соответствует самой внутренней левой скобке.

Существуют более простые рекурсивные алгоритмы вычисления выражений [2, 11, 28].

Польский математик Ян Лукашевич обнаружил, что при использовании функциональной записи выражения, в которой операция предшествует своим операндам (а не записывается между ними), скобки становятся излишними.

Пример 5.1

Бинарная операция сложения обычно записывается в виде $A + B$. Ее функциональная запись – это $+(A, B)$. При рассмотрении выражений, содержащих только бинарные операции скобки и запятую в записи можно опустить, тогда сложение можно записать как $+AB$.

5.1.2 Префиксная форма

Префиксная форма (или польская запись) формально может быть определена следующим образом:

1. Всякая переменная или константа есть выражение.

2. Если θ_1 – знак унарной (одноместной) операции, а α – выражение, то $\theta_1\alpha$ является выражением.
3. Если θ_2 – знак бинарной (двухместной) операции, а α_1 и α_2 – выражения, то $\theta_2\alpha_1\alpha_2$ является выражением.
4. Если θ_n – знак n -местной операции, а $\alpha_1, \alpha_2, \ldots, \alpha_n$ являются выражениями, то $\theta_n\alpha_1\alpha_2 \ldots \alpha_n$ – выражение.
5. Других выражений не существует.

Пример 5.2

Выражение языка ALGOL-60 $(A + B) * C \uparrow (D/(E + F))$, где $\uparrow$ – обозначение операции возведения в степень, можно представить в префиксной форме как $* + AB \uparrow C/D \ + EF$.

Префиксная форма не нарушает порядка следования операндов в выражении:

A+ B записывается как +AB

B+ A записывается как +BA

т. е. эта форма может быть использована как для записи коммутативных, так и некоммутативных операций. Кроме того, в отличие от обычной алгебраической записи выражений префиксная форма однозначно определяется порядок выполнения операций. Например, выражение $A + B + C + D$ можно интерпретировать по-разному (см. первый столбец табл. 10.1), а префиксная запись выражения всегда однозначно определяет порядок выполнения операций (см. второй столбец табл. 10.1).

Таблица 5.1. *Полноскобочная и префиксная формы выражения*

Полноскобочная форма выражения	**Префиксная форма выражения**
$\big((A + B) + C\big) + D)$	$+ + +ABCD$
$((A + B) + (C + D))$	$+ + AB + CD$
$(\big(A + (B + C)\big) + D)$	$+ + A + BCD$
$(A + ((B + C) + D))$	$+A + +BCD$
$(A + (B + (C + D)))$	$+A + B + CD$

Замечание Обычно в языках программирования операция сложения выполняется слева направо.

Префиксная форма довольно редко используется при проектировании трансляторов это связано с тем, что преобразование выражения в префиксную запись требует его просмотра справа налево, а в большинстве трансляторов просмотр и интерпретация входной программы выполняется слева направо.

5.1.3. Постфиксная форма

В компиляторах чаще всего используют постфиксную форму, в котором операции записываются справа от операндов. Постфиксная форма (польская инверсная запись – ПОЛИЗ) определяется следующими правилами:

1. Всякая переменная или константа есть выражение.

2. Если θ_1 – знак унарной (одноместной) операции, а α – выражение, то $\alpha\theta_1$ является выражением.
3. Если θ_2 – знак бинарной (двухместной) операции, а α_1 и α_2 – выражения, то $\alpha_1\alpha_2\theta_2$ является выражением.
4. Если θ_n – знак n -местной операции, а $\alpha_1, \alpha_2, \ldots, \alpha_n$ являются выражениями, то $\alpha_1\alpha_2 \ldots \alpha_n\theta_n$ – выражение.
5. Других выражений не существует.

Замечание Изображение операции “унарный минус” совпадает с изображением операции “бинарный минус”, то унарный минус можно представлять двумя способами: либо записывать его как бинарный оператор, т. е. $-B$ представлять в виде $0 - B$, либо для унарного минуса ввести новый символ операции, например символ ‘!’.

Пример 5.3
Выражение языка ALGOL-60 $(A + B) * C \uparrow (D/(E + F))$, где $\uparrow$ – обозначение операции возведения в степень, можно представить в польской инверсной записи как $AB + CDEF + /\uparrow *$.

Так же, как и префиксную форму выражения, польскую инверсную запись можно рассматривать как программу работы стековой машины, содержащую команды двух типов:
1. Чтение операнда из выражения вызывает запись его в магазин.
2. Чтение операции влечет за собой выполнение следующих действий:
   - выполнение операции, операнды для которой читаются из верхушки магазина;
   - запись результата операции в магазин.

При вычислении выражения, представленного в постфиксной записи, его элементы читаются слева направо.
Например, выражение $7\ 5\ 2 * +$ соответствует последовательности действий приведённой в табл. 10.2 (рассматриваемый элемент выражения выделен полужирным шрифтом).

Таблица 5.2. *Порядок вычисления выражения 7 5 2 * +*

Выражение	Тип элемента	Действие	Магазин
7 5 2 * +	операнд	Записать в магазин операнд (число 7)	7 ⊥
7 **5** 2 * +	операнд	Записать в магазин операнд (число 5)	5 7 ⊥
7 5 **2** * +	операнд	Записать в магазин операнд (число 2)	2 5 7 ⊥
7 5 2 ***** +	операция (*)	Прочитать из магазина операнд (число 2)	2 5 7 ⊥
		Прочитать из магазина операнд (число 5)	5 7 ⊥
		Выполнить операцию 2 * 5	7 ⊥
		Записать в магазин результат операции (число 10)	10 7 ⊥

7 5 2 * **+**	операция (+)	Прочитать из магазина операнд (число 10)	10 7 ⊥
		Прочитать из магазина операнд число (7)	7 ⊥
		Выполнить операцию 10 + 7	⊥
		Записать в магазин результат операции (число 17)	17 ⊥

5.1.3.1. Свойства ПОЛИЗ

Польская инверсная запись обладает следующими свойствами:

- однозначно определяет порядок выполнения операций;
- имеет простую синтаксическую структуру.

Свяжем с каждым элементом e_i польской инверсной записи выражения вес p в соответствии со следующими правилами:

- если элемент e_i – переменная или константа, то $p(e_i) = 1$;
- если элемент e_i – знак унарной операции, то $p(e_i) = 0$;
- если элемент e_i – знак бинарной операции, то $p(e_i) = -1$;
- если элемент e_i – знак n-местной операции, то $p(e_i) = -(n-1)$;

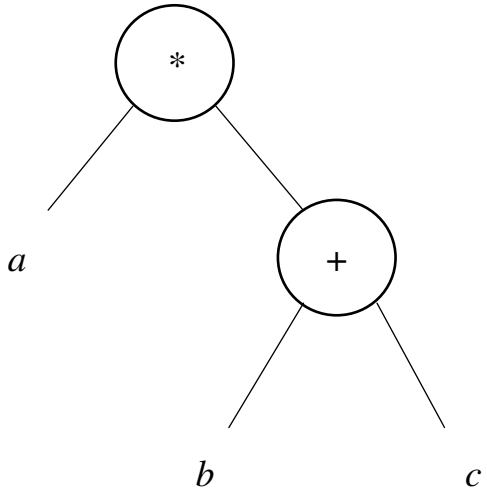
и определим функцию $P(k) = \Sigma\, p(i)$. Тогда для каждого выражения справедливы утверждения:

- $P(i) > 0$ для $1 < i < k$;
- $P(n) = 1$ для правильного выражения из n элементов.

5.1.3.2. Графическое представление выражений

Выражение можно изобразить в виде дерева. Пусть узлы дерева представляют собой операции, а ветви – операнды. Для бинарных операций лева ветвь соответствует левому операнду, а правая ветвь – правому операнду, причём в каждой ветви дерева узлы, которые лежат ниже, соответствуют операция, которые выполняются раньше.

Например, выражения $a * (b + c)$ соответствует дерево, изображенное на рис.



10.1

Рис. 10.1. Дерево выражения $a * (b + c)$

Префиксная форма этого выражения равна $* a + bc$, а постфиксная – $abc +*$. Интересно, что все три формы представления выражения (инфиксная, префиксная и постфиксная) являются, по существу, различными формами представления или обхода построенного бинарного дерева. Можно убедиться, что префиксная польская запись выражения $* +abc$ – это левое скобочное представление дерева, из которого удалены скобки, или *прямой обход* дерева. Аналогично, постфиксная запись $abc +*$ – это правое скобочное представление дерева, из которого удалены все скобки, или *обратный обход* дерева.

5.1.4. ПОЛИЗ как промежуточный язык

Польская инверсная запись выражений имеет два важных свойства, которые позволяют использовать ее не только для представления выражений, но и в качестве промежуточного языка для языковых процессов:

1. Действия, описываемые в ПОЛИЗ, можно выполнять (или программировать) в процессе ее одностороннего безвозвратного просмотра слева направо, т. к. операнды в ПОЛИЗ всегда предшествуют знаку операции.
2. Операнды ПОЛИЗ расположены в том же порядке, что в исходном (инфиксном) выражении. Перевод выражения в ПОЛИЗ сводится только к изменению порядка следования знаков операций.

Расширение ПОЛИЗ для представления других конструкций языков программирования (переменных с индексами, указателей функций, условных выражений, основных операторов языка программирования) выполняется очень просто: нужно придерживаться одного правила – *знак оператора должен следовать непосредственно за соответствующими ему операндами.*

Замечание “вычисление выражения” означает вычисление численного значения выражения в трансляторах интерпретирующего типа, так и *процесс анализа выражения и построения для него объектного кода* для компиляторов.

5.1.4.1. Элементы массивов

Пусть требуется вычислить выражение:

$$(a + b[i, j + 1]) * c + d,$$

в котором $b[i, j + 1]$ – элемент двухмерного массива.

Для представления в постфиксной записи элементов массива необходимо определить дополнительную операцию ПОЛИЗ, вычисляющую, *адрес элемента массива* (код операции SUBS). Операндами операции SUBS являются имя массива и значения индексов. Число операндов этой операции переменно. Оно зависит от размерности массива и определяется по формуле $k = n + 1$, где n – размерность массива. Операнды операции SUBS должны располагаться в определенном порядке: имя массива, значения индексных выражений и

константа k (целое без знака), определяющая число операндов. С учётом представления элементов массива ПОЛИЗ рассматриваемого выражения имеет вид:

$$a\ b\ i\ j\ 1 + 3\ \text{SUBS} + c * d +.$$

Графическое представление этого выражения приведено на рис. 10.2.

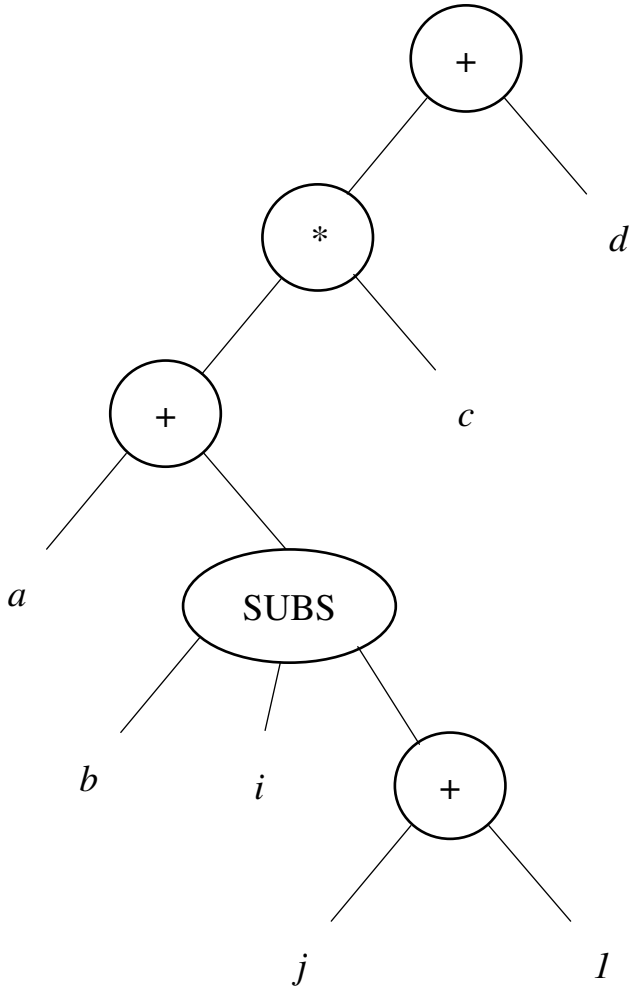


Рис. 10.2. Дерево выражения $(a + b[i, j + 1]) * c + d$

Замечание

При представлении в ПОЛИЗ элементов массивов можно явно не указывать число операндов операции SUBS. В этом случае имя массива должно располагаться непосредственно перед кодом операции, а информация о размерности массива, определяющая число операндов операции, должна храниться в другом месте, например, в таблице идентификаторов языкового процессора.

5.1.4.2. Указатели функций

С точки зрения представления в польской инверсной записи указатель функции (оператор обращения к функции) мало чем отличается от переменной с индексом. Для представления указателя функции введем в ПОЛИЗ дополнительную операцию CALL, имеющую переменное число операндов,

зависящее от числа аргументов в вызове функции. Количество операндов операции CALL определяется выражением:

$$k = n + 1,$$

где n – число аргументов функции.

Например, ПОЛИЗ выражения $a + f(i, j + 1)$ имеет вид $a\ f\ i\ j\ 1 + 3$ CALL $+$ (рис. 10.3).

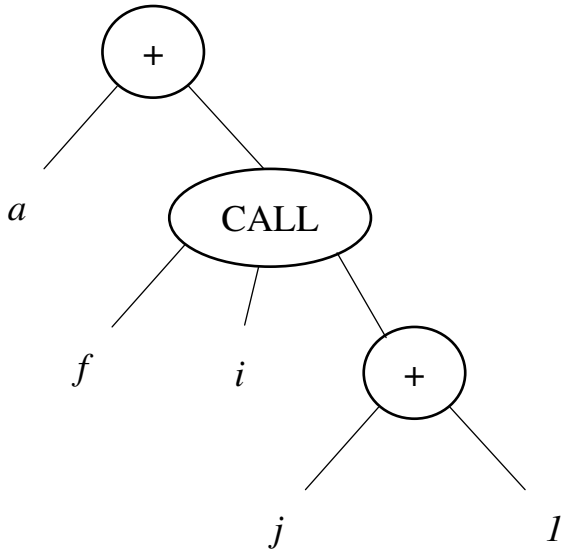


Рис. 10.3. Дерево выражения $a + f(i, j + 1)$

5.1.4.3. Условные выражения

В рассмотренных ранее выражениях порядок выполнения операций определялся приоритетом операций и скобками. В некоторых языках программирования имеются *условные выражения*, в которых порядок выполнения операций и значения операндов определяются *динамически* в зависимости от некоторых условий. Например, в языке C++ выражения $a > b ? 1 : 2$ принимает значение 1, если $a > b$, и значение 2 в противном случае. Условное арифметическое выражение $x + ($ if $a > b$ then 1 else 2) языка ALGOL-60 принимает значение $x + 1$, если $a > b$, и значение $x + 2$ в противном случае.

В приведенных примерах значения операндов при вычислении выражения изменяются динамически в зависимости от значения условия $a > b$.

Для обеспечения возможности представления условных выражений добавим в польскую инверсную запись следующие объекты:

- метки, которые могут помечать некоторые элементы ПОЛИЗ и использоваться для динамического изменения хода вычислений;
- дополнительные операции:
  - BF – бинарная операция “Условный переход по значению ложь”. Выражение $a\ m$ BF означает, что при a равном значению “ложь” необходимо перейти к рассмотрению элемента ПОЛИЗ, помеченного меткой m, в противном случае – перейти к рассмотрению очередного элемента ПОЛИЗ;

- BRL – унарная операция “Безусловный переход”, операндом которой является метка элемента, который будет рассматриваться следующим;
- DEFL – унарная операция “Определить метку”, операндом которой является метка, которую необходимо поместить перед некоторым элементом ПОЛИЗ.

Используя введенные допущения, условное выражение $x +$ (**if** $a > b$ **then** 1 **else** 2) можно представить в виде:

$$x\ a\ b > m_1\ \text{BF}\ 1\ m_2\ \text{BRL}\ m_1\ \text{DEFL}\ 2\ m_2\ \text{DEFL} +$$

На рис 10.4 приведён порядок использования элементов ПОЛИЗ при вычислении рассмотренного условного выражения: верхняя стрелка соответствует случаю $a > b$, а нижние стрелки – $a \leq b$.

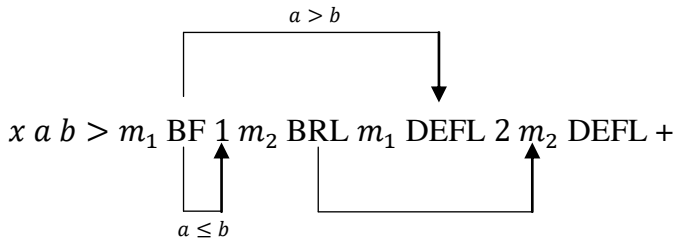


Рис. 5.4. ПОЛИЗ выражения $x +$ (if $a > b$ then 1 else 2)

Для представления условного выражения с помощью дерева требуется ввести пустые узлы, метки и дополнительные операции (BF, BRL, DEFL). Пустые узлы соответствуют разветвлению вычислительного процесса и вводятся для того, чтобы число ветвей, исходящих из узла, было равно числу операндов операции. На рис 10.5 приведено дерево условного выражения $x +$ (**if** $a > b$ **then** 1 **else** 2), при обратном обходе которого получается приведенное ранее представление выражения в ПОЛИЗ.

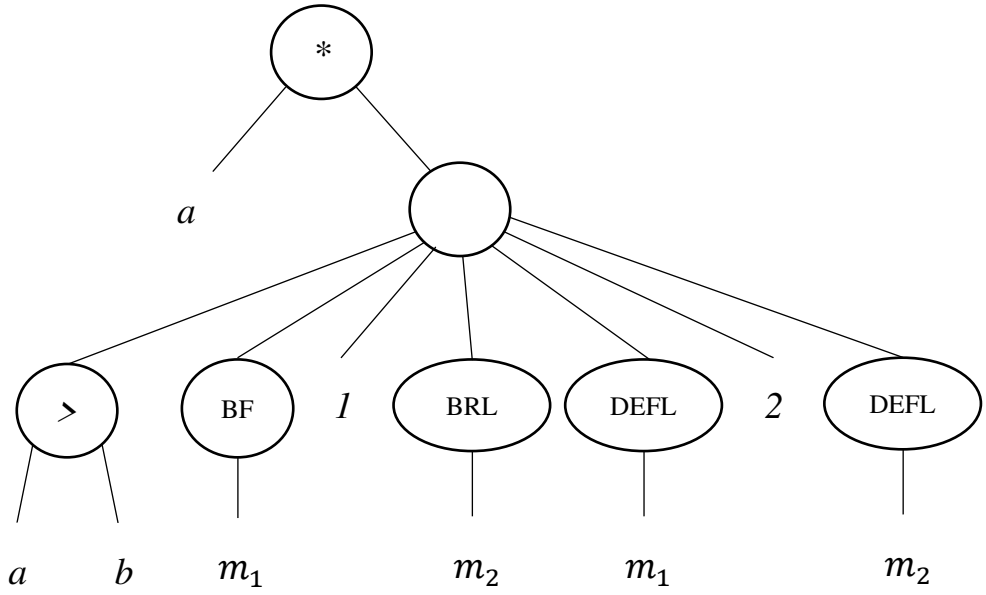


Рис. 10.5. Дерево условного выражения $x +$ (if $a > b$ then 1 else 2)

Замечание

Включение дополнительных объектов, отражающих динамические свойства условного выражения, усложняет построение дерева и его интерпретацию.

5.1.4.4. Оператор присваивания

Для представления оператора присваивания в польской инверсной записи требуется ввести бинарную операцию “Присвоить значение” (:=). Левый операнд операции определяет объект, которому нужно присвоить значение, а правый – вычисленное значение. Например, оператор присваивания языка Паскаль $a[j, k+1] \coloneqq sqr(m[j]) \,/\, 2$ может быть записан в ПОЛИЗ в виде:

$$a\ j\ k\ 1 + 3\ \text{SUBS}\ \text{sqr}\ m\ j\ 2\ \text{SUBS}\ 2\ \text{CALL}\ 2\ /\ \coloneqq .$$

5.1.4.5. Условный оператор

Используя результаты, полученные при рассмотрении условного выражения, условный оператор языка C++, имеющий формат:

if (<выражение>) <оператор1> **else** <оператор2>,

может быть представлен в польской инверсной записи следующим образом:

<выражение> m_1 BF <оператор1> m_2 BRL m_1 DEFL <оператор2> m_2 DEFL

В табл. 10.3 приведено представление основных операций, включенных в большинство языков программирования, в польской инверсной записи. Используя эти операции, можно разработать представление, других, более сложных операторов языков программирования. При необходимости перечисленные операции могут быть модифицированы, а их список расширен.

Таблица 5.3. *Представление основных операций в ПОЛИЗ*

Оператор	**Код операции**	**Операнды**	**Семантика оператора**
Вычисление адреса элемента массива	SUBS	$a, i_1, \dots, i_n, k$	Вычисление адреса элемента массива a; $i_1, \dots, i_n$ – индексные выражения; $k = n + 1$ – число операндов операции
Вызов функции	CALL	$f, x_1, \dots, x_n, k$	Вызов функции f с аргументами $x_1, \dots, x_n$; $k = n + 1$ – число операндов операции
Преобразование типа целый → вещественный	I2A	E	Преобразование типа значения выражение E
вещественный → целый	A2I	E	

Оператор присваивания	$:=$	a, b	Оператор присваивания $b := a$
Определение метки	DEFL	m	Определение метки m
Безусловный переход на метку	BRL	m	Переход на метку m (m – идентификатор метки)
Безусловный переход	BR	n	Безусловный переход на n (n – номер элемента ПОЛИЗ)

Таблица 10.3 *(окончание)*

Оператор	**Код операции**	**Операнды**	**Семантика оператора**
Условный переход по значению “ложь”	BF	r, m	Переход на метку m, если $r =$ **false**
Переход по условию: равно нулю больше нуля меньше нуля не меньше нуля не больше нуля	 BZ BP BM BPZ BMZ	m, r	Переход на m (m – метка или номер элемента ПОЛИЗ) при выполнении соответствующего условия для выражения r
Начало блока	BLBEG	—	Отмечает начало блока
Конец блока	BLEND	—	Отмечает конец блока

Рассмотрим далее разработку представления в ПОЛИЗ конструкций языков программирования, отсутствующих в табл. 10.3, на примере оператор цикла.

5.1.4.6. Оператор цикла

Разработку представления оператора цикла в ПОЛИЗ можно выполнить за два шага. Рассмотрим, например, оператор цикла языка Pascal, имеющего формат:

```
for i := <Выражение_1> to <Выражение_2> do <Оператор>
```

Сначала опишем порядок выполнения оператора **for**, используя условный оператор и оператор перехода:

```
i := <Выражение_1>;
```

```
m1:  if (i ≤ <Выражение_2>)
     then
          <Операторы>
          i := i + 1;
          goto m1
     endif;
```
m_4:

Теперь, зная представление в ПОЛИЗ условного оператора и оператора перехода, можно получить ПОЛИЗ оператора **for**:

i <Выражение_1> := m_1 DEFL i <Выражение_2> – m_4 BP <Оператор> i 1 + m_1 BRL m_4 DEFL

Недостатки этого представления оператора цикла становятся очевидными при разработке описания перевода. Это связано с тем, что часть операторов ПОЛИЗ, формируемая при просмотре заголовка цикла, должна следовать в ней *после* тела цикла, т. е. порядок следования операндов в ПОЛИЗе и в исходной программе не совпадает.

Опишем выполнение оператора цикла другим способом:

```
     i := <Выражение1>;
m1:  if i ≤ <Выражение2>
     then goto m2
     else goto m4
     endif
m3:   i := i + 1;
     goto m1;
m2:   <Операторы>;
     goto m3;
```
m_4:

В этом случае польская инверсная запись имеет вид:

i <Выражение_1> := m_1 DEFL i <Выражение_2> ≤ m_4 BF m_2 BRL m_3 DEFL i i 1+ := m_1 BRL m_2 DEFL <Оператор> m_3 BRL m_4 DEFL

При таком представлении следование операндов в исходном тексте и в ПОЛИЗ совпадает, что упрощает реализацию перевода.

Возможны и другие способы представления. Например, в [26] за счет введение другого набора операций, польская инверсная запись операторов цикла значительно упрощена.

Дополним ПОЛИЗ следующим операциями:

- m_1 m_2 a BRET – безусловный переход с возвратом. Эта операция имеет три операнда:
    - m_1 – метка, на которую должен быть выполнен безусловный переход;
    - m_2 – метка, на которую должен быть выполнен возврат по операции RET;
    - a – вспомогательная переменная, куда должна быть записана метка возврата;

- a RET – операция возврата, которая выполняет операцию перехода на метку, записанную в переменной a.

Пример 5.4

Рассмотрим выполнение польской инверсной записи:

m_1 DEFL <Операторы_1> $m_2\ m_1\ a$ BRET <Операторы_2> m_2 DEFL <Операторы_3> a RET

при условии, что представление элементов <Операторы_1>, <Операторы_2>, <Операторы_3> в ПОЛИЗ известно:

- выполняются <Операторы_1>;
- выполняется оператор BRET: значение метки m_1 записывается во вспомогательную переменную a, выполняется переход к элементу ПОЛИЗ, помеченному меткой m_2;
- выполняются <Операторы_3>;
- операция возврата RET выполняет переход к элементу ПОЛИЗ, метка которого записана в переменной a, т. е. к элементу, помеченному меткой m_1 (рис. 10.6), и процесс повторяется.

Заметим, что <Операторы_2> никогда не выполняются.

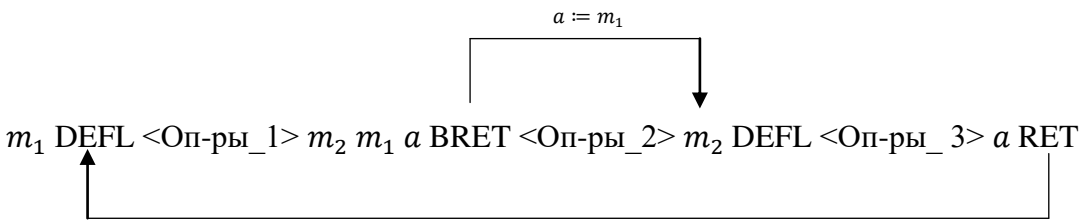


Рис. 5.6. Порядок выполнения ПОЛИЗ из примера 10.4

Используя введенные операции, рассматриваемый оператор цикла

for i := <Выражения_1> **to** <Выражения_2> **do** <Оператор>

Можно представить в ПОЛИЗ следующим образом:

i <Выражения_1> := $m_1\ i$ <Выражения_2> $\leq m_4$ BF $m_2\ m_1\ a$ BRET m_2 DEFL <Оператор> a RET m_4 DEFL.

Порядок выполнения ПОЛИЗ для этого оператора цикла приведен на рис. 10.7.

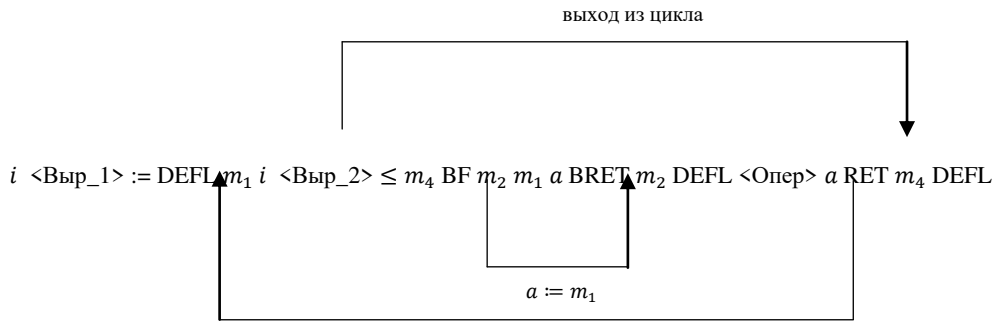


Рис. 5.7. Порядок выполнения ПОЛИЗ для оператора цикла `for`

Возможны и другие способы представления оператора цикла.

Замечание

Разработка представления конструкций языка программирования в ПОЛИЗ – это довольно сложная работа, при выполнении которой необходимо учитывать особенности языка программирования, выбранных методов анализа и целевой машины.

5.2. Тетрады

Для бинарных операций удобной формой представления программы после синтаксического анализа являются тетрады. Формат тетрад:

<код операции>, <операнд_1>, <операнд_2>, <результат>,

где <операнд_1> и <операнд_2> специфицируются аргументы, а <результат> – временное имя для хранения результата выполнения операции (переменная из рабочей области). В качестве операндов тетрад наряду с переменными и константами, определенными в исходной программе, могут выступать результаты ранее выполненных тетрад. Например, выражение: $a * b + c * d$ представляется в виде последовательности следующих тетрад:

$*, a, b, t_1$
$*, c, d, t_2$
$+, t_1, t_2, t_3.$

Замечание В тетрадах запрещены встроенные в операнды выражения.

Последовательность тетрад представляет собой программу, инструкции которой обрабатываются последовательно. Операнды одной тетрады должны быть одинакового типа. Для преобразования типа операнда можно использовать тетрады преобразования типа с кодами операций IA2 (целый — в вещественный) и A2I (вещественный — в целый). Поскольку операция преобразования типа одноместная, она записывается с пустым вторым операндом, например,

A2I, $a,\ \ , t$

Для представления унарного минуса в тетрадах также можно не использовать второй операнд. Тетрада

$-, a,\ \ , t$

интерпретируется как присвоение временной переменной t значения $-a$.

Множество очевидных (для представления выражений) операций может быть при необходимости расширено.

5.2.1 Переменные с индексами

Для представления переменных с индексами можно использовать тетраду

SUBS, $a,\ i,\ t,$

которая интерпретируется как операция доступа $a[i]$ и позволяет выбрать элемент одномерного массива.

Представление элементов многомерных массивов с помощью тетрад основана на приведении многомерного массива к одномерному (*см. разд. 1.7.5.2).* Так, если двухмерный массив a, имеющий n строк и m столбцов, отображается в памяти по строкам, нумерация индексов начинается с единицы и размер элемента массива – одно слово, то обращение к элементу массива $a[i, j]$ можно записать следующим образом:

$-, i, 1, t_1$
$*, t_1, i, t_2$
$+, t_2, j, t_3$
$-, t_3, 1, t_4$

SUBS, a, t_4, t_5

Первые четыре тетрады необходимы для вычисления значения смещения элемента массива $a[i, j]$ от начала массива по формуле: $(i-1)*m+j-1$.

5.2.2. Указатели функций

Обращение к функции записывается с помощью одной или нескольких тетрад в зависимости от числа аргументов функции. Так обращение к функции одно переменной $abs(x)$ записывается в виде одной тетрады

CALL, $abs, x, t,$

Возможна и другая реализация обращения к функции с использованием тетрад. Так обращение к функции двух переменных $f(x, y)$ можно записать в виде:

PARAM, x, ,

PARAM, y, ,

CALL , f, 2.

Здесь тетрады с кодом PARAM используются для определения параметров функции, а в тетраде CALL указывается имя функции и число её параметров.

5.2.3. Операторы

Формат тетрад для представления оператора присваивания и операторов безусловного и условного переходов приведен в табл. 10.4.

Таблица 5.4. *Формат тетрад для представления основных операторов*

Тетрада				**Семантика тетрады**
:=	b		a	$a := b$
DEFL	L			Определение метки l
BRL	I			Безусловный переход к тетраде с меткой I
BF	I	E		Переход к тетраде с меткой I, если значение выражения E равно “ложь”
BR	m			Безусловный переход на тетраду с номером m
BZL (BPL, BML)	m	E		Переход к тетраде с меткой m, если значение выражения E равно нулю (больше нуля, меньше нуля)
BZ (BP, BM)	m	E		Переход на тетраду с номером m, если значение выражения E равно нулю(больше нуля, меньше нуля)

Таблица 10.4. *(окончание)*

Тетрада				**Семантика тетрады**
BE (BP, BM)	m	e		Переход на тетраду с номером m, если значение выражения E равно нулю (больше нуля, меньше нуля)
BE (BL, BG)	m	e_1	e_2	Переход на тетраду с номером m, если значение выражения E_1 равно (меньше, больше) значения выражения E_2

В табл. 10.5. приведено представление оператора условного перехода и операторов цикла с помощью тетрад из табл. 10.4.

Замечание В табл. 10.5 функция signum служит для проверки знака выражения (первый операнд задает значение проверяемого выражения, а второй операнд возвращает знак выражения).

Тетрады удобно использовать для выполнения машинно-независимой оптимизации, в частности исключения лишних операций. Основным недостатком тетрад является большой объем памяти, необходимый для их хранения. Несмотря на то, что во многих тетрадах имеются свободные поля, результат выполнения операции всегда записывается в четвертое поле.
Внутреннее представление тетрады – запись, состоящая из четырех лексем. При этом коды операций тетрад, пустые поля, номера тетрад и временные переменные t_i являются лексемами специального типа.

Таблица 5.5. *Представление оператора условного перехода и операторов цикла с помощью тетрад из табл. 5.4.*

Оператор	**Формат оператора**			
	Код операции	**Операнд_1**	**Операнд_2**	**Результат**
if <выражение> **then** <операторы_1> **else** <операторы_2> **end**	<выражение> (результат в t)			
	BF	*Lelse*	*t*	
	<операторы_1>			
	BRL	*Lend*		
	DEFL	*Lelse*		
	<операторы_2>			
	DEFL	*Lend*		

Таблица 10.5. *(окончание)*

Оператор	**Формат оператора**			
	Код операции	**Операнд_1**	**Операнд_2**	**Результат**
for j := <выражение_1> **step** <выражение_2> **until** <выражение_3> **do** <оператор>	<выражение_1> (результат в t_1)			
	:=	t_1		*j*
	<выражение_2> (результат в t_2) <выражение_3> (результат в t_3)			
	DEFL – CALL * BPL	*Lbegin* *j* signum t_4 *Lend*	 t_3 t_2 t_5 t_6	 t_4 t_5 t_6
	<оператор>			

	+	j	t_2	j
	BRL	*Lbegin*		
	DEFL	*Lend*		
while <выражение> **do** <оператор>	DEFL	*Lbegin*		
	<выражение_1> (результат в t)			
	BF	t	*Lend*	
	<оператор>			
	BRL	*Lbegin*		
	DEFL	*Lend*		

5.3. Триады

Триады так же, как и тетрады, являются удобной промежуточной формой представления бинарных операций. Формат триады имеет вид:

<код операции>, <операнд_1>, <операнд_2>.

В качестве кодов операций триад можно использовать коды операций тетрад, за исключением операций условного перехода BE (BL, BM), которые для триад не используются.

В триадах отсутствует поле для записи результата. Если в качестве операнда некоторой триады нужно использовать результат выполнения другой (ранее выполненной) триады, то в соответствующее поле триады записывается ссылка на триаду, в которой этот результат был получен. Результаты выполнения триад хранятся отдельно. Например, запись в виде последовательности триады выражения

$$a * b - c \,/\, d$$

будет иметь следующий вид (номера триад указаны в скобках):

(1) $*,\ a,\ b$

(2) $/,\ c,\ d$

(3) $-, (1), (2)$

Количество триад и тетрад, необходимое для представления одних и тех же конструкций языка, является одним и тем же, но каждая триада занимает в памяти меньше места. Экономия памяти при использовании триад в качестве промежуточной формы программы по сравнению с тетрадами достигается также за счет того, что после выполнения триады, в которой одним из операндов является результат выполнения i-ой триады, ее результат может быть уничтожен.

При выполнении машинно-независимой оптимизации некоторые операции могут удаляться из промежуточного представления программы или переставляться на другое место. При использовании тетрад такие преобразования не вызывают затруднений. В случае же использования триад могут возникнуть осложнения из-за того, что триады ссылаются друг на друга. Поэтому, если машинно-независимая оптимизация будет выполняться, необходимо либо хранить все результаты выполнения триад, либо использовать *косвенные триады.*

Косвенные триады представляются двумя списками. В первом списке хранятся сами триады, причем одинаковые триады в список заносятся только один раз. Второй список содержит номера триад в порядке их выполнения. При использовании косвенных триад перестановка и исключение идентичных операций во время машинно-независимой оптимизации производятся путем преобразования списка, содержащего номера триад. При этом вид триад и их количество в списке не изменяются. Использование косвенных триад приводит к дополнительному сокращению объема памяти, необходимого для хранения промежуточной формы программы, если исходная программа содержит большое число идентичных операций (обычно это имеет место при наличии в программе большого числа индексированных переменных)ю

Пример представления последовательности операторов присваивания языка Паскаль с помощью косвенных триад приведены в табл. 5.6.

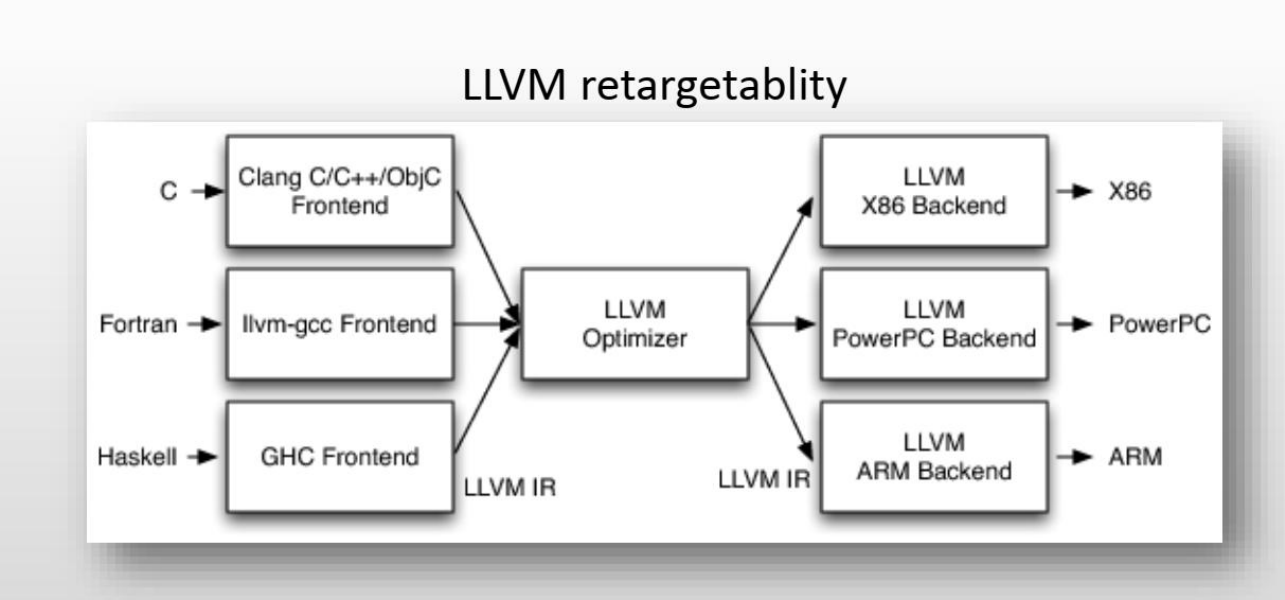
Таблица 5.6. *Представление последовательности операторов присваивания языка Паскаль с помощью косвенных триад*

Оператор	**Список триад**			**Порядок выполнения триад**
$A := B[j] + B[j+1];$	SUBS	B	j	(1), (2), (3), (4), (5), (1), (2), (3), (6), (7), (1), (2), (3), (8), (9), (10)
$C := B[j] - B[j+1];$	+	j	1	
$D := B[j] + 2 * B[j+1]$	SUBS	B	(2)	
	+	(1)	(3)	
	:=	(4)	A	
	•	(1)	(3)	
	:=	(6)	C	
	*	2	(3)	
	+	(1)	(8)	
	:=	(9)	D	

5.6. LLVM

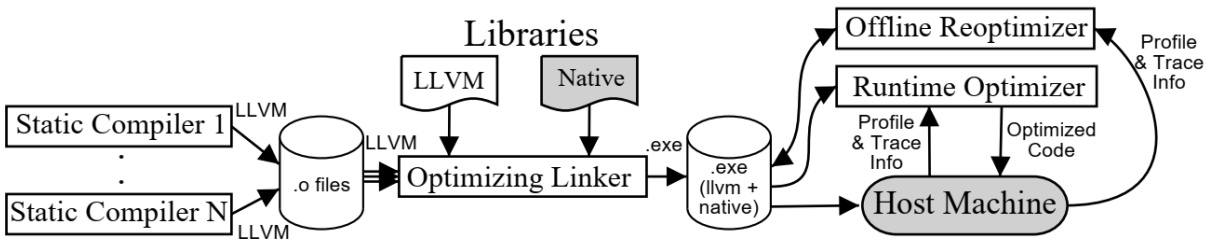
LLVM (ранее Low Level Virtual Machine[7]) — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня (так называемых

«фронтендов»), системы оптимизации, интерпретации и компиляции в машинный код. В основе инфраструктуры используется RISC-подобная платформонезависимая система кодирования машинных инструкций (байткод LLVM IR), которая представляет собой высокоуровневый ассемблер, с которым работают различные преобразования.



В основе LLVM лежит промежуточное представление кода (Intermediate Representation, IR), над которым можно производить трансформации во время компиляции, компоновки и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM 9.0.0 поддерживает статическую генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, RISC-V, Qualcomm Hexagon, NVPTX, SystemZ, Xcore. JIT-компиляция (генерация машинного кода во время исполнения) поддержана для архитектур x86, x86_64, PowerPC, MIPS, SystemZ, и частично ARM[12].

LLVM написана на C++ и портирована на большинство Unix-подобных систем и Windows. Система имеет модульную структуру, отдельные её модули могут быть встроены в различные программные комплексы, она может расширяться дополнительными алгоритмами трансформации и кодогенераторами для новых аппаратных платформ. LLVM включена обёртка API для OCaml.



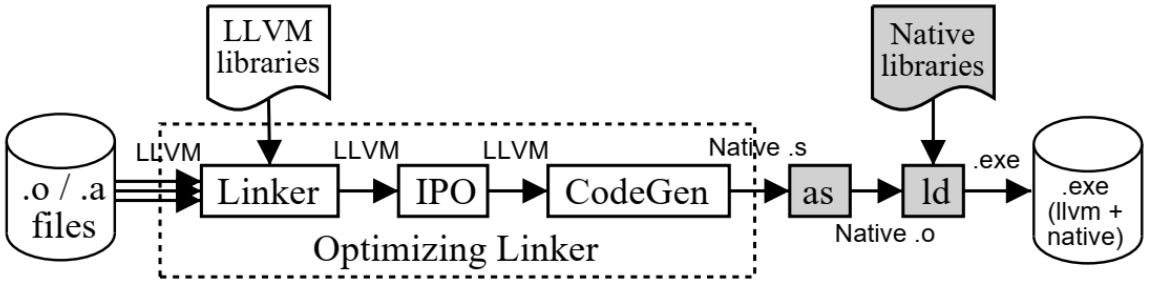


Figure 2: LLVM Optimizing Linker

How to build the LLVM

- http://llvm.org/docs/GettingStarted.html#getting-started
- Download llvm 3.2, clang., Compiler-RT from http://llvm.org/releases/download.html#3.2

```
$ tar zxf llvm-3.2.src.tar.gz
$ cd llvm-3.2.src/tool
$ tar zxf clang-3.2.src.tar.gz
$ mv clang-3.2.src.tar.gz clang
$ cd llvm-3.2.src/projects
$ tar zxf compiler-rt-3.2.src.tar.gz
$ mv compiler-rt-3.2.src compiler-rt
$ cd where-you-want-to-build-llvm
$ ../llvm/configure
$ make
```

5.5. Байт-коды JVM

Виртуальная машина Java базируется на понятии виртуального компьютера с логическим набором инструкций (псевдокодов), определяющих операции, которые может выполнить этот компьютер. *Интерпретатор* JVM – это программа, которая понимает семантику байт-кодов JVM и преобразует их в инструкции объектной машины. JVM базируется на концепции *стековой машины*.

Стековая машина не использует никакие физические регистры для передачи информации от одной инструкции к другой. Вместо этого используется стек, в

котором производятся вычисления. В JVM имеется единственный регистр, называемый *регистром PC,* который содержит *адрес* текущей выполняемой инструкции в потоке байт-кодов. Инструкции байт-кода выполняются последовательно. Для изменения порядка выполнения байт-кодов имеются специальные команды, изменяющие содержимое регистра PC.
Большинство семантических процедур, реализующих инструкции байт-кода, получают свои операнды из стека и помещают результаты обратно в стек. Инструкции могут иметь также аргументы, которые следуют в потоке байт-кодов непосредственно за самой инструкцией. В табл. 10.7 приводятся наиболее часто используемые инструкции байт-кода, упорядоченные по коду операции. Полный список всех инструкций JVM можно найти, например, в [8].

Пример 5.5
С использованием операций из табл. 10.7. байт-код фрагмента программы на Java

```
x = 0; a = 1; b = 2; c = 5;
x = a + b * c;
```

имеет следующий вид

03	Записать в стек 0
3C	Прочитать содержимое стека в `x`
04	Записать в стек 1
3D	Прочитать содержимое стека в `a`
10 02	Записать в стек 2
3E	Прочитать содержимое стека в `b`
10 05	Записать в стек 5
36 04	Прочитать содержимое стека в `c`
1C	Поместить в стек `a`
1B	Поместить в стек `b`
15 04	Поместить в стек `c`
68	Поместить в стек `bc*`
60	Поместить в стек `a+`
3C	Прочитать содержимое стека в `x`

Контрольные вопросы

1. Почему исходная программа переводится сначала в промежуточное представление, а затем в объектный код?
2. Как арифметическое выражение записывается в ПОЛИЗ? Как в ПОЛИЗ записывается унарный минус?
3. Почему в компиляторах используется для представления выражений польская инверсная запись?
4. Как производится вычисление выражения, представленного в польской инверсной записи?
5. Как графически представляется польская инверсная запись?

6. Как можно представить в тетрадах переменную с индексами, указатель функции?
7. Чем различаются триады и косвенные триады?
8. Что представляет собой виртуальная машина Java?
9. Как обрабатываются инструкции байт-кода?

Упражнения

1. Представьте графически и в польской инверсной записи выражения:
   1.1. $|cosx + \sin y - z|$.
   1.2. $\sqrt{b + |a|}$.
   1.3. $1 + a_{i,j} - ctg\frac{x}{\sqrt{x^2+1}}$.
2. Представьте в ПОЛИЗ, тетрадах и триадах операторы
   2.1 $a := b^2 + 4ac$.
   2.2 $if\ (b\ \&\&\ c < d)\ a++; else\ a+= 2$.
   2.3 $for\ j := n\ downto\ 1\ do\ a[j + 1]$.

Глава 6. Методы синтаксически управляемого перевода

Грамматики описывают только синтаксические аспекты формальных языков.

Синтаксически управляемая трансляция (англ. Syntax-directed translation, SDT) - метод реализации компилятора, при котором перевод исходного языка в выходной язык полностью выполняется анализатором.

6.1. Перевод и семантика

Определение. Отношение между входными цепочками языка и выходными цепочками языка называют трансляцией или переводом.

Существуют два подхода к описанию перевода:

1. использование системы, аналогичной формальной грамматике или являющейся её расширением, которая порождает перевод (точнее пару цепочек, принадлежавших переводу);

   Для перевода могут быть применены два метода:

   1. схема трансляции — описание грамматики с встроенными семантическими операциями $A \rightarrow B + C \{+\}$;
   2. семантические правила с атрибутами: грамматика может быть применена для выполнения семантических правил над атрибутами символов.

правило продукций	семантические правила
$A \rightarrow B + C$	$A.val \rightarrow B.val + C.val$

   Атрибуты могут включать в себя тип переменной, значение выражения, и т.п. Для символа A с атрибутом val обращение к атрибуту выглядит как A.val.

2. применение автомата, распознающего входную цепочку и выдающего на выходе перевод этой цепочки.

Перейдём к рассмотрению простейших способов описания перевода.

Определение. Пусть $\sum$ — входной алфавит и Δ — выходной алфавит. Переводом с языка $L_1 \subseteq \sum^*$ на язык $L_2 \subseteq \Delta^*$ называется отношение τ из $\sum^*$ в Δ^*, для которого L_1 — область определения, а L_2 — множество значений.

Если $(x, y) \in \tau$, то цепочка y называется выходом для цепочки x. В общем случае в переводе τ для одной входной цепочки может быть задано более одной выходной цепочки.

Для языков программирования перевод должен быть функцией, т. е. для каждой входной программы должно быть не более одной выходной программы.

Простейшие переводы можно задать при помощи гомоморфизма.

Пример 5.1. Пусть требуется перевести символы, образующие целые числа со знаком, в и названия. В этом случае гомоморфизм h можно определить следующим образом:

$\sum = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +\}$;

h(a) = a, если a ∈ ∑;

Для a ∈ ∑ гомоморфизм h(a) определяется в соответствии с табл. 5.1.

Таблица 5.1. Определение гомоморфизма h(a)

a	**h(a)**
1	один
2	два
3	три
4	четыре
5	пять
6	шесть
7	семь
8	восемь
9	девять
0	нуль
+	плюс
-	минус

Используя данное определение, перевод входной цепочки "-15", содержащей запись числа -15, будет иметь вид: "минус один пять".

Гомоморфизм позволяет описывать только простейшие переводы.

Нельзя задать при помощи гомоморфизма даже такие простые переводы, как τ_1 = {(x, y) | x – инфиксное выражение и y – префиксная запись выражения x} и τ_2 = {(x, y) | x – инфиксное выражение и y – постфиксная запись выражения x}. Нужны более мощные формализмы.

6.2 СУ-схемы

Схема синтаксически управляемого (СУ-схема) перевода представляет собой систему, порождающую пары цепочек, принадлежавших переводу. Неформально схема синтаксически управляемого перевода представляет собой грамматику, в которой каждому правилу приписан элемент перевода. При порождении цепочки каждый раз, когда правило участвует в этом процессе, элемент перевода генерирует часть выходной цепочки, соответствующей части входной цепочки, порождённой этим правилом.

Рассмотрим схему синтаксически управляемого перевода, описывающего перевод скобочного арифметического выражения, порождаемого грамматикой G_0, в соответствующую польскую инверсную запись G_1. Схема перевода представлена в табл. 5.2.

Таблица 5.2. СУ-схема перевода

N	Правила грамматики G_0	Элемент перевода G_1
1	E ⟶ E + T	E ⟶ {new Symbol} E {new FA}T +
2	E ⟶ T	E ⟶ T

3	T ⟶ T * P	T ⟶ TP *
4	T ⟶ P	T ⟶ P
5	P ⟶ (E)	P ⟶ E
6	P ⟶ i	P ⟶ i

Правилу грамматики E → E + T соответствует элемент перевода E ⟶ ET+. Используя СУ-схему, приведённую в табл. 5.2, определим перевод цепочки i * (i + i).

Определим левый вывод входной цепочки:

$E \Rightarrow^{2} T \Rightarrow^{3} T * P \Rightarrow^{4} P * P \Rightarrow^{6} i * P \Rightarrow^{5} i * (E) \Rightarrow^{1} i * (E + T) \Rightarrow^{2} i * (T + T) \Rightarrow^{4} i * (P + T) \Rightarrow^{6} i * (i + T) \Rightarrow^{4} i * (i + P) \Rightarrow^{6} i * (i + i)$

Полученный вывод 23465124646 определяет последовательность использования элементов перевода при порождении выходной цепочки, поэтому последовательность выводимых пар цепочек имеет следующий вид:

$(E, E) \Rightarrow^{2} (T, T) \Rightarrow^{3} (T * P, TP *) \Rightarrow^{4} (P * P, PP *) \Rightarrow^{6} (i * P, iP *) \Rightarrow^{5} (i * (E), i * E) \Rightarrow^{1} (i * (E + T), i * ET +) \Rightarrow^{2} (i * (T + T), i * TT +) \Rightarrow^{4} (i * (P + T), i * PT+) \Rightarrow^{6} (i * (i + T), i * iT +) \Rightarrow^{4} (i * (i + P), i * iP +) \Rightarrow^{6} (i * (i + i), i * ii +)$

Каждая выходная цепочка при порождении части входной цепочки получается путем замены (подходящего) не терминала выходной цепочки значением соответствующего ему элемента перевода.

Рассмотренная схема перевода относится к классу схем, называемых *схемами синтаксически управляемого перевода*.

Определение. Схемой синтаксически управляемого перевода называется пятёрка Tb = (V, ∑, Δ, P, S), где:

V — конечное множество нетерминальных символов;

∑ — конечный входной алфавит;

Δ — конечный выходной алфавит;

P — конечное множество правил вида A→ α, β, где α ∈ (V ∪ ∑)*, β ∈ (V ∪ Δ)* и вхождения не терминалов в цепочку β образуют перестановку вхождений не терминалов в цепочку α;

S — начальный символ (S ∈ V).

По определению, каждому вхождению определённого не терминала в цепочку α соответствует некоторое вхождение этого не терминала в цепочку β. Если некоторый не терминал входит в цепочку α более одного раза, то может возникнуть неоднозначность. Для исключения этого будем использовать верхний целочисленный индекс, например в правиле A → $B^{1}CB^{2}$, $B^{2}B^{1}C$ первой, второй и третьей позиции цепочки $B^{1}CB^{2}$ соответствуют вторая, третья и первая позиции цепочки $B^{2}B^{1}C$.

Примечание: выводимая пара цепочек Tb СУ-схемы определяется рекурсивно следующим образом:

1. (S, S) — выводимая пара, в которой символы S соответствуют друг другу.

2. Если ($\alpha A\beta$, $\alpha' A\beta'$) — выводимая пара, в которой два выделенных вхождения не терминала A соответствуют друг другу, и $A \to \alpha, \beta$ — правило из P, то ($\alpha\gamma\beta$, $\alpha'\gamma'\beta'$) — выводимая пара.

Вхождения не терминалов в цепочки γ и γ' соответствуют друг другу точно так же, как они соответствовали в правиле СУ-схемы. Вхождения не терминалов в цепочки α и β соответствуют вхождениям не терминалов в цепочки α' и β' в новой выводимой паре точно так же, как они соответствовали в старой выводимой паре. При необходимости это соответствие будет указываться верхними индексами.

Если между парами цепочек ($\alpha A\beta$, $\alpha' A\beta'$) и ($\alpha\gamma\beta$, $\alpha'\gamma'\beta'$) установлена описанная ранее связь, то будем писать ($\alpha A\beta$, $\alpha' A\beta'$) $\Rightarrow_{Tb}$ ($\alpha\gamma\beta$, $\alpha'\gamma'$ β').

Транзитивное замыкание, рефлексивно-транзитивное замыкание и k-ю степень отношения $\Rightarrow_{Tb}$ будем обозначать как $\Rightarrow^{+}_{Tb}$, $\Rightarrow^{*}_{Tb}$ и $\Rightarrow^{k}_{Tb}$, соответственно. В очевидных случаях можно опускать индекс Tb.

Определение. Переводом, определяемым схемой Tb, или τ(Tb) называют множество пар цепочек $\{(x, y) \mid (S, S) \Rightarrow^{*} (x, y), x \in \sum^{*}$ и $y \in \Delta^{*}\}$.

Пример 6.2 Пусть СУ-схема задана следующим образом:

Tb = ({S}, {i, +}, {i, +}, P, S), где множество правил P содержит правила

1. $S \to + S^1S^2, S^1 + S^2$
2. $S \to i, i$

Рассмотрим вывод в данной СУ-схеме:

$(S,S) \Rightarrow^1 (+ S^1S^2, S^1 + S^2)$

$\Rightarrow^1 (+ +S^3S^4S^2, S^3 + S^4 + S^2)$

$\Rightarrow^1 +++ S^5S^6S^4S^2, S^5 +S^6 + S^4 + S^2)$

$\Rightarrow^2 +++ iS^6S^4S^2, i +S^6 + S^4 + S^2)$

$\Rightarrow^2 +++ iiS^4S^2, i +i + S^4 + S^2)$

$\Rightarrow^2 +++ iiiS^2, i +i + i + S^2)$

$\Rightarrow^2 +++ iiii, i +i + i + i)$

Входной цепочке + + + iiii соответствует выходная цепочка i + i + i + i.

Перевод, задается СУ-схемой : $\tau(Tb) = \{(+)^k (i)^{k+1}, i(+ i)^k \mid k \geq 0\}$

Примечание: пусть Tb = (V, $\sum$, Δ, P, S) - СУ-схема, то τ(Tb) называется синтаксически управляемым переводом.

G_i = (V, $\sum$, P^i, S), где $P^i = \{A \to \alpha \mid A \to \alpha, \beta, \in P^i\}$, называется входной грамматикой СУ-схемы Tb.

G_j = (V, Δ, P^j, S), где $P' = \{A \to \beta \mid A \to \alpha, \beta, \in P^j\}$, называется выходной грамматикой СУ-схемы Tb.

Два способа синтаксически управляемого перевода: применение оператора **вывода** и метод преобразования **деревьев разбора**.

Метод преобразования деревьев вывода.

Дерево вывода входной грамматики G_i преобразуется в дерево вывода выходной грамматики G_j. Перевод входной цепочки α можно получить, построив ее дерево вывода, затем преобразовав это дерево в дерево вывода выходной грамматике, взяв крону выходного дерева как перевод цепочки α.

Алгоритм 6.1 Преобразование деревьев при помощи СУ-схемы

Вход: СУ-схема Tb = (V, ∑, Δ, P^i, S), входная грамматика G_i= (V, ∑, P^i, S), выходная грамматика G_j = (V, Δ, P^j, S) и дерево вывода D в G_i, с кроной, принадлежавшей $\sum^*$.

Выход: Дерево вывода D' в G_0, такое, что если x и y - кроны деревьев D и D' соответственно, то (x, y) ∈ τ(Tb).

Шаг 1. Применяется к внутренней вершине n дерева D, имеющей k прямых потомков $n_1,...,n_k$.

1.1 Устранить из множества вершин $n_1,...,n_k$ листья, помеченные терминальными символами или ε.

1.2 Пусть A→α – правило входной грамматики G_i, соответствующее вершине n и ее прямым потомкам, т. е. A – метка вершины n, и α образуется конкатенацией меток вершин $n_1,..., n_k$. Выбрать из P^i некоторое правило вида A → α, β (если таких правил несколько, то выбор произволен).

Переставить оставшиеся прямые потомки вершины n (если они есть) согласно соответствию между вхождениями не терминалов в α и β. (Поддеревья, корнями которых служат эти потомки, переставляются вместе с ними).

1.3 Добавить в качестве прямых потомков вершины и листья с метками так, чтобы метки всех ее прямых потомков образовали цепочку β.

1.4 Применить шаг 1 к прямым потомкам вершины n, не являющимися листьями, в порядке слева направо.

2. Повторять шаг 1 рекурсивно, начиная с корня дерева D.

Результат дерево D'.

Пример 6. 3. Дана СУ-схема Tb = ({S, A}, {0, 1}, {a, b}, P, S), где P состоит из правил:

1. S → 0AS, SAa 3. S → 1, b

2. A → 0SA, ASa 4. A → 1, b

Для входной грамматике вывод цепочки 1232343 имеет вид 0010111 (дерево вывода приведено на рис. 5.1).

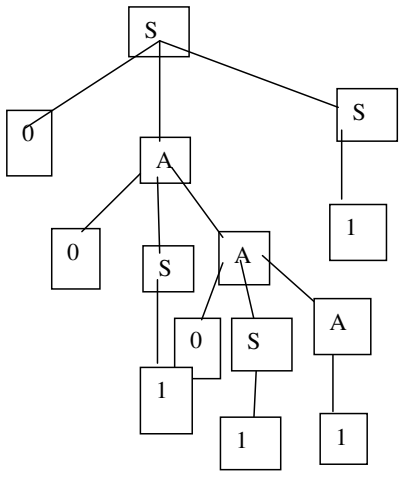


Рис. 5.1. Дерево вывода цепочки 0010111

Преобразуем это дерево алгоритмом 5.1.

Шаг 1. Применим шаг 1 алгоритма к корню дерева S:

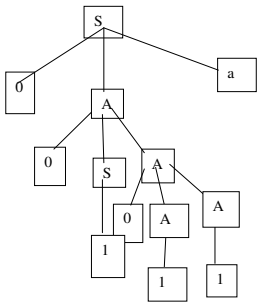
Устраняем левый лист дерева, помеченный 0. Находим правило входной грамматики 1. S → 0AS, соответствующее корню дерева S. У этого правила только один элемент перевода SAa, который определяет, что нужно поменять местами прямые потомки корня A и S. Добавляем прямой потомок к корню дерева и помечаем его a. Результат этих действий приведена на рис. 5.2, а.

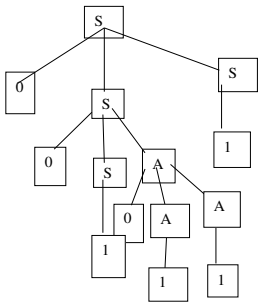
Шаг 2. Рассматриваем прямой потомок корня дерева – вершину, помеченную символом A.

Устраняем лист 0 по правилу (2) СУ-схемы, меняем местами прямые потомки рассматриваемой вершины. Добавляем прямого потомка к рассматриваемой вершине и получаем дерево, приведённое на рис. 5.2, б.

Шаг 3. Рассматриваем прямой потомок корня дерева – вершину, помеченную символом S.

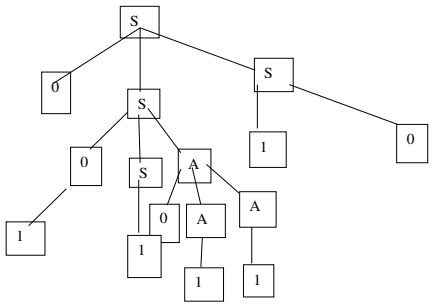
Устраняем лист 1. Находим соответствующее правило СУ-схемы – правило номер (3), и так как потомков у рассматриваемой вершины нет, то создаём прямой потомок b Для рассматриваемой вершины и получаем Дерево, приведённое на рис. 5.2, в

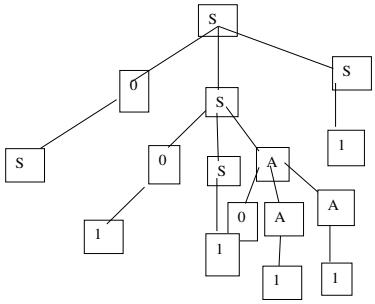




а

б





в **г**

Рис. 6.2. Преобразование деревьев

Продолжая рекурсивно выполнять шаг 1 алгоритма, получим результирующее дерево, приведённое на рис. 5.2, г.

1. Если x и y – кроны деревьев D_i и D_j из алгоритма 5.1, то $(x, y) \in \tau(Tb)$.
2. Если $(x, y) \in \tau(Tb)$, то существуют дерево D_i с кроной x и такая последовательность выборов вершин при обращении к шагу 2.2 алгоритма, что в результате получается дерево D_j с кроной y.

Замечание: порядок применений шага 2 алгоритма 5.1 к вершинам дерева не важен. Можно выбрать любой порядок, при котором каждая внутренняя вершина рассматривается точно один раз.

Определение. СУ-схема $Tb = (V, \sum, \Delta, P, S)$ называется простой, если для каждого правила $A \rightarrow \alpha, \beta \in P$ соответствующие друг другу вхождения не терминалов встречаются в α и β в одном и том же порядке. Перевод, определяемый *простой* СУ-схемой, называется *простым* синтаксически управляемым переводом.

Соответствие нетерминалов в выводимой паре простой СУ-схемы определяется порядком, в котором эти нетерминалы появляются в цепочках, поэтому для нее легко построить транслятор, представляющий собой преобразователь с магазинной памятью.

Пример 6.4. Рассмотрим простую СУ-схему, имеющую следующие правила:

1. $E \rightarrow (E), E$
2. $E \rightarrow E + E , E + E$

3. E → T, T
4. T → (T), T
5. T → A * A, A * A
6. T → i, i
7. A → (E + E), (E + E)
8. A → T, T

и вывод входной цепочки ((i * (i * i) + i) * i), равный 3457358684586863686.

Соответствующий вывод пар цепочек имеет вид:

(E, E) $\Rightarrow_3$ (T, T) $\Rightarrow_4$ ((T), T) $\Rightarrow_5$ ((A * A), A * A) $\Rightarrow_7$ (((E + E) * A,(E + E) * A)$\Rightarrow_3$ (((T + E) * A, (T + E) * A)$\Rightarrow_5$(((A * A + E) * A, (A * A + E) * A)$\Rightarrow_8$ (((T * A + E) * A, (T * A + E) * A)$\Rightarrow_6$ (((i * A + E) * A, (i * A + E) * A) $\Rightarrow_8$ (((i * T + E) * A, (i * T + E) * A) $\Rightarrow_4$ (((i * (T) + E) * A, (i * T + E) * A)$\Rightarrow_5$(((i * (A * A) + E) * A, (i * A * A + E) * A) $\Rightarrow_8$ (((i * (T * A) + E) * A, (i * T * A + E) * A) $\Rightarrow_6$ (((i * (i * A) + E) * A, i * i * A + E) * A) $\Rightarrow_8$ (((i * (i * T) + E) * A , (i * i * T + E) * A) $\Rightarrow_6$ (((i * (i * i) + E) * A ,(i * i * i + E) * A) $\Rightarrow_3$ (((i * (i * i) + T) * A, (i * i * i + T) * A) $\Rightarrow_6$ (((i * (i * i) + i) * A, (i * i * i + i) * A) $\Rightarrow_8$ (((i * (i * i) + i) * T, (i * i * i + i) * T) $\Rightarrow_6$ (((i * (i * i) + i) * i, (i * i * i + i) * i).

Для входной цепочки ((i * (i * i) + i) * i) СУ-схема порождает выходную цепочку (i * i * i + i) * i. анализ позволяет сделать вывод, что СУ-схема отображает арифметические выражения, порождаемые грамматикой G_j, в арифметические выражения, не содержащие избыточных скобок.

6.3. Транслирующие грамматики с операционными символами

Построим процессор, выполняющий перевод инфиксных арифметических выражений в польскую инверсную запись.

Пусть на входной ленте процессора находится цепочка a + b * c, и процессор должен работать следующим образом:

- прочитать входной символ, a;
- выдать символ, a на выходную ленту;
- почитать входной символ '+';
- почитать входной символ b;
- выдать символ b на выходную ленту;
- почитать входной символ '*';
- почитать входной символ c;
- выдать символ c на выходную ленту;
- выдать символ '*' на выходную ленту;
- выдать символ '+' на выходную ленту.

Эта последовательность операций определяется порядком следования операндов в инфиксной записи совпадающим с ПОЛИЗ, но операции в ПОЛИЗ должны следовать сразу за своими операндами. Обозначим операции чтения символа из входной ленты - читаемыми символами, а операции записи – символами, помещаемыми на выходную ленту,

заключёнными в фигурные скобки (операционными символами), то процесс перевода записывается следующим образом:

a {a} + b {b} * c {c} {*} {+}.

Определение. Транслирующей грамматикой называется пятерка объектов $G^T = (V, \sum_i, \sum_a, P, S)$, где $\sum_i$ - словарь входных символов, $\sum_a$ - словарь операционных символов, V - нетерминальный словарь, $S \in V$ - начальный символ транслирующей грамматики, P - конечное множество правил продукций вида $A \rightarrow \alpha$, в которых $A \in V$, а $\alpha \in (\sum_i \cup \sum_a \cup V)^*$. Транслирующая грамматика - это КС-грамматика, множество терминалов которой разбито на два множества: множество входных и множество операционных символов.

Алгоритм трансформации КС-грамматики G_0 в транслирующую грамматику.

Вход : КС-грамматиа G_0

Выход : G_0^T - транслирующая грамматика

1. Входной язык L описывается входной КС-грамматикой G_0.

2. В правила вывода G_{0r} грамматики вставляются операционные символы для описания семантических действий, связанных с соответствующими правилами.

В восходящих методах обработки языков широко применяются постфиксные транслирующие грамматики.

Определение. Постфиксной транслирующей грамматикой называется транслирующая грамматика у которой все операционные символы в правых частях правил вывода расположены правее всех входных и нетерминальных символов.

Транслирующая грамматика G_0^T, описывающая перевод инфиксных арифметических выражений в ПОЛИЗ, содержит следующие правила:

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

Грамматика G_0^T является постфиксной транслирующей грамматикой.

Определение. G_0 называется входной грамматикой для транслирующей грамматики G_0^T, если она получена из транслирующей грамматики путем вычеркивания всех операционных символов. $L(G_0)$ называется входным языком.

Например, входной грамматикой G_0 для транслирующей грамматики G_0^T является для инфиксных арифметических выражений, содержащая правила:

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

Рассмотрим левый вывод цепочки i + i * i во входной грамматике G_0^T:

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow P + T \Rightarrow i + T \Rightarrow i + T * P \Rightarrow i + P * P \Rightarrow i + i * P \Rightarrow i + i * i.$

Эта же последовательность правил для транслирующей грамматики G_0^T дает следующий левый вывод:

E ⇒ E + T {+} ⇒T + T {+} ⇒ P + T {+} ⇒ i {i} + T {+} ⇒ i {i} + T * P {*} {+} ⇒ i {i} + P * P {*} {+} ⇒ i {i} + i {i} * P {*} {+} ⇒ i {i} + i {i} * i {i} {*} {+}

Цепочки языка, порождаемого транслирующей грамматикой, называют активными цепочками. Входной частью активной цепочки называется последовательностью входных символов, полученная из активной цепочки после удаления всех операционных символов. Операционной частью активной цепочки называется последовательность операционных символов, полученная путем вычёркивания из активной цепочки всех входных символов.

Например, последовательность i + i * i является входной частью активной цепочки i {i} + i {i} * i {i} {*} {+}, а последовательность {i}{i}{i}{*}{+}- ее операционной частью.

Определение. Синтаксически управляемым переводом $\tau(G^T)$ урррprправляемым грамматикой G^T называется множество всех пар, первыми элементами которых являются входные части активной цепочки, а вторыми элементами — операционные части активной цепочки.

Применим алгоритм к входной грамматике G_0 для получения транслирующей грамматики $G_0{}^T$.

В ПОЛИЗ операнды располагаются в той же последовательности, что и в инфиксной записи, а знаки операций следуют непосредственно после своих операндов в том порядке, в котором они выполняются.

(в виде таблицы)

Для того чтобы идентификатор i выдавался сразу после его прочтения, правило (5) грамматики G_0 преобразуется к виду P → i {i}.

Чтобы знак операции сложения выдавался непосредственно после своих операндов, правило (1) заменяется на E → E + T {+}. Это правило интерпретируется следующим образом: обработка арифметического выражения E состоит из обработки первого операнда E, чтения символа '+', обработки второго операнда T и выдачи символа '+'. Аналогичные рассуждения позволяют следующим образом преобразовать правило (3) грамматики: T → T * P {*}.

Утверждение. Один и тот же перевод можно определить разными транслирующими грамматиками

Пример 6.5. Пусть две транслирующие грамматики $G_1{}^T$ и, $G_2{}^T$ определяющие перевод в ПОЛИЗ инфиксных бесскобочных арифметических выражений, выполняемых в порядке написания операций, выглядят следующим образом:

$G_1{}^T$

1. E → E + T {+}
2. E → E * T {*}
3. E → T
4. T → i {i}

G_2^T

1.E → i {i} R

2.R → + i {i} {+} R

3.R → * i {i} {*} R

4.R → ε

Выполним перевод цепочки i + i * i с использованием транслирующих грамматик G_1^T и G_2^T.

Активная цепочка для G_1^T порождается следующим образом:

E⇒² E * T {*}⇒¹ E + T {+} * T {*}⇒³ T + T {+} * T {*}⇒⁴ i {i} + T {+} * T {*}⇒⁴ i {i} + i {i}{+} * T {*} ⇒⁴ i {i} + i {i}{+} * i {i}{*}

Операционная часть цепочки {i}{i}{+}{i}{*} является переводом входной цепочки i + i * i.

Активная цепочка для G_2^T порождается следующим образом:

E⇒¹ i{i}R ⇒² i{i} + i {i}{+} R ⇒³ i {i} + i {i}{+} * i {i}{*} R ⇒⁴ i {i} + i {i}{+} * i {i}{*}

Операционная часть цепочки {i}{i}{+}{i}{*} также является переводом входной цепочки i + i * i. Рис. 5.3 показывает деревья вывода цепочки i{i}+i {i}{+} * i {i}{*} грамматик G_1^T и G_2^T соответственно.

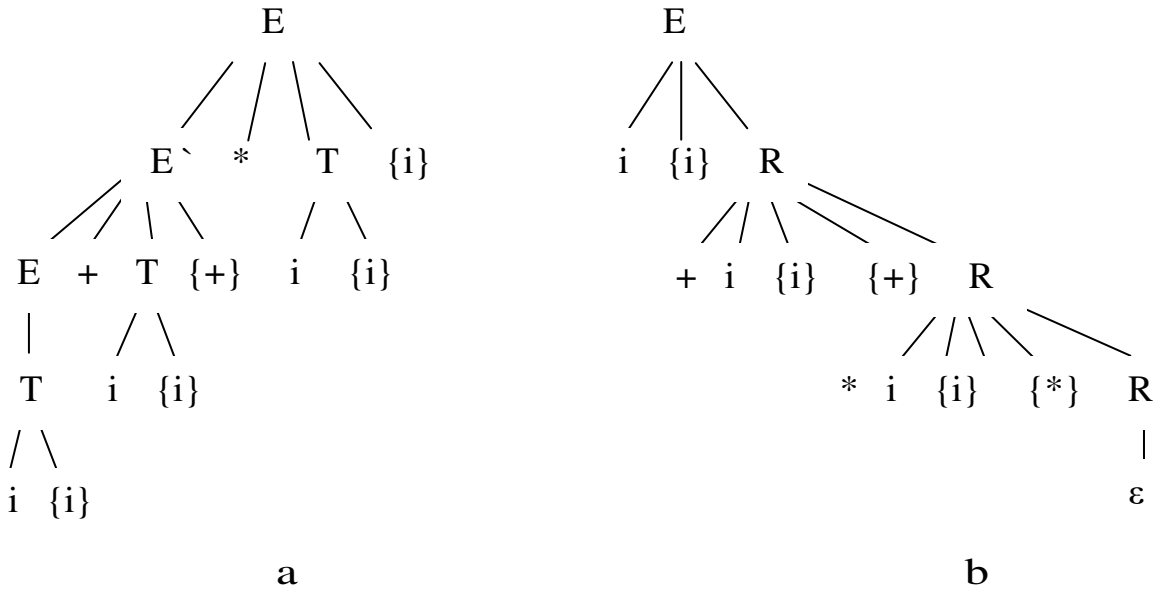


Рис. 6.3. Деревья вывода цепочки i{i}+ i {i}{+} * i {i}{*} по грамматикам G_1^T (a) и G_2^T(b)

Определение. Входную часть активной цепочки называют **входной цепочкой**, операционную часть – **выходной цепочкой**, а транслирующую грамматику G_0^T - грамматикой цепочечного перевода если при реализации появление входного символа в активной цепочке интерпретируется как операция чтения этого символа считывающим устройством, а вхождению операционного символа в правило вывода сопоставляется операция выдачи символа, заключённого в фигурные скобки.

Грамматика G_0^T является примером грамматики цепочечного перевода:

1. E → E + T {+}

2. E → T

4. T → P
3. T → T * P {*}
5. P → i {i}
6. P → (E)

Операционные символы интерпретируются именами семантических процедур, вызываемых при обработке входной цепочки. Активная цепочка задает последовательность наступления событий вызовов этих процедур по отношению к событиям чтения входных символов.

6.4 Атрибутные транслирующие грамматики

При определении транслирующей грамматики в понятие **входного символа** не включалось представление о том, что он является **лексемой**. Лексема состоит из двух компонентов: **типа и значения**.

Транслирующая грамматика описывает перевод только той части символа, который задает его тип. Расширением понятия грамматики цепочечного перевода, использующей при переводе оба компонента входного символа является **атриибитно транслирующая грамматики (АТ-грамматики)**. АТ-грамматики были предложены Дональдом Кнутом.

В АТ-грамматике символы снабжаются **атрибутами**, которые могут принимать значения из некоторого множества допустимых значений и **интерпретируются** как **семантическая** информация, связанная с конкретным вхождением символа в правило вывода грамматики.

При таком подходе значения атрибутов связываются с вершинами дерева вывода в АТ-грамматике, а правила вычисления значений атрибутов сопоставляются правилам вывода грамматики.

Атрибуты нетерминальных и операционных символов делятся на синтезированные и унаследованные атрибуты.

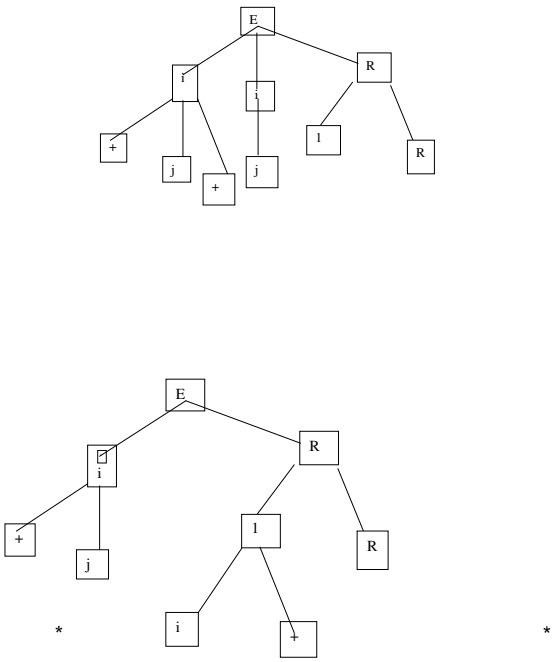
6.4.1 Синтезированные атрибуты

Пусть задана транслирующая грамматика G^T_3, допускающая в качестве входа арифметические выражения, построенные из символов множества {c, +, *, (,)}, где c - лексема, является целочисленной константой и порождающая на выходе значение этого выражения:

0. S → E {РЕЗУЛЬТАТ}

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

На рис. 5.4, а изображено дерево вывода для входной цепочки $c_5 + c_2 * c_3$ (индексы обозначают значения лексем). Входной цепочке должна соответствовать выходная цепочка {РЕЗУЛЬТАТ}.



а **б**

Рис. 5.4. Дерево вывода цепочки $c_5 + c_2 * c_3$
во входной (а) и атрибутной транслирующей (б) грамматиках.

Вершины не терминалов E, T и P в дереве вывода представляет собой некоторое подвыражение входного выражения.

КР 1 Алгоритм преобразования исходной транслирующей грамматики в АТ-грамматику. (на примере есть КР)

Введём для каждого из символов E, T и P по одному атрибуту, значением которого будет значение подвыражения, порождаемого этим не терминалом.

Входной символ "c" и операционный символ {РЕЗУЛЬТАТ} также имеют по одному атрибуту. Значение атрибута символа "c" равно значению константы, которую он представляет, значением операционного символа {РЕЗУЛЬТАТ} является результат вычисления выражения.

Атрибуты символов могут принимать любые целочисленные значения из области допустимых для выражений, описываемых соответствующей входной грамматикой G_3. На рис. 5.4, б изображено дерево вывода той же цепочки, что и на рис. 5.4, а, но на этом дереве не терминалы и операционный символ {РЕЗУЛЬТАТ} помечены значениями своих атрибутов.

Выберем для каждого атрибута каждого вхождения символа в правило вывода грамматики уникальное имя и включим атрибуты в правила вывода в виде подстрочных индексов соответствующих символов.

Замечание: одни и те же имена атрибутов можно использовать в нескольких правилах, т. к. они локальны в правиле грамматики.

Сопоставим каждому правилу вывода грамматики правило вычисления значения атрибута не терминала из левой части правила, которому соответствует нетерминальная вершина дерева вывода.

Рассмотрим, например, вершину T дерева, изображённого на рис. 5.4, а. Правило вывода, применённое к этой вершине, имеет вид: T → T * P. Оно определяет, что значение подвыражения, порождённого не терминалом из левой части правила, равно значению подвыражений, порождённых символами P и T из правой части правила, которые являются прямыми потомками вершины T.

Если p - атрибут символа T из левой части правила, а q и r - атрибуты символов T и P из правой части правила, то правило вычисления значения атрибута p будет иметь вид: p ⟵ q * r, где символ '⟵' - знак операции присваивания.

Таким образом, начиная с атрибутов входных символов и поднимаясь по дереву от кроны к корню, можно определить все значения атрибутов нетерминалов из левых частей правил вывода.

Правила вычисления значений атрибутов для AT-грамматики G_3^T, полученной по транслирующей грамматике G_3^T, будут выглядеть следующим образом:

0. $S \rightarrow E_q \{РЕЗУЛЬТАТ\}_p$
   $q \longleftarrow p$

1. $E_p \rightarrow E_q + T_r$
   $p \longleftarrow q + r$

2. $E_p \rightarrow T_q$
   $p \longleftarrow q$

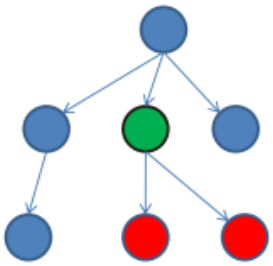
3. $T_p \rightarrow T_q * P_r$
   $p \longleftarrow q * r$

4. $T_p \rightarrow P_q$
   $p \longleftarrow q$

5. $P_p \rightarrow c_q$
   $p \longleftarrow q$

6. $P_p \rightarrow (E_q)$
   $p \longleftarrow q$

Определение. **Синтезированными атрибутами называют атрибуты значения которых вычисляются при движении по дереву вывода снизу вверх.** Термин "синтезированный" подчёркивает, что значение атрибута синтезируется из значений атрибутов потомков см. рис. Значение атрибута зеленой вершины вычисляется на основе красных вершин.



SDD involving only synthesized attributes is called ***S-attributed***

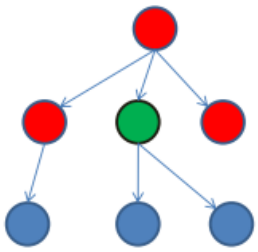
Synthesized Attributes

Attribute of the node is defined in terms of:
•Attribute values at children of the node
•Attribute value at node itself

Атрибуты операционных символов, например, {РЕЗУЛЬТАТ}, относятся к классу унаследованных атрибутов.

6.4.2. Унаследованные атрибуты

Определение. **Унаследованными атрибутами называют атрибуты, значения которых вычисляются из значений атрибутов вершины родителя или атрибутов его соседей в дереве вывода** см. рис. Значение атрибута зеленой вершины вычисляется на основе красных вершин.



Inherited Attributes

Attribute of the node is defined in terms of:
•Attribute values at parent of the node
•Attribute values at siblings
•Attribute value at node itself

Например, в грамматике G_3^T, значение атрибута p символа $\{РЕЗУЛЬТАТ\}_p$ равно значению атрибута не терминала E (соседа слева), порождающего все выражение. Оно может быть вычислено только после того, как будут определены значения всех синтезированных атрибутов не терминалов.

Рассмотрим пример АТ-Грамматики.

Кр (другой пример Python ?созадание списка переменных) Пример 6.7 Пусть входная КС-грамматика G, порождающая описания переменных в некотором языке программирования, выглядит следующим образом:

1. D → t i L 2. L → , i L 3. L → ε

Множество входных символов этой грамматики состоит из лексем: запятая, идентификатор i и имя типа t.

Семантика обработки описания заключается в занесении типа переменной в определённое поле элемента таблицы идентификаторов с помощью семантической операции установить ТИП с двумя параметрами: указатель на элемент таблицы индефикаторов, соответствующей описываемой переменной, и тип переменной.

Операцию установить ТИП лучше всего вызывать сразу после распознавания идентификатора. Указанная последовательность действий может быть описана транслирующей грамматикой G^T:

$D \rightarrow t\ i\ \{ТИП\}\ L$

$L \rightarrow\ ,\ i\ \{ТИП\}\ L$

$L \rightarrow \varepsilon$

Введём в транслирующую грамматику G^T атрибуты и правила их вычисления. Входные символы t и i имеют по одному атрибуту. Значением атрибута символа i является указатель на элемент таблицы идентификаторов, а атрибут символа t может принимать значения из множества {ЦЕЛЫЙ, ВЕЩЕСТВЕННЫЙ, ЛОГИЧЕСКИЙ}.

Операционный символ {ТИП} должен иметь два унаследованных атрибута, значения которых совпадают со значениями соответствующих фактических параметров процедуры установить ТИП. Значения унаследованных атрибутов символа {ТИП} для первого правила вывода приравниваются значениям соответствующих атрибутов входных символов i и t, входящих в правую часть того же правила левее символа {ТИП}.

Во второе правило тип описываемых переменных можно передать через унаследованный атрибут не терминала L. Значение этого унаследованного атрибута будет передаваться по дереву сверху вниз, начиная с вершины, где он получает начальное значение, равное значению атрибута входного символа t.

Атрибутная транслирующая грамматика G^{AT} выглядит следующим образом:

1. $D \rightarrow t_r\ i_a\ \{ТИП\}_{a1,\ r1}\ L_{r2}$

   $a1 \leftarrow a$

   $r1, r2 \leftarrow r$

2. $L_r \rightarrow\ ,\ i_a\ \{ТИП\}_{a1,\ r1}\ L_{r2}$

   $a1 \leftarrow a$

   $r1, r2 \leftarrow r$

3. $L_r \rightarrow \varepsilon$

Замечание: запись атрибутного правила в виде $r1,\ r2 \leftarrow r$ означает, что значение r присваивается одновременно атрибутам r1 и r2.

Входной цепочке $t_{целый}\ i_5,\ i_9,\ i_2$ соответствует дерево вывода в грамматике G^{AT}, изображённое на рис. 5.5. (414)

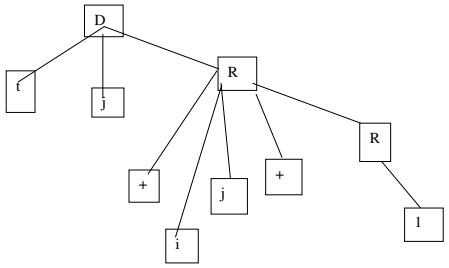


Рис. 6.5. Дерево вывода

В рассмотренных примерах все операционные символы имели унаследованные атрибуты. При определении атрибутной транслирующей грамматики можно обойтись без синтезированных атрибутов операционных символов. Необходимость в синтезированных атрибутах операционных символов может возникнуть в некоторых практических реализациях переводов, поэтому в определение АТ-Грамматики включены оба типа атрибутов.

6.4.3. Определение атрибутной транслирующей грамматики

Атрибутная транслирующая грамматика - это транслирующая грамматика, обладающая следующими дополнительными свойствами:

1. Каждый символ грамматики (входной, нетерминальный и операционной) имеет конечное множество атрибутов, и каждый атрибут имеет (возможно, бесконечное) множество допустимых значений.

2. Все атрибуты нетерминальных и операционных символов делятся на синтезированные и унаследованные.

3.Унаследованные атрибуты подчиняются следующим правилам:

- каждому вхождению унаследованного атрибута в правую часть правела вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;

- для каждого унаследованного атрибута начального символа грамматики задаётся начальное значение.

4.Правила вычисления значений синтезированных атрибутов определяются следующим образом:

- каждому вхождению синтезированного атрибута **нетерминального** символа в левую часть правила вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;

• каждому синтезированному атрибуту **операционного** символа сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов данного символа.

• каждому синтезированному атрибуту **операционного** символа сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов данного символа.

Атрибуты записываются в виде индексов соответствующих символов АТ-грамматики, при этом для каждого атрибута указывается его тип. Например:

$X_{a,b}$ синтезированный a, унаследованный b

Правила вычисления атрибутов записываются в виде операторов присваивания, левая часть которых - атрибут или список атрибутов, а правая часть - функция.

Рассмотрим АТ-грамматику G_A, описывающую перевод оператора присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ, УМНОЖИТЬ, ПРИСВОИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью - бесскобочное арифметическое выражение, выполняемое слева направо в порядке написания операций.

Входными символами грамматики являются символы множества {i, +, *, =}, где i - лексема, представляющая идентификатор. Каждый идентификатор имеет один атрибут, значением которого является указатель на соответствующий элемент таблицы идентификаторов.

Операционные символы {+}, {*}, {=} соответствуют кодам операций тетрад, получаемых на выходе. Символы {+} и {*} имеют по три унаследованных атрибута, значениями которых являются указатели на элементы таблицы, в которой хранятся значения левого операнда, правого операнда и результата соответственно. Операционный символ {=} имеет два унаследованных атрибута. Значение первого атрибута - указатель на элемент таблицы идентификаторов, соответствующий идентификатору из левой части оператора присваивания, а значением второго атрибута является указатель на элемент таблицы, в котором хранится значение выражения из правой части оператора присваивания.

Словарь нетерминальных символов состоит из символов S, E и R. Начальный символ S не имеет атрибутов, символ E имеет один синтезированный атрибут, значением которого является указатель на элемент таблицы, содержащий значение выражения, порождаемого E, а символ R имеет два атрибута: унаследованный (первый) и синтезированный. Значением унаследованного атрибута является указатель на элемент таблицы, соответствующий промежуточному результату, предшествующему R. Этот промежуточный результат является также левым операндом первой операции, порождаемой нетерминальным символом R, если такая операция имеет место. Значение синтезированного атрибута символа R - это указатель на элемент таблицы, соответствующий значению подвыражения, которое получается после

присоединения цепочки, порождённой символом R, к цепочке, представляющей левый операнд.

Правила вывода АТ-Грамматики G_A с соответствующими правилами вычисления следующих атрибутов:

E_t синтезированный t

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

Атрибуты операционных символов унаследованные.

1) $S \rightarrow i_{p_1} = E_{q_1}\ \{=\} p_2, q_2$

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

2) $E_{t_2} \rightarrow i_{p_1}\ R_{p_2},\ t_1$

$p_2 \leftarrow p_1$

$t_2 \leftarrow t_1$

3) $R_{p_1,\ t_2} \rightarrow i_{q_1}\ \{+\} p_2,\ q_2,\ r_1\ R_{p_2,\ t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

4) $R_{p_1,\ t_2} \rightarrow^* i_{q_1}\ \{*\} p_2,\ q_2, r_1\ R_{p_2,\ t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

5) $R_{p_1,\ t_2} \rightarrow \varepsilon$

$p_2 \leftarrow p_1$

В атрибутных правилах, связанных с правилами вывода 3) и 4), используется процедура-функция без параметров GETNEW, которая выдаёт значение указателя на свободную позицию таблицы, где будут запоминаться промежуточные результаты. Эта процедура не является функцией, т. к. при разных обращениях к ней получаются разные результаты. Пользуясь процедурой GETNEW, мы несколько отступаем от определения АТ-грамматики.

Вместо процедуры GETNEW, можно ввести дополнительные атрибуты для запоминания адресов занятых позиций таблицы. Однако при использовании процедуры GETNEW атрибутные правила получаются более простыми, и такой подход можно порекомендовать при практическом конструировании компиляторов.

АТ-Грамматики используются для получения атрибутных деревьев вывода, атрибутных активных цепочек и атрибутных переводов.

Кр: 2 Алгоритм. Построение атрибутного дерева вывода.

foreach (i [1,4,2]) S =>* w задана

1. Для транслирующей грамматике построить дерево вывода активной цепочки, состоящей из входных и операционных символов без атрибутов.

2. Присвоить начальные значения унаследованным атрибутам начального символа грамматики.

3. Присвоить значения атрибутам входных символов, входящих в дерево вывода.

4. Найти атрибут, отсутствующий в дереве вывода, аргументы правила вычисления которого уже известны. Вычислить значение этого атрибута и добавить его в дерево.

5. Повторять шаг 4 до тех пор, пока значения всех атрибутов символов, входящих в дерево вывода, не будут вычислены или пока шаг 4 может выполняться.

На рис. 5.6 приведено атрибутное дерево вывода в АТ-грамматике, соответствующее входной цепочке $i_5 = i_7 + i_2 * i_3$ при условии, что процедура GETNEW выдаёте последовательные адреса ячеек, начиная с адреса 100. (i2 bug)

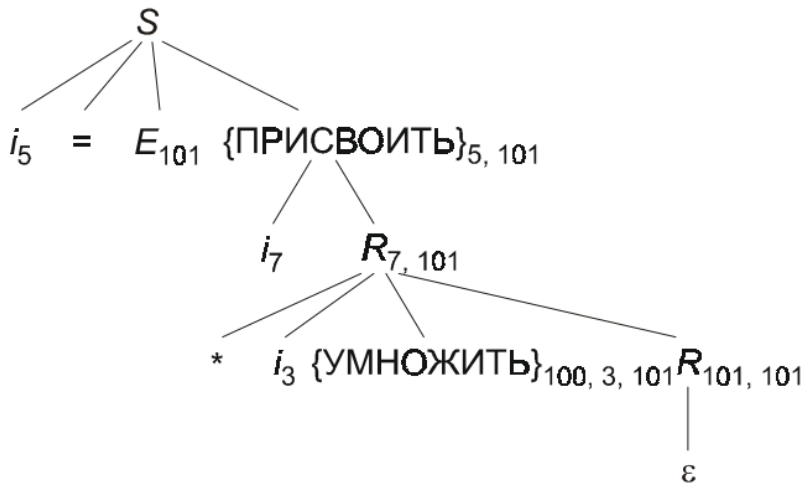


Рис. 5.6

При применении правила (5) синтезированному атрибуту не терминала R присваивается значение его унаследованного атрибута. Это значение передаётся вверх по дереву вывода как синтезированный атрибут не терминала из левой части правил (4), (3) и (2) и используется в качестве второго атрибута операционного символа {ПРИСВОИТЬ}.

Последовательность атрибутных входных и операционных символов, полученная по атрибутному дереву вывода в АТ-грамматике, называется атрибутной активной цепочкой.

Множество пар, первым элементом которых является атрибутная входная цепочка, а вторым элементом - атрибутная последовательность операционных символов атрибутной активной цепочки, называется атрибутным

Пример 6.8. Грамматика G_A для входной цепочки $i_5 = i_7 + i_2 * i_3$ определяет следующую атрибутную последовательность операционных символов:

$$\{+\}_{7,2,100}\ \{*\}_{100, 3, 101}\ \{:=\}_{5,100}$$

Определение. Дерево вывода в АТ-грамматике называется завершённым, если в результате применения алгоритма построения дерева значения всех атрибутов всех символов окажутся вычисленными.

Для каждого атрибута, входящего в дерево вывода, существует правило его вычисления, однако может оказаться, что между атрибутами возникла циклическая зависимость, из-за чего нельзя получить завершённое дерево.

Определение. АТ-грамматика называется корректной тогда и только тогда, когда в результате применения алгоритма построения атрибутного дерева вывода получается завершённое дерево.

Рассмотрим два подкласса корректных АТ-грамматик, используемых при проектировании компиляторов: L-атрибутные транслирующие грамматики и S-атрибутные транслирующие грамматики.

6.4.4. Вычисление значений атрибутов

После построения дерева вывода входной цепочки возникает вопрос о порядке вычисления значений атрибутов.

По определению, атрибут можно вычислить, если известны значения всех атрибутов, от которых зависит его значение. Число деревьев вывода так же, как и входных цепочек, бесконечно, поэтому важно по самой грамматике уметь определять, является ли множество её правил вычисления атрибутов корректным.

Алгоритмы проверки корректности АТ-Грамматики приведены в [2, 25]. Они основаны на построении графа зависимостей и euo анализе.

Узлами графа зависимостей служат атрибуты, которые нужно вычислить, а дугам ставятся в соответствие зависимости, определяющие, какие атрибуты вычисляются раньше, а какие позже.

Граф зависимостей R(D) представляет собой ориентированный граф, который строится для некоторого дерева вывода D следvующим образом:

- узлами R(D) являются пары (X, a), где X - узел дерева D, а a - атрибут символа, служащего меткой узла X;
- дуга из узла (X_1, a_1) в узел (X_2, a_2) проводится, если семантическое правило, вычисляющее значение атрибута a_1, непосредственно использует значение атрибута a_1.

Вычисление значения двоичного числа по его символьному представлению можно задать при помощи следующей атрибутной грамматики [24]:

$N_{v_4} \rightarrow L_{v_2,l_2,S_2} \,.\, L_{v_3,l_3,S_{23}}$

$L_{v_2,l_2,S_2} \rightarrow B_{v_1,S_1}$

$V_4 \leftarrow V_2 + V_3$

$V_2 \leftarrow V_1$

$S_2 \leftarrow 0$

$S_1 \leftarrow S_2$

$S_3 \leftarrow -l_3 l_2 \leftarrow 1$

$N_{v_4} \rightarrow L_{v_2,l_2,S_2}$

$B_{v_1,S_1} \rightarrow 0$

$V_4 \leftarrow V_2 V_1 \leftarrow 0$

$S_2 \leftarrow 0$

$L_{v_2,l_2,S_2} \rightarrow L_{v_3,l_3,S_3}\ B_{v_1,S_1}$

$B_{v_1,S_1} \rightarrow 1$

$V_2 \leftarrow V_3 + V_1 V_1 \leftarrow 1$

$S_1 \leftarrow S_2$

$S_3 \leftarrow S_2 + 1$

$l_2 \leftarrow l_3 + 1$

В этой грамматике атрибуты имеют следующую семантику: v – значение (рациональное число), s – масштаб (целое число) и - длина числа (целое число).

Используя грамматику, построим дерево D вывода цепочки 101.01

Для построения узлов графа зависимостей необходимо последовательно рассмотреть вершины дерева D и для каждой из них построить столько узлов, сколько атрибутов имеет символ, которым помечена эта вершина.

Например, для корня дерева, обозначенного символом N с одним атрибуттом, нужно в граф включить один узел, поместив его парой (N, V_4).

Для определения дуг графа необходимо использовать семантические правила. Например, рассмотрим узел Дерева зависимостей (N, V_4). При построении дерева D непосредственные потомки корня N определялись правилом грамматики $N_{v_4} \rightarrow L_{v_2,l_2,S_2}\ .\ L_{v_3,l_3,S_3}$. Атрибут V_4 не терминала N зависит, в соответствии с семантическим правилом $V_4 \leftarrow V_2 + V_3$, от атрибута V_2 левого сына, помеченного символом L, и атрибута V_3 правого сына. Поэтому в граф зависимостей необходимо включить дуги ((L, V_2), (N, V_4)) и ((L, V_3), (N, V_4)). Поступая аналогичным образом, мы можем построить граф зависимостей.

В [24] доказано, что атрибутная грамматика корректна, если граф зависимостей не имеет циклов.

Общие алгоритмы вычисления атрибутов весьма неэффективны, поэтому на практике используют методы, в которых для определения порядка вычисления не рассматриваются семантические правила. Например, если порядок вычисления атрибутов задаётся методом синтаксического анализа, то он не требует анализа семантических правил и построения графа зависимости.

6.5. Методика проектирования перевода

При проектировании перевода следует руководствоваться следующим:

1. Описание перевода строится последовательно для конструкций входного языка в порядке их усложнения, начиная с простейших конструкций, например: выражение, оператор присваивания, оператор цикла.

2. Выходная цепочка (результат перевода) включает в себя объекты (сущности) трех видов:

• объекты, передаваемые в выходную цепочку из входной цепочки;

• объекты, не изменяющиеся в процессе перевода (терминалы выходного языка);

•объекты, генерируемые во время перевода.

3. Транслирующая грамматика определяет порядок применения операционных символов (элементов перевода), строящих выходную цепочку, атрибуты же используются только для передачи значений символов из одних правил в другие.

4. Операционный символ, включаемый в правило вывода грамматики, может генерировать выходную цепочку до тех пор, пока в неё не должен быть включён объект, перемещаемый из входной цепочки и недоступный в данном правиле грамматики.

Проектирование описания перевода некоторой (одной!) конструкции входного языка выполняется в несколько этапов.

Неформальное описание перевода. Неформальное описание перевода представляет собой пару цепочек, первая из которых является конструкцией входного языка, а вторая - ее представлением в выходном языке. При разработке описания перевода рекомендуется:

- строить описание перевода на основе описания синтаксиса и семантики входного языка, учитывая все возможные форматы представления входной конструкции (например, if-then и if-then-else);
- конструкции, перевод которых уже описан и которые являются частью данной конструкции, считать терминальными;
- после построения неформального описания перевода выделить в нем символы, передаваемые из входной цепочки, и символы, генерируемые при его построении.

Описание синтаксиса. Описание синтаксиса выполняется в БНФ или расширенной БНФ. После стандартных преобразований БНФ преобразуется в КС-Грамматику, описывающую рассматриваемую конструкцию входного языка.

Определение транслирующей грамматики. проанализируем последовательность используемых при построении перевода правил грамматики, а также смоделируем действия, выполняемые операционными символами грамматики. Должны быть уточнены:

1. метод синтаксического анализа, используемый при анализе и переводе данной конструкции, и способ его реализации;

2. интерпретация операционных символов (Цепочка символов, помещаемых на выходную ленту, или имя подпрограммы, которую необходимо выполнить).

- Исходными данными при построении транслирующей грамматики являются: КС-Грамматика, описывающая синтаксис конструкции;
- последовательность правил грамматики (разбор), используемых процессором при построении перевода;
- объекты, доступные процессору, выполняющему перевод (для процессора с магазинной памятью это текущий символ входной цепочки и, в зависимости от метода синтаксического анализа, весь магазин или его часть).

Для построения транслирующей грамматики необходимо выполнить следующие действия:

1.Выбрать простейшую входную цепочку, соответствующую данной конструкции, и определить ее разбор.

2.В неформальном описании перевода выделить очередной (вначале первый) символ выходной цепочки, который должен быть передан в неё из входной цепочки.

3.На основании доступности (видимости) данных процессору и с учётом разбора, определить правило грамматики, в которое нужно включить элемент перевода (операционный символ), переносящий нужный символ в выходную цепочку. Если действия успешны, то перейти к шагу 5, иначе выполнить следующий шаг 4.

4.Если нельзя выполнить шаг 3, то соответствующий операционный символ включить в наиболее подходящее правило грамматики (в дальнейшем этот символ будет передан в него при помощи атрибутов), определив, если нужно, дополнительную выходную ленту процессора.

5.Возложить на новый (включённый) операционный символ действия по записи в выходную цепочку всех символов, вплоть до следующего символа, который должен быть передан из входной цепочки.

6.Повторить шаги 2-5 для всех символов, передаваемых из входной цепочки.

7.Тестировать транслирующую грамматику на более сложных, например, вложенных входных цепочках.

В простых случаях (простая конструкция входного языка, простое ее представление в выходном языке, удачные метод синтаксического анализа и его реализация, удачное расположение операционных символов) может оказаться, что для описания перевода достаточно только транслирующей грамматики.

Определение атрибутной транслирующей грамматики. При необходимости передачи данных между узлами синтаксического дерева транслирующую грамматику следует расширить до атрибутной транслирующей грамматики.

По определению, передача значений атрибутов в атрибутных транслирующих грамматиках возможна только в следующих направлениях:

- от символа левой части символам правой части правила вывода;
- от символов правой части правила символу левой части этого же правила вывода;
- между символами правой части правила вывода.

Для реализации передачи значения символа необходимо выполнить следующие действия:

1. Определить источник передаваемого значения (символ грамматики) и приписать ему атрибут.
2. Определить приёмник значения (символ грамматики) и приписать ему атрибут.
3. Учитывая ограничения в передаче атрибутов, определить маршрут передачи значения, для чего удобно пользоваться синтаксическим деревом грамматики.
4. Описать передачу значения атрибута правилами АТ-грамматики, стараясь, чтобы функции вычисления атрибутов были простейшими (копирующие правила).

Тестирование АТ-грамматики. Тестирование АТ-грамматики заключается в моделировании работы построенного по ней процессора, выполняющего перевод. При большом числе тестов и/или большой их длине эта работа может быть выполнена только при использовании соответствующих средств автоматизации. Для простых языковых конструкций, когда длина теста и их число невелики, а тестирование выполняется отдельно для каждой конструкции, начиная с простейших, работа эта не только необходима, но и реально выполнима.

Тестирование АТ-Грамматики позволяет определять два вида ошибок:

- ошибки в структуре выходной цепочки (неверный порядок следования символов в выходной цепочке);
- ошибки в значениях символов выходной цепочки, что связано с ошибками в передаче значений атрибутов.

При тестировании реального описания перевода желательно использовать комплексные тесты, определяющие оба вида ошибок.

6.6. Пример проектирования АТ-грамматики

Выполним перевод оператора цикла, имеющего следующий формат:

for <параметр> = <начальное значение> to <конечное значение> step <шаг>
<тело цикла> next <параметр>

Введём следующие ограничения:

<параметр> - идентификатор целой переменной;

<начальное значение>, <конечное значение>, <шаг> - целые положительные константы;

<тело цикла> - оператор цикла или терминал (не подлежащий переводу оператор);

кодирование символов входной и выходной программы не рассматривается.

Оператор цикла, который требуется перевести, имеет вид:

for i = c1 to c2 step c3 <тело Цикла> next i

Допустим, что выходной язык не содержит оператора цикла. Тогда результат перевода (рис. 5.9) можно представить следующей последовательностью операторов:

i := c1; m1: if i > c2 then goto m2; <тело цикла>; i := i + c3; goto m1; m2:

Анализ выполняемого перевода позволяет сделать вывод о том, что выходная цепочка включает в себя объекты (сущности) трех видов:

- Объекты, передаваемые в выходную цепочку из входной цепочки (переменная цикла i, начальное значение c1, конечное значение c2, шаг c3).
- Объекты, не изменяющиеся в процессе перевода (множество терминальных символов выходного языка { ;, :, :=, +, >, if, then, goto }.
- Объекты, генерируемые во время перевода (метки m1 и m2).

Составим БНФ, которая описывает синтаксис заданного оператора цикла, выбирая такую структуру описания, которая позволяет проиллюстрировать практически все проблемы синтеза АТ-грамматик:

(1)<оператор цикла> ::= <заголовок цикла><тело цикла>< конец цикла>

(2)<заголовок цикла> ::= for <начальное значение><конечное значение><шаг>

(3)<начальное значение> ::= <идентификатор> = <константа>

(4)<конечное значение> ::= to <константа>

(5)<шаг> ::= step <константа>

(6)<тело цикла> ::= <оператор цикла>

(7)<тело цикла> ::= <терминал>

(8)<конец Цикла> ::= next <идентификатор >

Введём обозначения: s - <оператор цикла>, A - <заголовок цикла>, B - <тело цикла>, C - < конец цикла >, D - <начальное значение>, E - <конечное значение>, F - <шаг>. Тогда правила вывода КС-грамматики, описывающей синтаксис входного языка, имеют вид:

1) S → A B C 5) F → step cns

2) A → for D E F 6) B → s

3) D → id = cns 7) B → term

4) E → to cns 8) C → next id

При построении транслирующей грамматики требуется сделать некоторые предположения о ходе выполнения перевода и порядке анализа входной цепочки и построения выходной цепочки.

Для определённости будем считать, что:

- используется восходящий метод синтаксического анализа, выполняемый процессором с магазинной памятью;
- выполняется цепочечный перевод, в котором операционный символ вида {OUT := "str"} означает запись в выходную ленту OUT цепочки символов "str".

Определим действия, выполняемые первым по порядку элементом перевода (операционным символом), и правило, в которое он должен быть помещён. Для этого ещё раз рассмотрим неформальное описание перевода (см. рис. 5.9). Очевидно, что начать запись в выходную ленту процессор сможет только тогда, когда ему будет доступен параметр цикла id. Когда процессор выполняет свёртку, используя правило грамматики с номером (3), то id входит в основу и Доступен процессору. Следовательно, в это правило мы можем включить

операционный символ, который записывает на выходную ленту начальный участок перевода, вплоть до символа cns. Правило транслирующей грамматики с номером (3) будет выглядеть следующим образом:

(3)D → id = cns { OUT:= "id := cns; m1 : if id >" }.

При выполнении свёртки по этому правилу операционный символ записывает на выходную ленту параметр цикла id. Запись этой переменной в OUT может быть выполнена функцией "читать третий сверху символ магазина процессора и записать его на выходную ленту". Аналогичные действия ("Читать верхний символ магазина и записать его на выходную ленту") выполняются для начального значения параметра цикла cns. Остальные символы не зависят от входной цепочки (терминалы выходного языка) и просто помещаются на выходную ленту.

Рассуждая подобным образом, мы получим следующие правила транслирующей грамматики, содержащие элементы перевода (в порядке их использования при анализе входной цепочки и построении выходной Цепочки):

(4) E→ to cns{ OUT:= "cns then goto m2;" }

(7) B→ term{ OUT:= "term;" }

(5) F→ step cns{ OUT:= "id := id + cns; goto m1; m2 : " }

Добавив остальные правила грамматики, получим следующую транслирующую грамматику:

1) S → A B C

2) A → for D E F

3) D → id = cns{ OUT:= "id := cns; m1 : if id >" }

4) E → to cns{ OUT:= "cns then goto m2;" }

5) F → step cns{ OUT:= "id := id + cns; goto m1; m2: " }

6) B → S

7) B → term{ OUT:= "term;" }

8) C → next id

При восходящих методах синтаксического анализа (см. гл. 8, 9) строится обращённый правый вывод (разбор) цепочки - последовательность номеров правил, используемых при свёртках, - который для рассматриваемого примера равен: (3), 4), 5), 2), 7), 8), 1)). При этом на выходной ленте формируется цепочка:

id := cns1; m1: if id > cns2 then goto m2; id := id + cns3; goto m1; m2: term;

Анализ полученной цепочки позволяет сделать следующие выводы:

1. Тело цикла (терминальный оператор term) включается неправильно (не перед увеличением параметра цикла на значение шага, а в конец выходной цепочки). Это связано с тем, что правило (5) применяется раньше, чем правило (7).

2. При использовании правила (5) значение параметра цикла id недоступно (id вытолкнут из магазина при свёртке по третьему правилу грамматики). Значение id должно быть передано в элемент перевода правила (5) для правильной генерации выходной цепочки.

3. При переводе вложенных циклов возникнет конфликт меток m1 и m2, т. к. m1 и m2 - фиксированные значения. Значения меток должны генерироваться при выполнении перевода и передаваться от места возникновения метки к месту её использования.

Ошибки, обнаруженные в выходной цепочке, можно легко устранить, расширив транслирующую грамматику до атрибутной транслирующей грамматики. Первую ошибку можно исправить несколькими способами:

- включить в правило (4) атрибут, в который записать координаты выходной цепочки, куда должно быть вставлено тело цикла, передать значение этого атрибута в правило (5) и использовать его для выполнения включения тела цикла в выходную Цепочку;
- связать с не терминалом F атрибут, значением которого является строка символов. При использовании правила (5) записать в эту строку операторы, реализующие изменение параметра цикла, затем передать значение этого атрибута в правило (1), где и переписать его значение в выходную цепочку;
- пополнить атрибутный процессор, выполняющий перевод, Дополнительной лентой, на которую записывать операторы, реализующие изменение переменной Цикла в правиле (5). В конце перевода при свёртке по правилу (1) сцепить вспомогательную ленту с выходной лентой.

Применение первых двух способов не вызывает принципиальных затруднений, но неэффективно при реализации процессора, поэтому используем последний способ устранения ошибки: на выходную ленту OUT1 будем за- писывать операторы установки начального значения параметра цикла, проверки завершения цикла и тело цикла, а на дополнительную ленту OUT2 - операторы изменения переменной цикла с заданным шагом.

Вторая и третья ошибки устраняются стандартным способом: включением в правила грамматики атрибутов и функций их вычисления.

Преобразуем транслирующую грамматику в транслирующую атрибутную грамматику.

Рассмотрим правило транслирующей грамматики:

$$D \rightarrow id = cns\{\ OUT1:= "id := cns;\ m1 : if\ id >"\ \}$$

При выполнении свёртки по этому правилу значения терминальных символов id и cns требуется передать из входной цепочки в выходную. Это реализуется передачей соответствующих атрибутов в операционный символ и ис- пользованием их при построении выходной цепочки:

$$D \rightarrow id_{a1} = cns_{a2}\ \{\ OUT1:= "id_{n1} := cns_{n2}\ ;\ m1: if\ id >"\ \}_{n1,\ n2}$$
$$s1, n1 \leftarrow a2; n2 \leftarrow a2$$

Запись "id l" в операционном символе означает, что в данное место выходной цепочки нужно поместить идентификатор, значение которого определяется унаследованным атрибутом n1 (значения id и cns - это не числовые значения этих объектов, а ссылки на таблицы, которые содержат необходимую информацию).

Так как переменная цикла id используется и в других правилах грамматики (правила 5) и 8)), то ее значение нужно передать в эти правила, связав с символом D атрибут и определив функцию для его вычисления. Окончательно правило грамматики примет вид:

3) $D_{s1} \rightarrow id_{a1} = cns_{a2}$ { OUT1:= "$id_{a1} := cns_{n2}$; m1: if id >" }$_{n1,\ n2}$

s1, n1 ⟵ a1;

n2 ⟵a2

Для определения способа передачи значения переменной цикла id (атрибут a1) в правила 5) и 8) грамматики, рассмотрим фрагмент дерева грамматики, изображённый на рис. 5.5.

Процесс передачи значения лексемы id (атрибут a1) из правила (3) грамматики отмечен на рис. 5.5 сплошными линиями:

1. Из правой части правила 3 значение атрибута входного символа a1 присваивается синтезированному атрибуту s1 символа D из левой части этого же правила.
2. Из правой части правила 2 значение синтезированного атрибута s1 нетерминального символа D присваивается унаследованному атрибуту n1 не терминала F из правой части этого же правила.
3. Из левой части правила 5 значение унаследованного атрибута n1 не терминала F передаётся в правую часть унаследованному атрибуту n2 операционного символа.

Соответствующие описанному процессу правила АТ-Грамматики имеют вид:

3) $D_{s1}= id_{a1}$ { OUT1 := "$id_{n1} := cns_{n2}$; m1: if id >" }$_{n1,\ n2}$

s1, n1 ⟵ a1;n2 ⟵ a2

2) A → for D_{s1} E F_{n1}

n1 ⟵ s1

5) F_{n1} → step cns { OUT2 := " id := id +cns; goto m1; m2 : "}$_{n2}$

n2 ⟵ n1

Передача значения параметра цикла в правило (8) грамматики изображено на рис. 5.5 штриховыми линиями и описывается следующими правилами АТ-грамматики:

2. A_{s2} → for D_{s1} E F_{n1}

s2, n1 ⟵ s1

(1)

S → A_{s2} B C_{n1}

n1 ⟵ s2

8) C_{n1} → next id { if n2 □ n3 then Error }

n2 ⟵ n1

Особенностью правила (8) построенной АТ-Грамматики является то, что его операционный символ должен интерпретироваться не как элемент цепочечного перевода, а как выполнение действий, напрямую не связанных с генерацией выходной цепочки. В этот операционный символ передаются значения параметра цикла из заголовка цикла и оператора его завершения. Если эти значения не равны, то должно возникать состояние ошибки процессора (Error).

Процесс генерации и передачи меток иллюстрируется на рис. 5.12. Метки m1 и m2 генерируются в операционных символах правил (3) и (4) грамматики соответственно с помощью процедуры-функции NewLabel, возвращающей при обращении к ней уникальное значение метки. Правила АТ-грамматики, описывающие процесс генерации меток, имеют вид:

3) D_{s1}, s2 → id_{a1} = cns_{a2}
{ OUT1:= "id_{n1} := cns_{n2}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$
s1, n1 ← 01;
n2 ← a2;s3 ← NewLabel;n3 ← s3;2 ← n3;

4) E_{s3} → to cns { OUT1 := "cns_{then} goto $Label_{\text{sl}}$;" }n2
s1 ← NewLabel ;n2 ← s1;s3 ← n2;

2) A → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$
n1 ← s1;n3 ← s3

5) $F_{n1,\ n2,\ n3}$ → step cns { OUT2 := "id := id + cns; goto Label$_{n4}$; Label$_{n5}$:"}$_{n4,\ n5}$
n4 ← n2;n5 ← n3

Сцепление выходных лент OUT1 и OUT2 должно быть выполнено один раз сразу после анализа и перевода всей входной цепочки. для реализации сцепления выходных лент грамматику необходимо пополнить новым (нулевым) правилом, включив в него операционный символ, выполнявши действия по конкатенации выходных лент. Результирующая АТ-грамматика будет иметь вид:

0)S0 → S { OUT1 := OUT1 ‖ OUT2 }
1)S → A_{s1}B C_{n1}
n1 ← S1
2)A_{s4} → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$
s4, n1 ← s1;n2 ← s2;nз ← sз;
3) $D_{s1,s2}$ → id_{a1} = cns_{a2}
{ OUT1 := " id_{n1} := cns_{n2}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$
s1, n1 ← 01; n2 ← 02; s3 ← NewLabel;n3 ← s3;s2 ← n3;
4)E_{s3} → to cns_{a2}; { OUT1 := "cns_{n1} then goto Label$_{s1}$;" }$_{n1,\ n2}$
n1 ← 01;s1 ← NewLabel;
n2 ← s1;s3 ← n2;
5)$F_{n1,\ n2,\ n3}$ → step cns_{a1}
{ OUT2 := "id_{n6} := id_{n6} + cns_{n7}; goto Label$_{n5}$; Label$_{n4}$:" }$_{n4,\ n5,\ n6}$
n4 ← n2;n5 ← n3;n6 ← n1;n7 ← a1;
6)B → S
7)B → term; { OUT1:= "term"; }
8)C_{n1} → next id_{a1} { if n2 ≠ nз then Error }$_{n4,\ n3}$
n2 ← n1;n3 ← a1

При переводе вложенных операторов цикла использовании двух выходных лент в процессоре и их сцепление при выполнении перевода может привести к

ошибкам в структуре выходной цепочки. рассмотрим перевод следующей входной цепочки:

```
for i1 = c5 to c12 step c1з
for i2 = c21 to c22 step c23
top
next i2
next i1
```

разбор которой в виде последовательности номеров правил и синтаксического дерева приведён на рис. 5.13.

К моменту использования правила (О) на выходных лентах OUT1 и OUT2 будут находиться следующие цепочки (на данном этапе проверки считаем, что вычисление атрибутов выполняется правильно):

```
OUT1:
i1 := c5;
Label5: if i1 > c12 then goto Label12;
i2 := c21;
Label21: if i1 > c22 then goto Label22; top;
OUT2:
i1 := i1 + c13; goto Label5; Label12:
i2 := i2 + c23; goto Label21; Label22:
```

После выполнения сцепления выходная (основная) лента будет содержать перевод:

```
OUT1:
i1 := c5;
Label5: if i1 > c12 then goto Label12;
i2 := c21;
Label21: if i1 > c22 then goto Label22; top;
OUT2:
i1 := i1 + c13; goto Label5; Label12:
i2 := i2 + c23; goto Label21; Label22:
i2 := c21;
Label21: if i1 > c22 then goto Label22;
 top;
i1 := i1 + c13; goto Label5;
Label12:
i2 := i2 + c23;
goto Label21;
Label22:
```

Анализ выходной цепочки показывает, что перевод построен неверно: операторы завершения, внешнего и вложенного циклов следуют в обратном порядке. Связано это с тем, что вначале на выходную ленту OUT2 записываются операторы завершения внешнего цикла, и только после этого вложенного. При формировании же перевода на выходную ленту вначале должны быть записаны операторы завершения вложенного цикла, а затем - внешнего. Для того чтобы исправить эту ошибку, необходимо запись на ленту OUT2 и чтение из неё выполнять с одного конца, т. е. реализовать ею как стек с

операциями PushOUT2 ("Втолкнуть на ленту OUT2") и PopOUT2 ("Вытолкнуть с ленты OUT2").

После включения операций PushOUT2 и PopOUT2 в правила (О) и (5) окончательно получим следующий вид атрибутной транслирующей грамматики:

0) S0 → S { while (OUT2 not empty) OUT1 := OUT1 5 PopOUT2 }

1)S → A_{s1} B C_{n1}

n1 ← s1

(2)

A_{s4} → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$

s4, n1 ← s1;n2 ← s2;n3 ← s3;

3)$D_{s1,s2}$ → id_{n1} = cns_{a1}

{ OUT1 := "id_{n1} := cns_{n1}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$

s1, n1 ← a1;n2 ← a2;s3 ← NewLabel;n3 ← s3;s2 ← n3;

4)E_{s3} → to cns_{a1} { OUT1 := "cnsnl then goto Label$_{s3}$;" }$_{n1,\ n2}$

n1 ← a1;s1 ← NewLabel;n2 ← s1;s3 ← n2;

5) $F_{n1,\ n2,\ n3}$ → step cns_{a1}

{ PushOUT2("id := "id_{n6} := id_{n6} + cns_{n7}; goto Label$_{n5}$; Label$_{n4}$:" :") }$_{n4,\ n5,\ n6}$

6) B → 8

7)B → term; { OUT1 := "term; }

8)C_{n1} → next id_{a1} { if n2 ≠ n3 then Error }$_{n2,\ n3}$

n2 ← n1;n3 ← a1

Рекомендуется самостоятельно проверить правильность окончательного варианта построения АТ-Грамматики для различных тестовых входных цепочек.

Контрольные вопросы

1.Как определяется понятие семантики языков программирования?

2.Как можно описать простейшие переводы?

3.Какие переводы называются синтаксически управляемыми?

4.Что такое элемент перевода?

5.Дайте определение СУ-схемы. Какие цепочки называются входными и выходными?

6.Как определить входную и выходную грамматики СУ-схемы?

7.Как происходит порождение пар цепочек под управлением СУ-схемы?

8.Как происходит преобразование деревьев под управлением СУ-схем?

9.Какие СУ-схемы называются простыми?

1О. Какие переводы позволяют описать простые СУ-схемы?

11.Какая грамматика называется транслирующей гамматикой?

12.Что такое входная и выходная грамматики транслирующей грамматики?

13. Как определяется активная цепочка, ее входная и операционная части?

14. Как связаны вывод цепочки и дерево вывода активной цепочки?

15. Какая Грамматика называется атрибутной транслирующей грамматикой?

16. Какие атрибуты называются унаследованными и синтезированными?

17. В чем основные различия между унаследованными и синтезированными атрибутами?

18. Что представляет собой правило вычисления атрибутов?

19. Как строится граф зависимостей?

2О. Как граф зависимостей используется при вычислении атрибутов?

21.Какие виды объектов включаются в выходную цепочку при переводе?

22.Из каких этапов состоит процесс построения транслирующей грамматики?

23.Из каких этапов состоит процесс построения АТ-Грамматики?

24. Какие типы ошибок позволяет найти тестирование АТ-Грамматики?

Упражнения

1.Разработайте простую СУ-схему, описывающую перевод арифметических скобочных выражений, содержащих операции '+' и '*' в:

1.1.постфиксную запись.

1.2.префиксную запись.

2.В языке ALGOL-6О выражения строятся с помощью операций, имеющих следующие приоритеты (в порядке убывания):

1)↑ 4)≤ < = ≠ > ≥ 7) ∨

2)*/ ÷ 5) ¬ 8) →

3)+ - 6)∧ 9) ≡

Разработайте простую СУ-схему, описывающую перевод инфиксных выражений, содержащих перечисленные операции, в постфиксную запись.

3.Постройте МП-преобразователи, реализующие СУ-схемы из упр. 1.

4.Для простой СУ-схемы:

E → + EE, EE +

E → * EE, EE *

E → a, a

постройте эквивалентный МП-преобразователь.

5.Для заданной СУ-схемы:

8 → a $S^{(1)}$ b A c $S^{(2)}$ d $S^{(3)}$, e A c $S^{(2)}$ d $S^{(3)}$ f $S^{(1)}$ g

S → i, i

A →i, i

постройте переводы для входных цепочек:

5.1.aibicd .

5.2.aaibicidibicid .

6.Постройте вывод в транслирующей грамматике G0T цепочек:

6.1.i* (i+i).

6.2.(i+i) * (i + i).

7.Определите транслирующую грамматику, допускающую в качестве входа произвольную цепочку из нулей и единиц и порождающую на выходе:

7.1.обращение входной цепочки.

7.2.цепочку 0n1m, где n - число нулей, а m - число единиц во входной цепочке.

8.Постройте транслирующую грамматику, определяющую перевод логических выражений, составленных из логических переменных, скобок и знаков операций дизъюнкции, конъюнкции и отрицания:

8.1.из инфиксной записи в ПОЛИЗ.

8.2.из ПОЛИЗ в инфиксную запись.

8.3.из инфиксной записи в функциональную запись такую, например, что выражение a ∩ (b ∪ c) будет представлено в виде f∩ (a, f∪ (b, c)), где f∩ и f∪ - отдельные символы.

9.Постройте АТ-Грамматику, описывающую перевод оператора присваивания некоторого Гипотетического языка программирования в цепочку тетради с кодами операций: ПРИСВОИТЬ, СЛОЖИТЬ, ВЫЧЕСТЬ, УМНОЖИТЬ, ДЕЛИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью бесскобочное арифметическое выражение, выполняемое справа налево в порядке написания операций. В арифметическом выражении можно использовать идентификаторы и знаки арифметических операций: +, -, *, /.

10. Постройте АТ-грамматику, описывающую перевод оператора присваивания некоторого гипотетического языка программирования в цепочку тетрад с кодами операций: ПРИСВОИТЬ, И, ИЛИ. НЕ. левой частью оператора присваивания является идентификатор, а правая часть представляет собой логическое выражение, составленное из идентификаторов, скобок и знаков логических операций: ∩, ∪, ¬. порядок выполнения логического выражения определяется обычным приоритетом логических операций.

11. Постройте АТ-грамматику, описывающую перевод в тетрадку с кодом операции ПЕРЕХОД_ПО_РАВНО условного оператора, которому соответствует следующее правило входной грамматики:

условный оператор → if = goto .

Лексема представляет собой номер оператора, к которому осуществляется переход в случае равенства, а - идентификатор.

12. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

S → EE → (E)

E → E := EE → i

E → E + E

где - лексема, значением которой является указатель на элемент таблицы идентификаторов, а семантика выражений такая же, как в языке С, постройте АТ-грамматику, которая проверяет, представляет ли левая Часть выражения значение, и, в случае успеха, стройте ПОЛИЗ выражения.

13. Для входной грамматики, описывающей объявление переменных, с правилами вывода:

1. D→ L 4.→ int

2. L→ , L 5.→ real

3. L→ :

постройте АТ-грамматику, которая заносит значение типа каждого идентификатора в таблицу идентификаторов.

14. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

E → (E + E)

E → (E * E)

E → i

где - лексема, значением которой является указатель на элемент таблицы идентификаторов, определить постфиксную 8-атрибутную грамматику, описывающую перевод этих выражений в цепочку тетради с концами операций: СЛОЖИТЬ и УМНОЖИТЬ, и построить 8-атрибутный ДМП-процессор, выполняющий заданный перевод.

Глава 7. Языковые процессоры: L-атрибутные и S-атрибутные транслирующие грамматики

L-атрибутные и S-атрибутные транслирующие грамматики - два подкласса корректных АТ-грамматик, часто используемых при проектировании языковых процессоров.

7.1. L-атрибутные и S-атрибутные транслирующие грамматики

++КР алгоритм проверки АТ-грамматики.

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

1. Аргументами правила вычисления значения унаследованного атрибута символа из правой части правила вывода могут быть только унаследованные атрибуты символа из левой части и произвольные атрибуты символов из правой части, расположенные левее рассматриваемого символа.
2. Аргументами правила вычисления значения синтезированного атрибута символа из левой части правила вывода являются унаследованные атрибуты этого символа или произвольные атрибуты символов из правой части.
3. Аргументами правила вычисления значения синтезированного атрибута операционного символа могут быть только унаследованные атрибуты этого символа.

Является ли произвольная АТ-грамматика L-атрибутной, можно проверить, независимо исследуя каждое правило вывода и каждое правило вычисления значения атрибута. Примером L-атрибутной транслирующей грамматики является грамматика G^A.

При выполнении условия 1 определения унаследованные атрибуты каждой вершины дерева вывода зависят (непосредственно или косвенно) только от атрибутов входных символов, расположенных в дереве левее данной вершины, что позволяет использовать L-атрибутные грамматики в нисходящих синтаксических анализаторах. Условия 2 и 3 введены с целью сделать АТ-грамматику корректной.

В L-атрибутной транслирующей грамматике атрибуты символов A, B и C из правила вывода

A →BC можно вычислять в следующем порядке:

унаследованные атрибуты символа A;

унаследованные атрибуты символа B;

синтезированные атрибуты символа B;

унаследованные атрибуты символа C;

синтезированные атрибуты символа C;

синтезированные атрибуты символа A.

Определение. АТ-грамматика называется S-атрибутной тогда и только тогда, когда она является L-атрибутной и все атрибуты нетерминалов синтезированные.

Ограничения, накладываемые на L-атрибутную транслирующую грамматику, позволяют вычислять значения атрибутов в процессе нисходящего анализа входной цепочки. Нисходящий детерминированный анализатор для LL(1)-

грамматик требует, чтобы L-атрибутная транслирующая грамматика, описывающая перевод, имела форму простого присваивания.

7.2 Форма простого присваивания

При определении АТ-грамматики в правила вычисления атрибутов записывались в виде операторов присваивания, левые части которых представляют собой атрибут или список атрибутов, а правые части — функцию, использующую в качестве аргументов значения некоторых атрибутов. Простейший случай функции из правой части оператора присваивания - тождественная или константная функция, например a ← b или x, y←1.

Определение. Правило вычисления атрибутов называется **копирующим правилом** тогда и только тогда, когда левая часть правила - это атрибут или список атрибутов, а правая часть – константа или атрибут. Правая часть называется источником копирующего правила, а каждый атрибут из левой части – приемником копирующего правила.

Если источники нескольких копирующих правил совпадают, то их приемники можно объединить в одну левую часть. Например, правила z, w← a и x, y←z можно записать в виде: x, y, z, w← a , т. к. источнику второго правила z, согласно первому правилу, присваивается значение a. Аналогично x←y и a, b ← y можно записать как a, b, x ←y.

Определение. Множество копирующих правил называется независимым, если источник каждого правила из этого множества не входит ни в одно из других правил множества.

Два независимых копирующих правила, нельзя объединять.

Определение. Атрибутная транслирующая грамматика имеет форму простого присваивания тогда и только тогда, когда:

1. Некопирующими являются только правила вычисления синтезируемых атрибутов операционных символов.
2. Для каждого правила вывода грамматики соответствующее множество копирующих правил независимо.

Пример. Пусть дана L-атрибутная транслирующая грамматика, порождающая префиксные арифметические выражения над константами, в форме простого присваивания. Атрибутные правила вывода этой грамматики имеют следующий вид: E_p - синтезированный p, $\{ответ\}_r$ - унаследованный r:

$G = (\{S, E_p\}, \{c_r, *, +\}, \{\{ответ\}_r, P, S\})$, где

P = {

1. $S \rightarrow E_p\{ответ\}_r$
   $r \leftarrow p$
2. $E_p \rightarrow +E_q\, E_r\, \{сложить\}_{A,B,R}$
   $p \leftarrow q+ r$
3. $E_p \rightarrow *E_q\, E_r$
   $p \leftarrow q* r$
4. $E_p \rightarrow c_r$
   $p \leftarrow r$

}

Правила преобразования:

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

   $\{f\}$ x_1 x_2, ... , x_n, p

   унаследованные $x_1, x_2, \ldots, x_n$

   синтезированный p

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

2. Для каждого некопирующего правила z_i, $z_2, \ldots, z_ь \leftarrow f(y_1, y_2, \ldots, y_n)$ связанного с некоторым правилом вывода грамматики, найти символы a1 ... , a,,, которые не содержатся в этом правиле вывода, и вставить в его правую часть символ$\{f\}a_n \leftarrow a_2, \ldots, a_n$. Заменить некопирующее правило на следующие (n + 1) копирующих правил:

   $a_i \leftarrow y_n$ для каждого аргумента y_i,

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

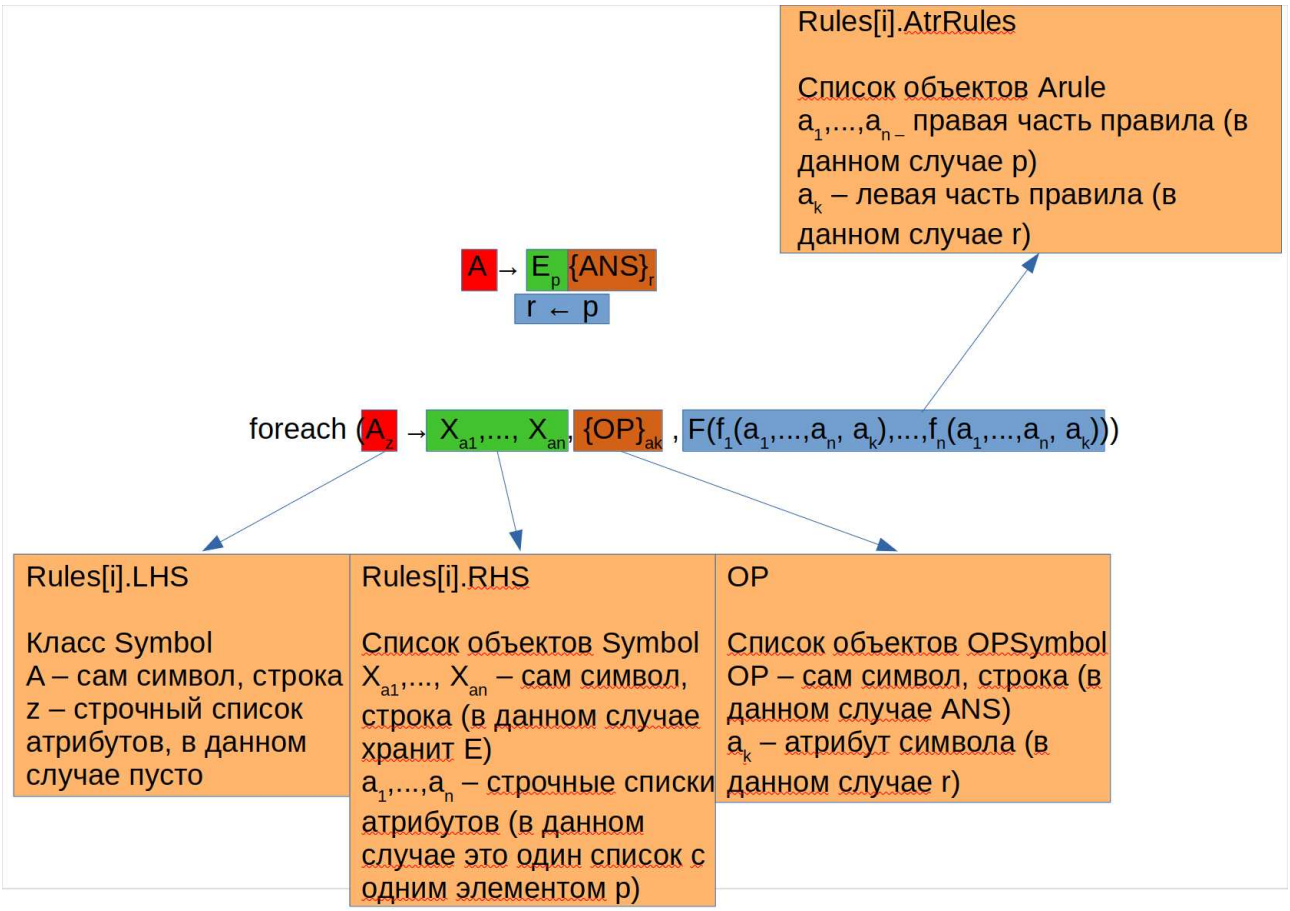
При включении в правило вывода операционного символа необходимо соблюдать следующие условия:

- операционный символ должен располагаться правее всех символов правой части правила вывода, атрибутами которых являются аргументы $y_1, \ldots, y_n$;
- операционный символ должен располагаться левее всех символов правой части правила вывода, атрибутами которых служат $z_1, \ldots, z_n$;
- с учётом предыдущих ограничений операционный 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.

2. 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.

В фреймворке реализация под пунктом **I4**.



Структура данных

Symbol – класс символа
Содержит в себе поля:

ARule – класс правила для атрибутов
Содержит в себе поля:

ArrayList LH – левая часть правила
ArrayList RH – правая часть

OPSymbol – класс

Rule – класс правила
Содержит в себе поля:

Symbol LHS – левая часть правила
List<Symbol> RHS – правая часть правила, состоящая из списка символов
List<ARule> AtrRules – список правил для атрибутов

```
public class ATgrammar {
    public class Symbol{
        public string sym=null;
        public ArrayList atrs=null; // список атрибутов символа

        public Symbol(string s, ArrayList a){
            sym = s;
            atrs = new ArrayList(a);
        }
        public Symbol(string s){
            sym = s;
        }

        public void print(){
            Console.Write(sym);
            if(atrs !=null){
                Console.Write("_{");
                for(int i =0; i < atrs.Count; ++i){
                    Console.Write(atrs[i]);
                    if(i != (atrs.Count -1)){
                        Console.Write(", ");
                    }
                }
                Console.Write("} ");
            }
        }
    }

    public class OPSymbol{
        public Symbol sym=null;
        public ARule rul=null;

        public OPSymbol(string s, ArrayList a, ArrayList L, ArrayList R){
            sym = new Symbol(s, a);
            rul = new ARule(L, R);
        }

        public OPSymbol(string s, ArrayList a){
            sym = new Symbol(s, a);
        }

        public void print_sym(){
            sym.print();
        }

        public void print(){
            sym.print();
            Console.Write("\n");
            rul.print();
        }
    }
```

```
public class Rule{
    public Symbol LHS; // left part of rule
    public List<Symbol> RHS; // right part of rule
    public List<ARule> AtrRules; //rules for atributes

    public Rule(Symbol L,  List<Symbol> R,  List<ARule> ARules){
        LHS = L;
        RHS = new List<Symbol> (R);
        AtrRules = new List<ARule> (ARules);
    }

    public void print(){
        LHS.print();
        Console.Write(" -> ");
        for(int i=0; i < RHS.Count; ++i){
            RHS[i].print();
        }
        Console.Write("\n");
        for(int i=0; i < AtrRules.Count; ++i){
            AtrRules[i].print();
            if( i != (AtrRules.Count -1))
                Console.Write("\n");
        }

    }


}

public class ARule{
    public ArrayList LH; // left part of Arule
    public ArrayList RH; // right part of Arule

    public ARule(ArrayList L, ArrayList R){
        LH=new ArrayList(L);
        RH=new ArrayList(R);
    }
    public void print(){
        for(int i =0; i < LH.Count; ++i){
            Console.Write(LH[i]);
            if(i != (LH.Count -1)){
                Console.Write(", ");
            }
        }
        Console.Write(" <- ");
            for(int i =0; i < RH.Count; ++i){
            Console.Write(RH[i]);
            }
    }
}
```

```
public List<Symbol> V = null; //список нетерминалов
public List<Symbol> T = null; //список входных символо
public List<OPSymbol> OP = null; //список операционных символов
public List<Rule> Rules = new List<Rule>(); //список правил порождения
public Symbol S0; //начальный символ
// public ArrayList OPrules = new ArrayList(); //список правил оперционных символов

public ATgrammar(){}
public ATgrammar(List<Symbol> Ns,List<Symbol> Ts, List<OPSymbol> OPs /* типо правила  ,*/, Symbol S0){
    this.OP=OPs;
    this.V=Ns;
    this.T=Ts;
    this.S0=S0;
} //конструктор

public void Addrule(Symbol LeftNoTerm /*левый нетерминал и его атрибуты*/, List<Symbol> Right /*правая часть правил  в виде символов*/, List<ARule> AtrRule /*правила атрибутов*/ ){
    this.Rules.Add(new Rule(LeftNoTerm, Right, AtrRule));
} //добавление правила
```

```
public void Print(){//печать грамматики
    Console.Write("\nAT-Grammar G = (V, T, OP, P, S)");
    Console.Write("\nV = { ");//нетерминальные символы
    for( int i = 0; i < V.Count; ++i){
        V[i].print();
        if(i != V.Count-1)
            Console.Write(", ");
    }
    Console.Write(" },");

    Console.Write("\nT = { ");//терминальные
    for( int i = 0; i < T.Count; ++i){
        T[i].print();
        if(i != T.Count-1)
            Console.Write(", ");
    }
    Console.Write(" },");

    ArrayList OPRules = new ArrayList(); //счётчик операционных символов, у которых есть правила атрибутов
    Console.Write("\nOP = { ");//операционные
    for( int i = 0; i < OP.Count; ++i){
        OP[i].print_sym();
        if(OP[i].rul != null)
            OPRules.Add(i);
        if(i != OP.Count-1)
            Console.Write(", ");
    }
    Console.Write(" },");

    Console.Write("\nS = ");
    S0.print();

    //печать правил атрибутов операционных символов
    if(OPRules.Count !=0){
        Console.Write("\nOperation Symbols Rules:\n");
        foreach(int i in OPRules){
            OP[i].print();
            Console.Write("\n");
        }
    }
    //печать правил грамматики
    if(Rules.Count !=0){
        Console.Write("\nGrammar Rules:\n");
        for( int i = 0; i < Rules.Count; ++i){
            Console.Write("\n");
            Rules[i].print();
            Console.Write("\n");
        }
    }
}
```

```
    private bool IsOper(string s){
        return s == "+" || s == "-" || s == "*" || s == "/";
    }
    public void transform(){
        Console.WriteLine("\nPress Enter to start\n");
        Console.ReadLine();
        for(int i = 0; i < Rules.Count; ++i){
            for(int j= 0; j < Rules[i].AtrRules.Count; ++j ){//обработка jго атрибутного правила iго правила грамматики
                string NewOpS = "";
                ArrayList atrs= new ArrayList();
                ArrayList atrs_l= new ArrayList();
                for(int k= 0; k < Rules[i].AtrRules[j].RH.Count; ++k ){ //проверка наличия функции в правой чатси правила
                    if(IsOper((string)Rules[i].AtrRules[j].RH[k])){
                        NewOpS += Rules[i].AtrRules[j].RH[k]; //создание имени для нового операционного символа
                    }else{
                        atrs.Add(Rules[i].AtrRules[j].RH[k] + "'"); //создание дублирующих символов для правил A <- a, но в формате a' <- a.
                        atrs_l.Add(Rules[i].AtrRules[j].RH[k]);  //список атрибутов, входящих в функцию

                    }
                }
                if((NewOpS.Count()) == 0 ) // проверка, что нет функций в правй части правила
                    continue;
                NewOpS += i.ToString()  + j.ToString(); //создание более уникального имени операционного символа
                atrs.Add(atrs[0]+"_ans"); // добавление атрибута для результата функции


                this.OP.Add(new OPSymbol("{"+NewOpS+"}", atrs, new ArrayList(){ atrs[0]+"_ans"}, Rules[i].AtrRules[j].RH));// добавление операционного символа с атрибутами и атрибутным правилом
                for(int k=0; k < atrs.Count-1; ++k){ //добавление копирующих правил a' <- a
                    Rules[i].AtrRules.Add(new ARule(new ArrayList(){atrs[k]}, new ArrayList(){atrs_l[k]}));
                }
                Rules[i].AtrRules.Add(new ARule(new ArrayList(Rules[i].AtrRules[j].LH), new ArrayList(){atrs[0]+"_ans"}));//добавление правила z1, ... , zm <- p, где p - результат функции операционного символа
                Rules[i].AtrRules.RemoveAt(j); //удаление правила с функцией в правой части
                j-=1;
                for(int k=Rules[i].RHS.Count-1; k>=0 ; --k){//поиск самой левой позиции для вставки операционного символа, начиная с самой правой позиции
                    int k1;
                    if(Rules[i].RHS[k].atrs == null)//проверка того, что есть атрибуты у к-го символа правой части правила грамматики
                        continue;
                    for(k1=0; k1< Rules[i].RHS[k].atrs.Count; ++k1){//проверка, что у к-го символа нет атрибута, который есть у операционного символа, если он есть, то дальше мы не двигаемся и вставляем операционный символ перед ним, инче идём дальше до конца
                        if(atrs_l.Contains(Rules[i].RHS[k].atrs[k1]))
                            break;
                    }
                    if(k1<Rules[i].RHS[k].atrs.Count){ //нашли такой символ, справа от которого вставляем операционный
                        Rules[i].RHS.Insert(k+1,new Symbol("{"+NewOpS+"}", atrs));
                        break;
                    }
                    if(k==0){ //дошли до конца правила и не нашли символа с хотя бы одним атрибутом, совпадающим с атрибутами операционного символа. Такого быть не должно, т.к. это означает, что атрибутные правила содержат атрибуты, отсутствующие у правила грамматики
                        Rules[i].RHS.Insert(k,new Symbol("{"+NewOpS+"}", atrs));
                        break;
                    }
                }
            }
            //поиск лишних атрибутов в правилах типа
            //a1, ... , am <- k
            //b1, ..., k, ..., bn <- g
            // и замена на b1, ..., a1, ... , am, ..., bn <- g с удалением правила a1, ... , am <- k
            for(int r = 0 ; r < Rules[i].AtrRules.Count; ++r){
                bool deleted = false;
                for(int l = r+1 ; l < Rules[i].AtrRules.Count; ++l){
                    if(Rules[i].AtrRules[l].LH.Contains(Rules[i].AtrRules[r].RH[0])){
                        Rules[i].AtrRules[l].LH.Remove(Rules[i].AtrRules[r].RH[0]);
                        deleted = true;
                        foreach(string s in (Rules[i].AtrRules[r].LH))
                            Rules[i].AtrRules[l].LH.Add(s);
                    }
                }
                if(deleted){
                    Rules[i].AtrRules.RemoveAt(r);
                    r-=1;
                }
            }
            Console.WriteLine("\nChange for " +(i+1).ToString() +"th rule\n");
            Rules[i].print();
            Console.ReadLine();
        }
    }

}

}//end abstract class Grammar
```

Алгоритм преобразования произвольной L-атрибутной транслирующей грамматики в эквивалентную L-атрибутную грамматику в форме простого присваивания.

Вход: КС грамматика G = (T, V, P, S_0)
Выход: эквивалентная КС грамматика G' = (T, V, P', S_0)

foreach ($A_p \rightarrow X_{a1},..., X_{an}$, {$OP_1,..., OP_{ak}$ }, p <- F{$f_1(a_1,...,a_n, a_k),...,f_n(a_1,...,a_n, a_k)$})
  foreach (f $\in$ F)
    a = $\varnothing$ пустое множество атрибутов
    if (f $\neq f_o$) не копирующее, f_o - копирующая функция
      foreach ($a_j \in \{a_1,...,a_n, a_k\}$)
        if($a_j \notin$ T)
          F = F $\cup$ $f_0(a_j', a_j)$
          a = a $\cup$ a_j'
        end if
      end foreach
      F = F $\cup$ $f_0(a_1, a_1'\_ans)$ - копирующая функция
      a = a $\cup$ $a_1'\_ans$
      F = F – f //удалить из правил правило f
      OP_1 ,..., OP_{ak} = OP_1 ,..., $OP_{ak} \cup Op_a$ добавить новый операционный символ
    end if
  end foreach

end foreach

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 ← q + r и p ← 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 {УМНОЖИТЬ}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 ← A + B, а для операционного символа {УМНОЖИТЬ}A, B, R — R ←A X B.

Для того чтобы преобразованная грамматика также была L-атрибутной, символ {СЛОЖИТЬ} необходимо поместить правее всех символов правой части правила вывода (2), т. к. одним из аргументов сложения является атрибут самого правого символа E. Атрибут, получающий в качестве своего значения результат сложения, в определении места расположения символа {СЛОЖИТЬ} не участвует, т. к. он не приписан ни к одному из символов правой части. Аналогичные рассуждения относительно операционного символа {УМНОЖИТЬ} определяют крайнюю правую позицию правой части правила (3), как единственно возможное место расположения этого символа. Полученная в результате преобразования L-атрибутная грамматика в форме простого присваивания имеет вид:

Ep синтезированный p
{ОТВЕТ}r, унаследованный r
{СЛОЖИТЬ}A, B, R унаследованный A, B
R ← A + B синтезированный R
{УМНОЖИТЬ}A, B, R унаследованный A, B
R ←A* B синтезированный R

$S \rightarrow E_p$ {ОТВЕТ}r
r ← p
$E_p \rightarrow + E_q E_p$ {СЛОЖИТЬ}A, B, R
A← q
B ←r
P ←R
$E_p \rightarrow * E_q E_p$ {УМНОЖИТЬ}A, B, R
A←q
B← r
p ←R

$$E_p \rightarrow c_r$$
$$p \leftarrow r$$

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.

7.3. 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.

7.3.1 Реализация синтаксически управляемого перевода для транслирующей грамматики

Преобразуем "1-предсказывающий" алгоритм разбора для LL(1)-грамматик, включив в него действия, обеспечивающие перевод входной цепочки, порождаемой входной грамматикой, в цепочку операционных символов и запись этой цепочки на выходную ленту. Преобразованный таким образом алгоритм в дальнейшем будем называть нисходящим детерминированным процессором с магазинной памятью (нисходящий ДМП-процессор). При этом, если из контекста ясно, что речь идет о нисходящем методе анализа, слово "нисходящий" будем опускать.

В транслирующей грамматике множество терминальных символов разбито на множество входных символов $\sum_i$, и множество операционных символов $\sum_a$.

Расширим алфавит магазинных символов, добавив в него операционные символы. Тогда $V_p = \sum_i \cup \sum_a \cup N$. Затем доопределим управляющую таблицу M, которая для транслирующей грамматики цепочечного перевода задает отображение множества $(V_p \cup \{\perp\})$ X $(\sum I \cup \{\in\})$ в множество, состоящее из следующих элементов:

(β, у), где $\beta \in V_p^*$ — цепочка из правой части правила транслирующей грамматики A →уβ, а $y \in \sum_a^*$;

ВЫДАЧА (X), где $X \in \sum_a$

ВЫБРОС;

ДОПУСК;

ОШИБКА.

Пусть FIRST(x) = a, где x — неиспользованная часть входной цепочки. Тогда работу ДМП-процессора в зависимости от элемента управляющей таблицы M(X, o) можно определить следующим образом:

1. (x, Xα, π) ⊢ (x, βα), если M(X, a) = (β, y). Верхний символ магазина X заменяется цепочкой β ∈ Vp*, и в выходную цепочку дописывается цепочка операционных символов y. Входная головка при этом не сдвигается.

2. (x, Xα, π) ⊢(x, α, πX), если M(X, a) = ВЫДАЧА{X). Это означает, что если верхний символ магазина — операционный символ, то он выталкивается из магазина и записывается на выходную ленту. Входная головка не сдвигается.

Действия, соответствующие элементам управляющей таблицы: ВЫБРОС, ДОПУСК и ОШИБКА, остаются теми же, что и в "1-предсказывающем" алгоритме для LL(1)-грамматик.

Опишем алгоритм построения управляющей таблицы M.

Кр: (есть)

Алгоритм построения управляющей таблицы для транслирующей грамматики цепочечного перевода, входной грамматикой которой является LL(1)-грамматика.

Вход: Транслирующая грамматика цепочечного перевода G^{Tb} = (T, $\sum_i$, $\sum_a$, P, S), входная грамматика которой является LL(1)- грамматика.

Выход : Корректная управляющая таблица M для грамматики G^{Tb}.

Описание алгоритма:

Управляющая таблица M определяется на множестве (N ∪ $\sum_i$ ∪ $\sum_a$ ∪{⊥}) X $\sum_i$ ∪ {ε}) по следующим правилам:

Если A→yβ - правило вывода грамматики GT, где y ∈ $\sum_a$* , a 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.

M(X, a) = ВЫДАЧА(X) для всех x ∈ $\sum_a$ И a ∈ $\sum_i$ ∪ {ε}.

M(a, a) = ВЫБРОС для всех a ∈ $\sum_i$.

M (⊥ , ε) = допуск

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 G1 путем включения в нее операционных символов {i}, {+} и {*}:

E →E ′ (5)T′ →* P{*} T′

E ′ →+ T{+} E′6)T′ →ε

E ′ →ε

P →i {i}

T →PT′

(8)P →(E)

Управляющая таблица должна содержать 14 строк, помеченных символами из множества N∪∑i ∪ ∑a ∪ {⊥}, и 6 столбцов, помеченных символами из множества $\sum_i \cup \{ \varepsilon \}$.

Построение управляющей таблицы для строк, отмеченных символами из множества N∪∑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	(	)	+	*	E
E	TE’ ,ε	TE’,ε				
E’			ε, ε	*P{*} T’, ε		ε, ε
T	PE’, ε	PE’,ε				
T’			ε, ε	ε, ε	*P{*} T’, ε	ε, ε
P	i{i} ,ε	(E),ε				
i	ВЫБРОС					
(		ВЫБРОС				
)			ВЫБРОС			
+				ВЫБРОС		
*					ВЫБРОС	
⊥						ДОПУСК
{i}	ВЫДАЧА({i})					
{+}	ВЫДАЧА({+})					
{*}	ВЫДАЧА({*})					
Начальное содержимое магазина - E⊥						

Начальное содержимое магазина — E⊥

Для входной цепочки i +i * i ДМП-процессор проделает следующую последовательность тактов:

(i +i * i , E⊥, ε) ⊢ (i +i * i , TE'⊥, ε)

⊢ (i +i * i , PT'E'⊥, ε)

⊢(i +i * i , {i} T'E'⊥, ε

⊢ (+ i * i , {i}T'E'⊥, ε)

⊢ (+ i * i , {i}T'E'⊥, {i})

⊢ (+i * i , {i}E'⊥, {i})

⊢ (+i * i , +T {+} E'⊥,{i})
⊢ (i * i , T {+} E'⊥,{i})
⊢ (i * i , PT '{+} E'⊥, {i})
⊢ (i * i , i {i} T '{+} E'⊥, {i})
⊢ (*i , {i} T '{+} E'⊥, {i})
⊢ (* i , T '{+} E'⊥, {i} {i})
⊢ (* i , *P{*}T'{+}E'⊥, {i} {i})
⊢ (i , P{*}T'{+}E'⊥, {i} {i})
⊢ (i , i {i} {*}T'{+}E'⊥, {i} {i})
⊢ (ε , {i}{*}T'{+}E'⊥, {i} {i})
⊢ (ε , {i}{*}T'{+}E'⊥, {i} {i})
⊢ (ε , T'{+}E'⊥, {i} {i}{i}{*})
⊢ (ε , {+}E'⊥, {i} {i}{i}{*})
⊢ (ε , E'⊥, {i} {i}{i}{*}{+})
⊢ (ε , ⊥, {i} {i}{i}{*}{+})

ДМП-процессор можно использовать в качестве базового процессора для других видов синтаксически управляемого перевода, если операцию выдачи операционного символа в выходную ленту заменить операциями вызова соответствующих семантических процедур.

7.3.2 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.

КР L-атрибутный ДМП-процессор**:** 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.

!!! Магазинный символ с n атрибутами представляется в магазине **объектом** (n + 1)-ой ячейками, верхняя из которых содержит имя символа, а остальные — поля для атрибутов. Поля магазинного символа доступны для записи и извлечения атрибутов от момента вталкивания символа в магазин до момента выталкивания его из магазина.

В момент вталкивания символа в магазин в поле для каждого синтезированного атрибута и атрибута входного символа заносится указатель на связанный список полей, соответствующих унаследованным атрибутам, где этот

атрибут должен запоминаться, а поле для каждого унаследованного атрибута остается пустым. Содержимое полей синтезированных атрибутов и атрибутов входных символов остается неизменным в течение всего времени нахождения символа в магазине, а поля, соответствующие унаследованным атрибутам, приобретают значения атрибутов к моменту времени, когда магазинный символ окажется в верхушке магазина.

+++

Пусть x— остаток входной цепочки, и FIRST(x) = a. Опишем действия, которые должен выполнять L-атрибутный ДМП-процессор в зависимости от элемента управляющей таблицы M(X, o), где X— символ в верхушке магазина.

Начальная конфигурация. В магазине находится маркер дна и начальный символ грамматики. Поля начального символа грамматики, соответствующие унаследованным атрибутам, заполняются начальными значениями атрибутов, которые задаются L-атрибутной транслирующей грамматикой, а в поля, соответствующие синтезированным атрибутам, заносятся пустые указатели, которые служат маркерами конца списков.

M(X, a) = ВЫБРОС (символ в верхушке магазина совпадает с текущим входным символом). В этом случае каждое поле верхнего магазинного символа содержит указатель на список полей магазина, в которых требуется поместить значение атрибута текущего входного символа. Операция ВЫБРОС расширяется таким образом, что каждый атрибут текущего входного символа копируется во все поля списка на который указывает соответствующее поле верхнего магазинного символа.

M(X, a) = ВЫДАЧА(X) (в верхушке магазина находится операционный символ). Операция ВЫДАЧА(X) ДМП-процессора расширяется следующим образом:

унаследованных атрибутов извлекаются из соответствующих полей верхнего магазинного символа и используются затем при выдаче символа в выходную цепочку. При этом следует помнить, что если операционный символ появился в результате преобразования исходной атрибутной транслирующей грамматики в форму простого присваивания, то такой символ в выходную цепочку не выдается;

значение синтезированных атрибутов вычисляются по правилам вычисления атрибутов, связанным с данным операционным символом, после чего значение каждого синтезированного атрибута помещается во все поля списка, на который указывает соответствующее поле символа из верхушки магазина.

M(X, a) = (β, y) (в верхушке магазина находится нетерминал). В этом случае L-атрибутный ДМП-процессор вталкивает в магазин цепочку символов β из правой части распознаваемого правила B DSLFTN WTGJXRE операционных символов y. При этом вычисляются атрибуты операционных символов, которые не вталкиваются в магазин, и заполняются поля атрибутов магазинных символов и символов цепочки β, вталкиваемых в магазин.

Источниками атрибутных правил, связанных с правилами вывода L-атрибутной транслирующей грамматики в форме простого присваивания, могут быть только константы, унаследованные атрибуты нетерминалов из левой части правил вывода, атрибуты входных и операционных символов, синтезированные атрибуты нетерминалов из правой части правил вывода. В табл. 1.2 приведены

значения источников копирующих правил в момент времени, когда верхним символом магазина является нетерминалы унаследованные атрибуты символов из правой части правил вывода и унаследованные атрибуты символов из правой части правил вывода. В табл. 1.3 приведены поля магазина, соответствующие приемникам атрибутных правил, которые необходимо заполнить во время перехода L-атрибутного ДМП-процессора при M{X, a) = (β, y).

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

Вычисляет значения синтезированных атрибутов операционных символов, которые не вталкиваются в магазин, и выдает цепочку операционных символов с их атрибутами в выходную цепочку, если эти символы не появились в результате преобразования грамматики в форму простого присваивания.

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

Если источник копирующего правила недоступен, то после вталкивания символа в магазин в соответствующие поля символа заносятся указатели на список полей, где будут храниться значения унаследованных атрибутов.

Действия L-атрибутного ДМП-процессора для элементов управляющей таблицы, имеющих значения ДОПУСК и ОШИБКА, остаются теми же самыми, что у ДМП-процессора.

Построим L-атрибутный ДМП-процессор для L-атрибутной транслирующей грамматики в форме простого присваивания.

На рис. 1.4 показано представление полей магазинных символов, а в табл. 1.5 приведена управляющая таблица для транслирующей грамматики, полученной из исходной L-атрибутной транслирующей грамматики путем вычеркивания из нее атрибутов.

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

S	i:=E{:=}, ε				
E	iR, ε				
R			+ i{+} R, ε	*i{*}	ε, ε
/	ВЫБРОСС				
:=		ВЫБРОС			
+			ВЫБРОС		
*				ВЫБР	
⊥					ДОПУСК
{:=}	ВЫДАЧА({:=},p, q)				
{+}	ВЫДАЧА({+},p, q. г)				
{*}	ВЫДАЧА({*},p, q. г)				

Начальное содержимое магазина - S⊥

7.4. Атрибутный перевод методом рекурсивного спуска

Сначала рассмотрим, каким образом можно изменить процедуры для распознавания цепочек, порождаемых нетерминальными символами грамматики, для реализации перевода, описываемого транслирующей грамматикой цепочечного перевода.

В этом случае правила составления процедур дополняются следующим правилом: если текущим символом правой части правила вывода грамматики является операционный символ Y, ему соответствует вызов процедуры записи операционного символа в выходную цепочку output (Y).

В качестве примера реализации перевода с использованием метода рекурсивного спуска рассмотрим транслирующую грамматику, описывающую перевод инфиксных арифметических выражений в польскую инверсную запись. Эта грамматика построена на основе входной грамматики G1 и имеет следующие правила:

$S \rightarrow E\perp$
$E \rightarrow TE'$
$E' \rightarrow +T\{+\}\ E' \mid \varepsilon$
$T \rightarrow PT'$
$T' \rightarrow *P\{*\}\ T' \mid \varepsilon$
$P \rightarrow (E) \mid i\ \{i\}$

+++ **Кр:** реализовать перевод для транслирующих грамматик методом рекурсивного спуска

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 7.1.1

```
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).

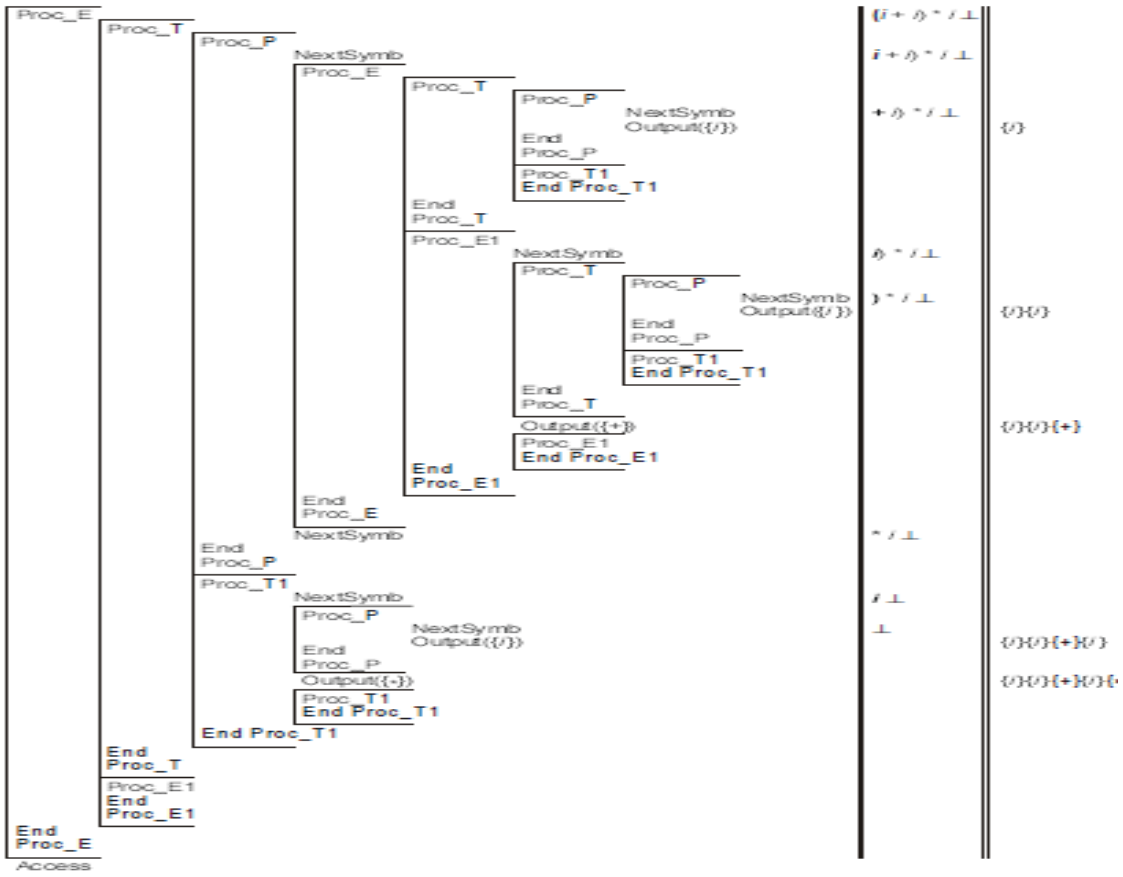


Рисунок 1.2. Порядок вызова процедур при переводе цепочки (i + i) * i ⊥Для того, чтобы методом рекурсивного спуска можно было выполнять L-атрибутный перевод, введем в процедуры распознавания нетерминальных символов параметр для каждого атрибута этого символа. При этом если параметру процедуры соответствует 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 С имеется единственный механизм передачи параметров — "вызов по значению" [36]. Эффект вызова по ссылке в языке С может быть получен, если использовать в качестве параметров указатели.

Поскольку имена атрибутов нетерминальных символов используются в качестве параметров процедур для распознавания этих нетерминалов, при именовании атрибутов необходимо выполнять дополнительное требование, заключающееся в том, что все вхождения некоторого нетерминала в левые части правил вывода АТ-грамматики должны иметь один и тот же список имен атрибутов. Например, в грамматике не может быть таких правил вывода:

$R_{p_1,\ t_2} \rightarrow +\, i_{q_1}\ \{^*\}_{p_2,\ q_2,\ r_1}\ R_{r_1,\ t_1}$

$R_{p_1,\ t_2} \rightarrow +\, i_{q_1}\ \ \{^*\}_{p_2,\ q_2,\ r_1}\ R_{r_1,\ t_1}$

$R_{p_1,\ p_2} \rightarrow \varepsilon$

т. к. первые два вхождения нетерминала R имеют атрибуты $p_1\ t_2$, а последнее вхождение — p_1, p_2 В этом случае необходимо выбрать какой-то один список имен атрибутов нетерминала R (например, p и t), использовать эти имена при описании типа атрибутов этого нетерминала (например, унаследованный p, синтезированный t) и переименовать его атрибуты соответствующим образом.

Указанные ограничения на имена атрибутов не распространяются на атрибуты символов из правых частей правил вывода грамматики.

После того как атрибуты в левых частях правил и в описании их типов переименованы, для упрощения или исключения некоторых правил вычисления атрибутов можно использовать новое соглашение об обозначениях атрибутов, которое формулируется следующим образом: "Если два атрибута получают одно и то же значение, то им можно дать одно и то же имя при условии, что для этого не нужно изменять имена атрибутов нетерминала в левой части правила вывода".

Рассмотрим несколько примеров.

$A \rightarrow B_x\ C_y\ D_z$

y, z←x

Атрибутам x, y и z присваивается одно и то же значение, поэтому им можно дать общее имя. Используя новое имя a, получим правило вывода грамматики ($A \rightarrow B_x\ C_y\ D_z$), которое не требует правила вычисления атрибута x.

$A_x \rightarrow B_y\ \{f)_z$

y, z←x

Атрибутам можно дать одно имя, но оно должно быть x, для того чтобы не изменилось имя атрибута в левой части правила. Выполнив замену имен, получим правило вывода грамматики $A_x \rightarrow B_x\ \{f\}^*$, при этом отпадает необходимость в атрибутном правиле.

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

y, z, t ←x

Атрибуты y, z, t, x имеют одно и то же значение, но им нельзя дать одно имя, поскольку нетерминал в левой части правила вывода имеет два атрибута: x и y. В данном случае можно получить лишь частичное упрощение:

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

$y \leftarrow x$

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;

1. обработка операционного символа. Для каждого вхождения операционного символа в правую часть правила вывода грамматики в процедуру включается фрагмент кода, который по соответствующим атрибутным правилам вычисляет значения синтезированных атрибутов операционного символа и присваивает вычисленные значения переменным, соответствующим синтезированным атрибутам. Затем вызывается процедура выдачи операционного символа вместе с атрибутами в выходную строку;
2. обработка правил вычисления атрибутов. Для каждого правила вычисления атрибутов, сопоставленного правилу вывода грамматики, в процедуру включается фрагмент кода, который вычисляет значение атрибута и присваивает это значение каждой переменной из левой части атрибутного правила (если соглашение об одинаковых именах выполнено, то в левой части атрибутного правила будет только один атрибут). Фрагмент кода можно поместить в любом месте процедуры, которое находится:
   - после точки, где используемые в правиле атрибуты уже вычислены;
   - перед точкой, где впервые используется вычисленное значение атрибута;
   - головной модуль. Все имена синтезированных атрибутов начального символа грамматики становятся локальными переменными головного модуля. Список фактических параметров вызова процедуры распознавания начального символа грамматики содержит начальные значения унаследованных атрибутов и имена синтезированных атрибутов.

Пример 7.1. Рассмотрим L-атрибутную транслирующую грамматику в качестве примера реализации атрибутного перевода с использованием метода рекурсивного спуска.

Для повышения наглядности переименуем атрибуты символов грамматики таким образом, чтобы всем вхождениям символов в правые части разных правил вывода соответствовали разные имена атрибутов, а также выберем одинаковые имена для атрибутов нетерминала R из левой части правил вывода с номерами (3), (4) и (5). Пусть p — унаследованный атрибут нетерминала R, а t — его синтезированный атрибут. После преобразования получим следующую грамматику:

Et синтезированный t

Rp, t унаследованный p синтезированный t

Атрибуты операционных символов унаследованные.

(0) $S_0 \rightarrow S \perp$

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

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

(3) $R_{p,\ t} \rightarrow I_{q1}$ {+}p, q_1, r_1 $R_{r1,\ t}$
r1 ← GETNEW

(4) $R_{p,\ t} \rightarrow X\ I_{q2}$ {*}p, q_1, , r_1 $R_{r1,\ t}$
r2 ← GETNEW

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

++ **Кр:** Атрибутный перевод операторов присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ,

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 7.2.2.

Листинг 7.2.2

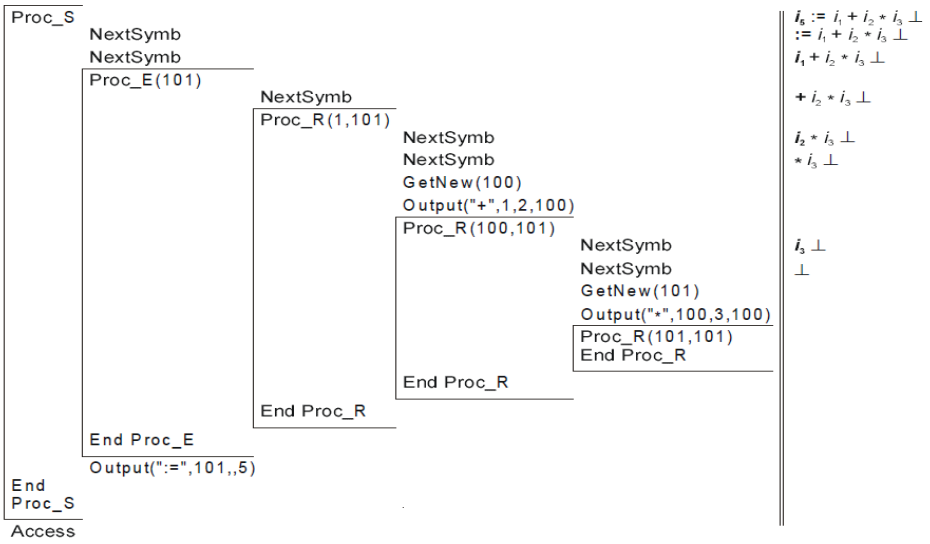
```
Procedure  Recurs_Method_ATG (List_Token: tList, List_Tetr: tListTetr);
{List_Token -входная цепочка, List_Tetr - цепочка тетрад}

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 $i_5 = i_1 + i_2 * i_3$ ⊥ в цепочку тетрад. Справа в строке, соответствующей вызову процедуры NextSymb, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после вызова этой процедуры.

S-атрибутный ДМП-процессор



КР Математическая модель **восходящего** ДМП-процессора

Для любого восходящего синтаксического анализатора, последовательность операций **переноса и свертки**, выполняемых при обработке допустимых входных цепочек, можно описать с помощью транслирующей грамматики.

+++ end

Входной для этой транслирующей грамматики является грамматика, на основе которой построен анализатор. Для получения транслирующей грамматики в самую крайнюю правую позицию каждого i-го правила входной грамматики вставляется операционный символ {СВЕРТКА, i}. Например, транслирующая грамматика, построенная по входной грамматике G_q = ({E, T\ P), {i, +, *, (,)}, P, E), где P = {E → E + T, E→T, T→ T* P, P, P →i, P → (E)}, будет выглядеть следующим образом:

E → E + T{СВЕРТКА, 1}

E →T{СВЕРТКА, 2}

T→ T* P {СВЕРТКА, 3}

T→ P {СВЕРТКА, 4}

P → i {СВЕРТКА, 5}

P → (E) {СВЕРТКА, 6}

Если каждый входной символ в активной цепочке интерпретировать как представление операции ПЕРЕНОС, выполняемой в момент времени, когда этот символ является текущим входным символом, то активная цепочка в точности описывает последовательность операций переноса и свертки, выполняемых при обработке входной цепочки. 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 {СВЕРТКА_6} {СВЕРТКА 3} {СВЕРТКА_ 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 З - выдачу операционного символа {*}, а СВЕРТКА 6 — выдачу операционного символа {i}.

Определение. Восходящий ДМП-процессор - это синтаксический анализатор, дополненный формальными действиями по выполнению перевода.

7.5. Реализация S-атрибутного ДМП-процессора

КР (есть проба) Восходящий ДМП-процессор преобразуется в S-атрибутный ДМП-процессор, реализующий атрибутный перевод, определяемый постфиксной S-атрибутной транслирующей грамматикой.

В S-атрибутном ДМП-процессоре каждый магазинный символ - объект для представления атрибутов.

!!!заменить на объект

Аналогично L-атрибутному ДМП-процессору, примем, 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, предназначенные для атрибутов, заполняются значениями атрибутов в момент вталкивания символа в магазин и не изменяются до момента выталкивания его из магазина.

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

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

Расширенная операция свертки использует эти значения для вычисления значений всех атрибутов операционных символов, связанных с правилом вывода транслирующей грамматики, и значений всех атрибутов нетерминала из

левой части правила. Значения атрибутов операционных символов используются для выдачи результатов в выходную ленту или выполнения других действий, определяемых этими символами. Атрибуты нетерминала из левой части правила вывода записываются в соответствующие поля магазинного символа, который соответствует этому нетерминалу и вталкивается в магазин во время свертки.

На рис. 6.6 изображена последовательность состояний магазина S-атрибутного ДМП-процессора, осуществляющего перевод Цепочки $i_2 + i_3 \times i_5$ в последовательность тетради со знаками операций СЛОЖИТЬ и УМНОЖИТЬ. S- атрибутная транслирующая Грамматика, входной грамматикой которой является основная грамматика, имеет вид:

E_p- СИНТЕЗИРОВАННЫЙ p

(1)$E_{p_2} \rightarrow E_{q_1} + E_{t_1} \{+\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

$p_1, p_2 \leftarrow$ GETNEW

(3)$E_{p_2} \rightarrow E_{q_1} * E_{t_1} \{*\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

$p_1, p_2 \leftarrow$ GETNEW

(5)$E_{p_2} \rightarrow p_1$

$p_2 \leftarrow p_1$

(6)$E_{p_2} \rightarrow (E_{p_1})$

$p_2 \leftarrow p_1$

На рис. 6.6, а приведено начальное состояние магазина. Рис. 6.6, б представляет результат операции ПЕРЕНОС входного символа с атрибутом 2 в магазин, а рис. 6.6, в результат выполнения операции СВЕРТКА, выполняемой в соответствии с правилом (5). Рис. 6.6, г и д иллюстрируют результат выполнения операции ПЕРЕНОС символов '+' и i_3 соответственно из входной строки в магазин, а рис. 6.6, е - результат выполнения операции СВЕРТКА символа i_3 в E_3. Аналогично выполняются операции переноса и свёртки для символов '*' и i_5 (рис. 6.6, ж-е).

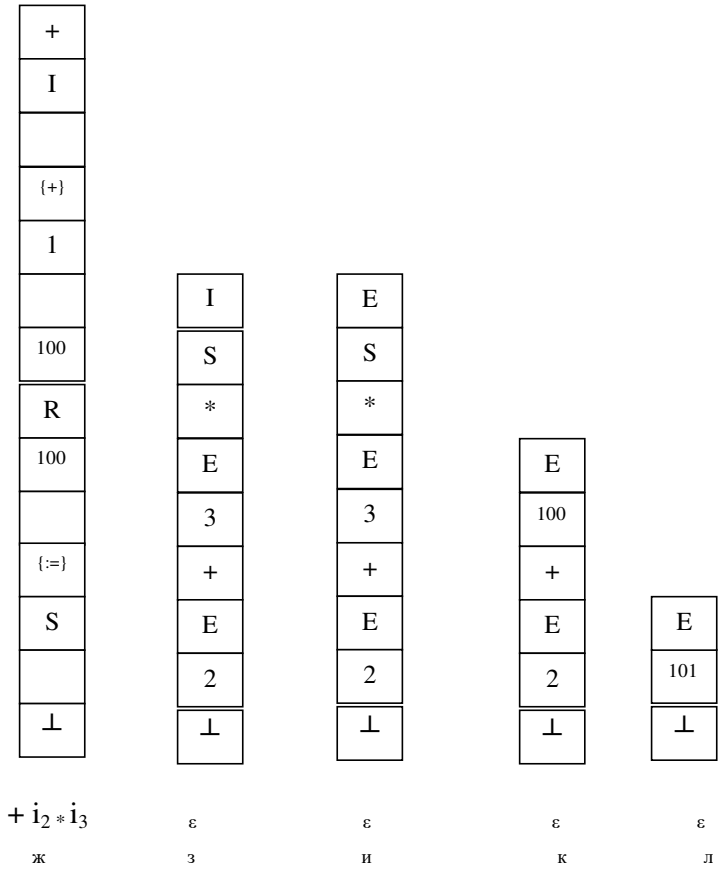


Рис. 6.6. Последовательность состояний магазина при обработке входной цепочки $i_2 + i_3 * i_5$

Рис. 6.6, и иллюстрирует результат выполнения операции СВЕРТКА, выполняемой в соответствии с правилом (1). При этом поля атрибутных символов из правой Части этого правила используются для формирования тетрады,

*	3	5	100

которая выдаётся в выходную цепочку.

Далее выполняется операция СВЕРТКА по правилу (3) с выдачей тетрады:

+	2	100	101

(рис. 6.6, к-л).

Контрольные вопросы

1. Дайте определение L-атрибутной и S-атрибутной транслирующих грамматик.
2. Какое правило вычисления атрибутов называется копирующим?
3. В каком случае АТ-грамматика имеет форму простого присваивания? Приведите процедуру преобразования произвольной АТ-грамматики в форму простого присваивания.

4. Приведите алгоритм построения управляющей таблицы для транслирующей грамматики Цепочечного перевода, входной грамматикой которой является LL(!)-Грамматика.
5. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов Грамматики в L-атрибутном ДМП-процессоре.
6. Приведите процедуру преобразования ДМП-процессора в L-атрибутный ДМП-процессор.
7. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый транслирующей грамматикой?
8. С какой целью осуществляется переименование имён атрибутов нетерминальных символов из левых частей правил вывода L-атрибутной транслирующей грамматики при реализации вывода L-атрибутного перевода методом рекурсивного спуска?
9. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый L-атрибутной транслирующей Грамматикой?
10. Каким образом восходящий анализатор можно расширить действиями по выполнению перевода?
11. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов грамматики в S-атрибутном ДМП-процессоре.

Упражнения

Следующие правила вывода грамматики являются частью некоторой атрибутной грамматики. Атрибуты p, q и r – унаследованные, а s и f – синтезированные. Для каждого правила определите, от каких атрибутов могут зависеть правила вычисления p и r, чтобы это правило было L- атрибутным?

1.1. $A_{s,\,q} \rightarrow a_{и}\ A_{t,\,p}\ B_{r}$.

1.2. $A_{s,\,q} \rightarrow \{c\}_{p}\ b_{и}\ A_{t,\,r}$.

1.3. $A_{s,\,q} \rightarrow c_{и}\ B_{r}\ A_{t,\,p}$.

Приведите следующие правила вывода АТ-Грамматики к форме простого присваивания (имена унаследованных атрибутов начинаются с символа i, а имена синтезированных атрибутов – с символа s):

2.1. $A_{s_1},\ i_1 \rightarrow E$

$s_1 \leftarrow \sin(i_1)$

2.2. $E_{s_1},\ i_1 \rightarrow A_{s_2}\ B_{s_3},\ i_2\ C_{s_4},\ i_3\ D_{s_5},\ i_4$

$i_2 \leftarrow i_1$

$i_3 \leftarrow i_1 * i_1$

$i_4 \leftarrow i_2 * i_3$

$s_1 \leftarrow s_2 * s_3$

Преобразуйте L-атрибутную грамматику цепочечного перевода к форме простого присваивания и постройте для нее L-атрибутный

ДМП- процессор.

A_p СИНТЕЗИРОВАННЫЙ p

B_p СИНТЕЗИРОВАННЫЙ p

$\{b\}_p$ УНАСЛЕДОВАННЫЙ p

1. $S \rightarrow a_p\ A_q\ B_r\ A_s\ \{b\}t$

3. $A_p \rightarrow$ Ь r ← p + 2q

p ← Ь

t ← r + s

2. $A_p \rightarrow a_q\ A_r\ B_s\ B_t\ B_u\ S$

4. $B_p \rightarrow a_q\ A_r\ \{b\}_t$

s ← q + r

t ← r + q

t ← q * s - 4

u, p ← r + t

Для Грамматики, заданной в упр. 2, постройте процессор методом рекурсивного спуска.

Для входной грамматики, описывающей синтаксис арифметических выражений, с правилами вывода

E → (E + E)

E → (E * E)

E → i

Где i – лексема, значением которой является указатель на элемент таблицы идентификаторов, определите постфиксную S-атрибутную грамматику, описывающую перевод этих выражений в цепочку тетрад с кодами операций: СЛОЖИТЬ и УМНОЖИТЬ, и постройте S-атрибутный ДМП-процессор, выполняющий заданный перевод.

Библиографический список

1.*Ахо А., Ульман Д.* Теория синтаксического анализа перевода и компиляции // Пер. с англ. - М.: Мир, 1978.

2.Ульман, Джеффри, Д. Компиляторы: Принципы, технологии, инструменты. Пер. с англ. — М.: Издательский дом "Вильяме", 2008 и 2003.

3.*Р.Хантер.* Проектирование и конструирование компиляторов. Пер. с англ. – М.: Финансы и статистика, 1984.

4.*Н.Вирт.* Алгоритмы + структуры данных = программы. Пер. с англ. – М.: МИР, 1985.

5. Д. Грис. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.

6. Ф. Льюис, Д. Розенкранц, Р. Стирнз. Теоретические основы проектирования компиляторов. — М.: Мир, 1979. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. — Т.1,2. — М.: Мир, 1979.

7. Ф. Вайнгартен. Трансляция языков программирования. — М.: Мир, 1977.

8. А. Ахо., Р. Сети, Дж. Ульман. Компиляторы: принципы, технологии, инструменты. — М.: «Вильямс», 2001.
9. А. Ахо, М. Лам, Р.Сети, Дж. Ульман. Компиляторы: принципы, технологии и инструментарий. — М.: «Вильямс», 2008.
10. О Опалева Э. А., Самойленко В.П. Языки программирования и методы трансляции. - СПб.: БХВ- Петербург, 2005.
11. *А А.С.Семенов.* Построение класса фрактальных систем по шаблону на примере дерева Фибоначчи // РАН. Информационные технологии и Вычислительные системы. – М.: N2, 2005 стр.10-17.