Глава 3 Контекстно-свободные языки

3.1 Свойства и распознаватели КС-языков

3.1.1 Автоматы с магазинной памятью – распознаватели КС-языков

Распознаватели, определяющие КС-языки, моделируются автоматами с магазинной памятью (МПА). Дадим строгое определение такого автомата.

Автомат с магазинной памятью (МПА) – это семерка $P=(Q,V,Z,\delta,q_0,z_0,F)$, где

- Q – множество состояний УУ автомата;
- V – конечный входной алфавит (множество допустимых символов);
- Z – специальный конечный алфавит магазинных символов автомата (обычно в него входят терминальные и нетерминальные символы грамматики, но могут использоваться и другие символы), $V \subseteq Z$;
- δ – функция переходов, отображающая множество $Q\times(V\cup\{\lambda\})\times Z$ на конечное множество подмножеств множества $(Q\times Z^*)$;
- q_0 – начальное состояние автомата: $q_0 \in Q$;
- z_0 – начальный символ магазина: $z_0 \in Z$;
- F – множество конечных состояний автомата: $F \subseteq Q$, $F \neq \varnothing$.

В отличие от конечного автомата (КА), рассмотренного в предыдущей главе, МП-автомат имеет «стек» магазинных символов, который играет роль дополнительной, или внешней памяти. Переход МПА из одного состояния в другое зависит не только от входного символа и текущего состояния, но и от содержимого стека. Таким образом, конфигурация МПА определяется уже тремя параметрами: состоянием автомата, текущим символом входной цепочки и содержимым стека.

Итак, *конфигурация*, или *мгновенное описание (МО)* МПА – это тройка $(q,w,\alpha)\in Q\times V^*\times Z^*$, где q – текущее состояние УУ, w – непрочитанная часть входной цепочки, α – содержимое магазина. Если $w=\lambda$, считается, что цепочка прочитана; если $\alpha=\lambda$, то магазин считается пустым.

Начальная конфигурация МПА определяется как (q_0,w,z_0), $w\in V^*$. Множество *конечных конфигураций* – как (q,λ,z), где $q\in F$, $z\in Z^*$.

Такт работы МП-автомата будем обозначать отношением $\vdash$ на множестве конфигураций и описывать в виде $(q,aw,t\alpha)\vdash(q',w,\gamma\alpha)$, если $(q',\gamma)\in\delta(q,a,t)$, где $q,q'\in Q$, $a\in V\cup\{\lambda\}$, $w\in V^*$, $t\in Z$, $\gamma,\alpha\in Z^*$. При выполнении такта автомат, находясь в состоянии q, считывает символ входной цепочки ‘a’ и сдвигает входную головку на одну ячейку вправо, а из магазина

удаляет верхний символ, соответствующий условию перехода, и заменяет цепочкой согласно правилу перехода. Первый символ цепочки становится вершиной стека. Состояние автомата изменяется на q′. Допускаются переходы, при которых считывающая головка не сдвигается и входной символ игнорируется, тогда он становится входным символом при следующем такте, но состояние УУ и содержимое стека может измениться. Такие переходы называются *λ–тактами*. Автомат может проделывать λ–такты, когда уже прочёл входную цепочку или же в процессе её прочтения; но, если магазин пуст, следующий такт невозможен.

Итак, находясь в состоянии q и наблюдая символ входной ленты x, МПА на одном такте работы может проделать со стеком в зависимости от правил перехода одно из следующих действий:

1) Удалить из стека верхний символ α: $\delta(q,x,\alpha)=\{(q,\lambda)\}$.
2) Оставить содержимое стека без изменений: $\delta(q,x,\alpha)=\{(q,\alpha)\}$.
3) Дописать в стек с верхним символом α символ x: $\delta(q,x,\alpha)=\{(q,x\alpha)\}$. В общем случае может быть дописан другой символ, отличный от прочитанного, или даже цепочка символов, например $\delta(q,x,\alpha)=\{(q,y\alpha)\}$, где $y\in Z^+$. В некоторых случаях это бывает очень удобно.
4) Заменить верхний символ стека α символом или цепочкой y: $\delta(q,x,\alpha)=\{(q,y)\}$, где $y\in Z^+$.

МПА *допускает цепочку символов w*, если, получив эту цепочку на вход, он может перейти в одну из конечных конфигураций, когда по окончании цепочки автомат находится в одном из конечных состояний: $(q_0,w,z_0) \vdash^* (q,\lambda,z)$, где $q\in F$, $z\in Z^*$. После окончания прочтения цепочки автомат может проделать некоторое количество λ–тактов.

Язык, определяемый МП-автоматом P – это множество всех цепочек символов, допускаемых этим автоматом:
$L(P) = \{w \mid \exists q\in F \mid (q_0,w,z_0) \vdash^* (q,\lambda,z)$, где $z\in Z^* \}$.

Два автомата P_1 и P_2 *эквивалентны*, если они определяют один и тот же язык: $L(P_1)=L(P_2)$.

Говорят, что МПА *допускает цепочку символов с опустошением магазина*, если при окончании разбора цепочки автомат находится в одном из своих конечных состояний, а стек пуст, т.е. получена конфигурация (q,λ,λ), $q\in F$.

Язык, заданный автоматом P, допускающим цепочки с опустошением стека, обозначается как L_λ (P). Для любого МП-автомата всегда можно построить эквивалентный ему МПА, допускающий цепочки с опустошением стека: $\forall$ МПА P $\exists$ МПА P′ $\mid$ $L(P)=L_\lambda(P')$.

Кроме обычного МПА, существует понятие расширенного МПА. *Расширенный МПА* может заменять не один символ в вершине стека, а цепочку символов конечной длины, на некоторую другую цепочку

символов. Для любого расширенного МПА всегда можно построить эквивалентный ему обычный МП-автомат. Следовательно, классы МПА и расширенных МПА эквивалентны.

Пример. Построим МПА с опустошением стека, определяющий язык $L=\{0^n1^n \mid n\geq 0\}$ – множество цепочек, в которых сначала подряд стоит некоторое количество нулей, а затем так же подряд столько же единиц.

Работа данного МП-автомата P должна состоять в том, что он копирует в магазин начальную часть входной цепочки, состоящую из нулей, а затем (как только на входе начнут появляться единицы) устраняет из магазина по одному нулю на каждую прочитанную единицу. Если нули в магазине и единицы на входе закончились одновременно, это означает, что их количества равны. Заметим, что в общем случае символы магазина могут отличаться от символов входной цепочки – например, на каждый прочитанный на входе 0 в магазин может записываться символ ‘a’. Построим этот МПА.

$P=(\{q_0,q_1\},\{0,1\},\{Z,0\},\delta,q_0,Z,\{q_0\})$, где функция переходов имеет вид:

$\delta(q_0,0,Z)=\{(q_0,0Z)\}$ $\delta(q_0,1,0)=\{(q_1,\lambda)\}$ $\delta(q_0,\lambda,Z)=\{(q_0,\lambda)\}$

$\delta(q_0,0,0)=\{(q_0,00)\}$ $\delta(q_1,1,0)=\{(q_1,\lambda)\}$ $\delta(q_1,\lambda,Z)=\{(q_0,\lambda)\}$

Следует отметить, что при появлении во входной цепочке первой единицы после последовательности нулей состояние МПА должно измениться для того, чтобы исключить возможность прочтения нулей вперемешку с единицами. Т.е. одно состояние – то, в котором читаются все нули и записываются в стек (q_0), другое (q_1)– при котором эти нули из стека удаляются.

В качестве конечного состояния можно взять новое состояние q_2, а можно использовать начальное q_0. Правило $\delta(q_0,\lambda,Z)=\{(q_0,\lambda)\}$ необходимо для того, чтобы автомат мог принимать пустую цепочку, которая существует в языке.

Пусть входная цепочка имеет вид w=’0011’. Тогда смена конфигураций выглядит следующим образом:

$(q_0,0011,Z) \vdash (q_0,011,0Z) \vdash (q_0,11,00Z) \vdash (q_1,1,0Z) \vdash (q_1,\lambda,Z) \vdash (q_0,\lambda,\lambda)$.

Цепочка допущена заданным автоматом.

Если рассмотреть цепочку, не относящуюся к данному языку, например, α=’001101’, то автомат проделает следующую последовательность действий:

$(q_0,001101,Z) \vdash (q_0,01101,0Z) \vdash (q_0,1101,00Z) \vdash (q_1,101,0Z) \vdash (q_1,01,Z)$ $\vdash (q_0,01,\lambda)$. На последнем шаге автомат проделал λ-такт, в результате чего пришел в конечное состояние и опустошил стек, но цепочка осталась недочитанной. Далее автомат работать не может, следовательно, цепочка не принимается.

Утверждение

1. Для произвольной КС-грамматики всегда можно построить МП-автомат, распознающий задаваемый этой грамматикой язык.
2. Для произвольного МП-автомата всегда можно построить КС-грамматику, которая будет задавать язык, распознаваемый этим автоматом.

МПА называется *недетерминированным*, если из одной и той же его конфигурации возможен более чем один следующий переход.

МПА называется *детерминированным*, если из любой его конфигурации возможно не более одного следующего такта. Класс ДМП-автоматов и соответствующих языков заметно уже, чем весь класс КС-языков и соответственно, МП-автоматов. В отличие от КА, не для каждого МПА возможно построить эквивалентный ему ДМП-автомат.

ДМП-автоматы определяют особый подкласс среди КС-языков, называемый *детерминированными КС-языками*. Все языки, относящиеся к этому классу, могут быть построены с помощью однозначных КС-грамматик, следовательно, они играют особую роль в теории языков программирования. Поэтому ДМПА необходимы при создании компиляторов. На основе ДМПА может быть построен синтаксический распознаватель любого языка программирования.

3.1.2 Свойства КС-языков

Основное свойство КС-языков: Класс КС-языков замкнут относительно операции подстановки. Это означает, что, если в каждую цепочку символов КС-языка вместо некоторого символа подставить цепочку символов из другого КС-языка, то полученная цепочка также будет принадлежать КС-языку.

Класс КС-языков замкнут относительно операций объединения, конкатенации, итерации, изменения имен символов.

Замечание:

Класс КС-языков не замкнут относительно операции пересечения и операции дополнения.

Пример:

Пусть есть два языка $L_1=\{a^n b^n c^i \mid n>0,\ i>0\}$ и $L_2=\{a^i b^n c^n \mid n>0,\ i>0\}$. Оба эти языка являются КС, но, если взять их пересечение, то получим язык $L=L_1 \cap L_2=\{a^n b^n c^n \mid n>0\}$, который уже не является КС-языком.

Для КС-языков разрешимы проблемы пустоты языка и принадлежности заданной цепочки языку – для их решения достаточно построить МПА, распознающий данный язык. Но проблема эквивалентности двух произвольных грамматик является неразрешимой. Неразрешима и более узкая проблема – эквивалентности заданной произвольной КС-грамматики и произвольной регулярной грамматики. Тем не менее, для некоторых КС-грамматик можно построить

эквивалентную им однозначную грамматику.

Детерминированные КС-языки представляют собой более узкий класс в семействе КС-языков. Этот класс не замкнут относительно операций объединения и пересечения, хотя, в отличие от всех КС-языков в целом, замкнут относительно операции дополнения.

Для класса детерминированных КС-языков разрешима проблема однозначности. Доказано, что, если язык может быть распознан с помощью ДМПА, он может быть описан с помощью однозначной КС-грамматики. Поэтому данный класс используется для построения синтаксических конструкций языков программирования.

Как и в случае с регулярными языками, для проверки того факта, что некоторый язык не принадлежит классу КС-языков, служит лемма о разрастании КС-языков. Она выполняется для любого КС-языка. Если лемма не выполняется, то это означает, что язык не относится к типу контекстно-свободных.

Лемма о разрастании КС-языков:

Если взять достаточно длинную цепочку символов, принадлежащую произвольному КС-языку, то в ней всегда можно выделить две подцепочки, длина которых в сумме больше нуля, таких, что, повторив их сколь угодно большое число раз, можно получить новую цепочку символов, принадлежащую данному языку.

Формальная запись: Если L – это КС-язык, то $\exists\ k>0$, $k\in N$ $|$если $|\alpha|\geq k$ и $\alpha\in L$, то $\alpha=\beta\delta\gamma\varphi\mu$, где $\delta\varphi\neq\lambda$, $|\delta\gamma\varphi|\leq k$ и $\beta\delta^i\gamma\varphi^i\mu\in L\ \forall\ i\geq 0$.

Пример:

Проверим, является ли язык $L=\{a^nb^nc^n\,|\,n>0\}$ КС-языком. Пусть это так, тогда для него должна выполняться лемма о разрастании КС-языков. Значит, существует некоторая константа k, о которой идет речь в этой лемме. Возьмем цепочку этого языка $\alpha=a^kb^kc^k$, $|\alpha|>k$. Если её записать в виде $\alpha=\beta\delta\gamma\varphi\mu$, то по условиям леммы $|\delta\gamma\varphi|\leq k$, следовательно, цепочка $\delta\gamma\varphi$ не может содержать вхождения всех трех символов ‘a’, ‘b’, ‘c’, т.е. в ней нет или ‘a’, или ‘c’. Рассмотрим цепочку $\beta\delta^0\gamma\varphi^0\mu=\beta\gamma\mu$. По условиям леммы, она должна принадлежать языку L, но она содержит либо k символов ‘a’, либо k символов ‘c’. Но $|\beta\gamma\mu|<3k$, следовательно, какие-то из символов языка входят в цепочку меньшее число раз, чем другие. Такая цепочка не может принадлежать заданному языку. Следовательно, язык L не является КС-языком.

3.2 Преобразование КС-грамматик

3.2.1 Цели преобразований грамматик

В общем случае для КС-грамматик невозможно проверить их однозначность и эквивалентность. Но для конкретных случаев бывает можно и нужно привести заданную грамматику к некоторому определённому виду таким образом, чтобы получить грамматику, эквивалентную исходной. Заранее определённый вид зачастую позволяет упростить работу с языком и построение распознавателей для него. Итак, преобразования грамматик могут преследовать две цели:

1) упрощение правил грамматики;
2) облегчение создания распознавателя языка.

Не всегда эти цели удается совместить, тогда необходимо исходить из того, что является главным в конкретной задаче. В теории языков программирования основой является создание компилятора для языка, поэтому главной становится вторая цель. Следовательно, можно пренебречь упрощением правил (и даже смириться с некоторым их усложнением), если при этом удастся упростить построение распознавателя языка.

Все преобразования грамматик условно разбиваются на две группы:

1. исключение из грамматики тех правил и символов, без которых она может существовать (позволяет упростить правила);
2. изменение вида и состава правил грамматики. При этом могут появиться новые правила и нетерминальные символы (упрощений правил нет).

3.2.2 Приведённые грамматики

Приведёнными (или грамматиками *в каноническом виде*) называются грамматики, которые не содержат недостижимых и бесплодных символов, циклов и пустых правил (λ-правил).

Рассмотрим некоторую грамматику G(VT,VN,P,S) и дадим необходимые определения.

Нетерминальный символ $A \in VN$ называется *бесплодным* (или *бесполезным*), если из него нельзя вывести ни одной цепочки терминальных символов, т.е. $\{\alpha \mid A \Rightarrow^* \alpha,\ \alpha \in VT^*\} = \varnothing$.

В простейшем случае символ является бесплодным, если во всех правилах, где он находится в левой части, он встречается также и в правой части. В более сложных случаях бесполезные символы могут находиться в некоторой взаимной зависимости, порождая друг друга. Если из правил грамматики удалить такие символы, то эти правила станут проще.

Символ $x \in (VT \cup VN)$ называется *недостижимым*, если он не встречается ни в одной сентенциальной форме грамматики G. Это значит, что он не может появиться ни в одной цепочке вывода. Для исключения всех недостижимых символов не обязательно рассматривать все сентенциальные формы грамматики, достаточно воспользоваться специальным алгоритмом удаления недостижимых символов. После удаления таких символов правила также упрощаются.

λ-правилами, или правилами с пустой цепочкой, называются все правила грамматики вида $A \to \lambda$, $A \in VN$. Грамматика G называется *грамматикой без λ-правил*, если в ней нет правил вида $(A \to \lambda)$, $A \in VN$, $A \neq S$, и существует только одно правило $(S \to \lambda) \in P$, если $\lambda \in L(G)$ и при этом S не встречается в правой части ни одного правила грамматики G. Для упрощения процесса построения распознавателя цепочек языка L(G) любую грамматику целесообразно привести к виду без λ-правил.

Циклом в грамматике G называется вывод вида $A \Rightarrow^* A$, $A \in VN$. Очевидно, что такой вывод бесполезен, поэтому в распознавателях КС-языков рекомендуется избегать возможности появления циклов.

Циклы возможны в случае существования в грамматике *цепных правил* вида $A \to B$, $A,B \in VN$. Достаточно устранить цепные правила из набора правил грамматики, чтобы исключить возможность появления циклов.

Для того чтобы преобразовать произвольную КС-грамматику к каноническому виду, необходимо выполнить следующие действия (причём именно в том порядке, каком они перечислены):

- удалить все бесплодные символы;
- удалить все недостижимые символы;
- удалить λ-правила;
- удалить цепные правила.

Для каждого из названных действий существует свой алгоритм. Рассмотрим эти алгоритмы в том порядке, каком требуется их реализовывать.

1 Удаление бесплодных символов

Выполняется пошаговое построение множества Y_i, которое не содержит бесплодных символов рассматриваемой грамматики. На каждом шаге к уже построенному множеству добавляются новые нетерминальные символы. При этом первоначально в него включают те символы, из которых могут быть выведены терминальные цепочки, а на последующих шагах – символы, из которых могут быть выведены цепочки, состоящие как из терминальных символов, так и из символов уже построенного множества. После того, как с некоторого шага множество Y_i перестанет изменяться, процесс построения считается законченным. В итоговое множество нетерминальных символов должны входить только символы из

построенного множества Y_i.

1. $Y_0=\varnothing$, i:=1.
2. $Y_i=Y_{i-1}\cup\{A\in VN \mid \exists(A\to\alpha)\in P, \alpha\in(Y_{i-1}\cup VT)^*\}$.
3. До тех пор, пока не станет $Y_i=Y_{i-1}$, выполняется i:=i+1 и снова шаг 2.
4. Новая грамматика: множество терминальных символов VT′ совпадает со старым VT: VT′=VT, VN′=Y_i, S′=S, а в P′ входят те правила из P, которые содержат только символы из множества ($Y_i\cup$VT).

2 Удаление недостижимых символов

Данный алгоритм строит множество достижимых символов V_i исходной грамматики. Сначала в это множество входит только целевой символ S грамматики G, затем оно пополняется на основе правил этой грамматики. Множество считается построенным, если в него невозможно добавить ничего нового. Все символы, не вошедшие в это множество, являются недостижимыми, следовательно, их требуется исключить из словаря и из правил грамматики.

1. $V_0=\{S\}$, i:=1.
2. $V_i=V_{i-1}\cup\{\boldsymbol{x} \mid \boldsymbol{x}\in(VN\cup VT)$ и $(A\to\alpha\boldsymbol{x}\beta)\in P, A\in V_{i-1}\cap VN, \alpha,\beta\in(VN\cup VT)^*\}$.
3. До тех пор, пока не станет $V_i=V_{i-1}$, выполняется i:=i+1 и снова шаг 2.
4. Новая грамматика: множество терминальных символов VT′=VT∩V_i, множество нетерминальных символов VN′= VN∩V_i, S′=S, а в P′ входят те правила из P, которые содержат только символы из множества V_i.

Замечание:

Оба рассмотренных алгоритма – удаления бесплодных и недостижимых символов – относятся к первой группе алгоритмов, т.е. их применение приводит к упрощению грамматики, сокращению количества правил и уменьшению объема алфавита.

Пример работы рассмотренных алгоритмов:

Пусть дана грамматика G({a,b,c},{A,B,C,D,E,F,G,S},P,S), где P:

$S\to aAB \mid E$	$D\to a \mid c \mid Fb$
$A\to aA \mid bB$	$E\to cE \mid aE \mid Eb \mid ED \mid FG$
$B\to ACb \mid b$	$F\to BC \mid EC \mid AC$
$C\to A \mid bA \mid cC \mid aE$	$G\to Ga \mid Gb$.

Удалим бесплодные символы.

1. $Y_0=\varnothing$, i:=1.
2. $Y_1=\{B,D\}$, $Y_1\neq Y_0$, i:=2.
3. $Y_2=\{B,D,A\}$, $Y_2\neq Y_1$, i:=3.
4. $Y_3=\{B,D,A,S,C\}$, $Y_3\neq Y_2$, i:=4.
5. $Y_4=\{B,D,A,S,C,F\}$, $Y_4\neq Y_3$, i:=5.
6. $Y_5=\{B,D,A,S,C,F\}$, $Y_5=Y_4$.

7. Бесплодными символами оказались символы, не вошедшие в множество Y_5, т.е. E и G. Строим новую грамматику: $VT'=VT=\{a,b,c\}$, $VN'=Y_i=\{A,B,C,D,F,S\}$, $S'=S$. Получили грамматику $G'(\{a,b,c\},\{A,B,C,D,F,S\}, P', S)$,

P′:	$S\to aAB$	$C\to A \mid bA \mid cC$
	$A\to aA \mid bB$	$D\to a \mid c \mid Fb$
	$B\to ACb \mid b$	$F\to BC \mid AC$

Удалим недостижимые символы:

1. $V_0=\{S\}$, i:=1.
2. $V_1=\{S,A,B,a\}$, $V_1\neq V_0$, i:=2.
3. $V_2=\{S,A,B,a,b,C\}$, $V_2\neq V_1$, i:=3.
4. $V_3=\{S,A,B,a,b,C,c\}$, $V_3\neq V_2$, i:=4.
5. $V_4=\{S,A,B,a,b,C,c\}$, $V_4=V_3$.
6. Строим новую грамматику: множество нетерминальных символов $VN''=\{S,A,B,a,b,C,c\}\cap VN=\{A,B,C,S\}$, множество терминальных символов $VT''=\{S,A,B,a,b,C,c\}\cap VT=\{a,b,c\}$, $S''=S$, а в P′′ входят те правила из P′, которые содержат только символы из множества V_i. В итоге получаем грамматику $G''(\{a,b,c\},\{A,B,C,S\}, P'', S)$, где

P′′:

$S\to aAB$	$B\to ACb \mid b$
$A\to aA \mid bB$	$C\to A \mid bA \mid cC$

Видно, что получена значительно более простая грамматика, чем исходная.

3 Устранение λ-правил

Алгоритм преобразования грамматики к виду без λ-правил работает с некоторым множеством нетерминальных символов W_i. Это множество включает символы, из которых либо есть непосредственный вывод пустой цепочки, либо переходы в такие символы.

1. $W_0=\{A\in VN \mid \exists(A\to\lambda)\in P\}$, i:=1.
2. $W_i=W_{i-1}\cup\{A\in VN \mid \exists\,(A\to\alpha)\in P, \alpha\in W_{i-1}^*\}$.
3. До тех пор, пока не станет $W_i=W_{i-1}$, выполняется i:=i+1 и снова шаг 2.
4. Новая грамматика: $VT'=VT$, $VN'=VN$, в P′ входят все правила из P, кроме правил вида $A\to\lambda$.
5. Если $(A\to\alpha)\in P$ и в цепочке α присутствуют символы из W_i, то на основе цепочки α строится множество цепочек $\{\alpha'\}$ путём исключения из α всех возможных комбинаций символов из W_i, и все правила вида $A\to\alpha'$ добавляются в P′.
6. Если $S\in W_i$, то $\lambda\in L(G)$. Если S входит в правую часть хотя бы одного

правила, то в VN′ добавляется новый символ S′, который становится целевым символом новой грамматики, а в P′ добавляются два новых правила: $S' \to \lambda \mid S$. Иначе S′=S.

Рассмотренный алгоритм часто приводит к увеличению количества правил грамматики, но позволяет упростить построение распознавателя для данного языка.

Пример:

Рассмотрим грамматику G({a,b,c},{A,B,C,S},P,S), где P:

$S \to AaB \mid aB \mid cC$
$A \to AB \mid a \mid b \mid B$
$B \to Ba \mid \lambda$
$C \to AB \mid c$

Удалим λ-правила:

1. $W_0=\{B\}$, i:=1.
2. $W_1=\{B,A\}$, $W_1 \neq W_0$, i:=2.
3. $W_2=\{B,A,C\}$, $W_2 \neq W_1$, i:=3.
4. $W_3=\{B,A,C\}$, $W_3=W_2$.
5. VT′=VT, VN′=VN, в P′ входят все правила из P, кроме правила $B \to \lambda$.
6. Рассмотрим отдельно каждое из правил множества P′.

$S \to AaB \mid aB \mid cC$. Нужно исключить все комбинации A,B,C. Получим: $S \to Aa \mid aB \mid a \mid a \mid c$. Добавим новые правила, исключая дубликаты. Получим: $S \to AaB \mid aB \mid cC \mid Aa \mid a \mid c$.

$A \to AB \mid a \mid b \mid B$. Исключая все комбинации A и B, получим $A \to A \mid B$. Добавлять нечего, т.к. правило $A \to B$ в множестве P′ уже есть, а $A \to A$ не имеет смысла.

$B \to Ba$. Исключив из этого правила B, получим $B \to a$, следовательно, окончательно $B \to Ba \mid a$.

$C \to AB \mid c$. Исключив все комбинации A и B, получим $C \to A \mid B$, после добавления в P′ получится $C \to AB \mid A \mid B \mid c$.

7. $S \notin W_3$, поэтому не нужно добавлять новый символ S′, S′=S.

В итоге получим грамматику G′({a,b,c},{A,B,C,S},P′,S), где правила P′ имеют вид:

$S \to AaB \mid aB \mid cC \mid Aa \mid a \mid c$
$A \to AB \mid a \mid b \mid B$
$B \to Ba \mid a$
$C \to AB \mid A \mid B \mid c$.

Построенная грамматика эквивалентна исходной и не содержит λ-правил.

4 Устранение цепных правил

Чтобы устранить цепные правила, для каждого нетерминального символа $\boldsymbol{X}\in VN$ последовательно строится специальное множество цепных символов $N^{\boldsymbol{X}}=\{B \mid X\Rightarrow^{*}B\}$ а затем на основании построенных множеств выполняются преобразования правил P.

1. Шаги 2–5 выполнить для всех нетерминальных символов $\boldsymbol{X}\in VN$.
2. $N^{\boldsymbol{X}}_0=\{X\}$, i:=1.
3. $N^{\boldsymbol{X}}_i= N^{\boldsymbol{X}}_{i-1}\cup\{B \mid (A\to B)\in P, A\in N^{\boldsymbol{X}}_{i-1}\}$.
4. Пока $N^{\boldsymbol{X}}_i\neq N^{\boldsymbol{X}}_{i-1}$, выполняется i:=i+1 и снова шаг 3.
5. Когда станет $N^{\boldsymbol{X}}_i=N^{\boldsymbol{X}}_{i-1}$, строим $N^{\boldsymbol{X}} = N^{\boldsymbol{X}}_i\backslash\{X\}$ и переходим к шагу 2 – к очередному нетерминальному символу, до тех пор, пока они не будут рассмотрены все.
6. Новая грамматика: VT′=VT, VN′=VN, S′=S, в P′ входят все правила из P, кроме правил вида $A\to B$.
7. Для всех правил $(B\to\alpha)\in P'$, если $B\in N^{\boldsymbol{X}}$, то в P′ добавляются правила вида $X\to\alpha$.

Данный алгоритм так же, как и предыдущий, хотя и увеличивает количество правил грамматики, но упрощает построение распознавателей.

Пример:

Рассмотрим устранение цепных правил для грамматики, построенной в предыдущем примере.

Грамматика G({a,b,c},{A,B,C,S},P,S), где правила P:

$S\to AaB \mid aB \mid cC \mid Aa \mid a \mid c$

$A\to AB \mid a \mid b \mid B$

$B\to Ba \mid a$

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

Устраним цепные правила. Рассмотрим все нетерминальные символы, начиная с целевого.

1. $N^{\boldsymbol{S}}_0=\{S\}$, i:=1.
2. $N^{\boldsymbol{S}}_1=\{S\}$, $N^{\boldsymbol{S}}_1=N^{\boldsymbol{S}}_0$. $\mathbf{N}^{\boldsymbol{S}}=N^{\boldsymbol{S}}_1\backslash\{S\}=\varnothing$.
3. $N^{\boldsymbol{A}}_0=\{A\}$, i:=1.
4. $N^{\boldsymbol{A}}_1=\{A,B\}$, $N^{\boldsymbol{A}}_1\neq N^{\boldsymbol{A}}_0$, i:=2.
5. $N^{\boldsymbol{A}}_2=\{A,B\}$, $N^{\boldsymbol{A}}_2=N^{\boldsymbol{A}}_1$, $\mathbf{N}^{\boldsymbol{A}}=N^{\boldsymbol{A}}_2\backslash\{A\}=\{B\}$.
6. $N^{\boldsymbol{B}}_0=\{B\}$, i:=1.
7. $N^{\boldsymbol{B}}_1=\{B\}$, $N^{\boldsymbol{B}}_1=N^{\boldsymbol{B}}_0$, $\mathbf{N}^{\boldsymbol{B}}=N^{\boldsymbol{B}}_1\backslash\{B\}=\varnothing$.
8. $N^{\boldsymbol{C}}_0=\{C\}$, i:=1.

9. N^{C}_1={C,A}, $N^{C}_1 \neq N^{C}_0$, i:=2.

10. N^{C}_2={C,A,B}, $N^{C}_2 \neq N^{C}_1$, i:=3.

11. N^{C}_3={C,A,B}, $N^{C}_3 = N^{C}_2$, $\mathbf{N^{C}} = N^{C}_3 \backslash \{C\} = \{A,B\}$.

Получили: $N^{S}=\varnothing$, $N^{A}=\{B\}$, $N^{B}=\varnothing$, $N^{C}=\{A,B\}$, S′=S. Построим множество правил P′:

S→AaB | aB | cC | Aa | a | c

A→AB | a | b

B→Ba | a

C→AB | c.

Поскольку элементами множеств N^{X} являются только нетерминалы A и B, требуется рассматривать правила только для символов A и B:

A→AB | a | b. Поскольку $A \in N^{C}=\{A,B\}$, необходимо добавить правило C→AB | a | b. Но C→AB уже есть, следовательно, добавляем только C→a | b.

B→Ba | a. Поскольку $B \in N^{C}=\{A,B\}$ и $B \in N^{A}=\{B\}$, необходимо добавить правила C→Ba | a и A→Ba | a. После всех добавлений и устранения дублирующих правил получим новые правила грамматики:

S→AaB | aB | cC | Aa | a | c

A→AB | a | b | Ba

B→Ba | a

C→AB | c | a | b | Ba.

Итоговое множество правил не содержит пустых и цепных правил. Хотя количество правил и увеличилось, но при построении распознавателя с таким множеством работать проще.

3.2.3 Контрольные вопросы

1. С какой целью выполняются преобразования грамматик?
2. Может ли результатом преобразований грамматики являться усложнение вида её правил?
3. Какого вида преобразования грамматик выполняются для упрощения процесса построения распознавателя?
4. Какие грамматики называются приведёнными?
5. Может ли терминальный символ быть недостижимым?
6. Какой символ называется бесплодным? Какой символ может быть бесплодным – терминальный или нетерминальный?
7. Какая грамматика называется грамматикой без λ-правил? Допустимо ли наличие пустых правил в такой грамматике?
8. Какие преобразования необходимо выполнить, чтобы привести КС-грамматику к каноническому виду? К какому виду относятся эти преобразования – они упрощают правила грамматики или усложняют их?

9. Какое множество строится при удалении недостижимых символов – сразу нужное или сначала противоположное ему множество?
10. Какое множество строится при удалении бесплодных символов?

3.3 КС-грамматики в нормальной форме

3.3.1 Нормальная форма Хомского

Нормальная форма Хомского, или бинарная нормальная форма – одна из предопределённых форм для правил КС-грамматики. В нормальную форму Хомского можно преобразовать любую КС-грамматику, но сначала её необходимо привести к каноническому виду.

КС-грамматика ***G(VT,VN,P,S)*** называется *грамматикой в нормальной форме Хомского*, если в её множестве правил P присутствуют только правила следующего вида:

1. $A \to BC$, где $A, B, C \in VN$.
2. $A \to a$, где $A \in VN$, $a \in VT$.
3. $S \to \lambda$, если $\lambda \in L(G)$ и S не должно встречаться в правых частях других правил.

Никакие другие правила не могут встречаться среди правил грамматики в нормальной форме Хомского.

Грамматика в нормальной форме Хомского называется также грамматикой в бинарной нормальной форме (БНФ), т.к. на каждом шаге нетерминальный символ может быть заменен только на два других нетерминальных символа.

Для преобразования грамматики к БНФ сначала требуется преобразовать её к приведенному виду. Соответствующий алгоритм был рассмотрен выше, поэтому будем считать, что КС-грамматика уже находится в каноническом виде.

Рассмотрим алгоритм преобразования грамматики к БНФ.

Сначала множество нетерминальных символов новой грамматики строится на основе множества нетерминальных символов исходной грамматики: $VN' = VN$.

Затем алгоритм работает с правилами P исходной грамматики и в зависимости от их вида строит множество правил P' и пополняет множество нетерминальных символов VN'.

1. Правила вида $A \to a$, $A \to BC$, $S \to \lambda$, где $A, B, C \in VN$, $a \in VT$, переносятся во множество P' без изменений.

2. Если встречается правило вида $(A \to aB) \in P$, где $A, B \in VN$, $a \in VT$, то во множество правил P' добавляются правила $A \to \langle AaB \rangle B$ и $\langle AaB \rangle \to a$, а новый символ $\langle AaB \rangle$ добавляется во множество нетерминальных

символов VN′.

3. Если встречается правило вида (A→Ba)∈P, выполняется аналогичное действие: во множество правил P′ добавляются правила A→B<ABa> и <ABa>→a, а новый символ <ABa> добавляется во множество нетерминальных символов VN′.

4. Если встречается правило вида (A→ab)∈P, где A∈VN, a, b∈VT, то во множество правил P′ добавляются правила A→<Aa><Ab> и <Aa>→a, <Ab>→b, а новые символы <Aa>, <Ab> добавляются во множество нетерминальных символов VN′.

5. Если встречается правило вида $(A \to X_1X_2\ldots X_k) \in P$, где k>2, A∈VN, $\forall i\ X_i \in VT \cup VN$, то во множество правил P′ добавляются правила
$A \to X_1'\langle X_2\ldots X_k\rangle$,
$\langle X_2\ldots X_k\rangle \to X_2'\langle X_3\ldots X_k\rangle$,
…
$\langle X_{k-1}X_k\rangle \to X_{k-1}'X_k'$,
где все новые нетерминальные символы <…> добавляются во множество нетерминальных символов VN′. Что касается символов X_i', то их вид зависит от того, чем являлся исходный символ X_i. Если для некоторого i $X_i \in VN$, то $X_i' \equiv X_i \in VN'$; если $X_i \in VT$, то X_i' – новый нетерминальный символ, т.е. $X_i' \in VN'$ и в P′ нужно добавить правило $X_i' \to X_i$.

Проиллюстрируем данный алгоритм на примере.

Пример: Преобразование КС-грамматики к БНФ.

Дана грамматика G({a,b,c},{A,B,C,S},P,S) в каноническом виде, где P:

S→AaB | Aa | bc
A→AB | a | aC
B→Ba | b
C→AB | c

Первоначально множество нетерминальных символов VN′={A,B,C,S}, затем оно может пополняться. Последовательно рассматриваем правила P и анализируем каждое из них.

1) S→AaB – относится к пункту 5 алгоритма. Следовательно, в P′ включаем S→A<aB>, <aB>→<a>B, <a>→a, а в множество VN′ добавляем <aB>,<a>.

2) S→Aa – относится к 3 пункту алгоритма. Следовательно, в P′ нужно включить S→A<a′>, <a′>→a, а в множество VN′ добавить <a′>. Но ранее уже появился нетерминал <a>, из которого выводился символ a, поэтому можно использовать его, тогда можно ограничиться правилом S→A<a>.

3) S→bc – относится к 4 пункту. Следовательно, в P′ нужно включить S→<b><c>, <b>→b, <c>→c, а в множество VN′ добавить <b>, <c>.

4) A→AB и A→a – в соответствии с первым пунктом алгоритма

включается в P′ без изменений.

5) A→aC – относится ко 2 пункту. Следовательно, в P′ нужно включить A→<a>C, причем замечание из (2) справедливо и в данном случае.

6) B→Ba – аналогично (2), добавляется правило B→B<a>.

7) B→b, а также C→AB, C→c соответствуют требуемому виду правил и переносятся в грамматику без изменений.

Таким образом, получаем грамматику G′({a,b,c},{A,B,C,<aB>,<a>, <b>,<c>,S},P′,S) в нормальной форме Хомского, где P′:

S→A<aB> | A<a> | <b><c>
A→AB | a | <a>C
B→B<a> | b
C→AB | c
<aB>→<a>B
<a>→a
<b>→b
<c>→c

Хотя эта грамматика содержит большее количество правил, чем исходная, но построение распознавателя для неё значительно упрощается.

3.3.2 Левая рекурсия. Нормальная форма Грейбах

Символ A∈VN в КС-грамматике ***G(VT,VN,P,S)*** называется *рекурсивным*, если для него существует цепочка вывода вида $A \Rightarrow^{+} \alpha A \beta$, где $\alpha, \beta \in (VN \cup VT)^{*}$.

Виды рекурсии различают в зависимости от цепочек α и β. Если α=λ, а β≠λ, то рекурсия называется *левой*, а грамматика – *леворекурсивной*. Если наоборот: α≠λ, а β=λ, то рекурсия называется *правой*, а грамматика – *праворекурсивной*. Если обе цепочки α=β=λ, то рекурсия представляет собой *цикл*. В дальнейшем будем рассматривать только приведённые грамматики, в которых нет цепных правил и, соответственно, циклов.

Замечание.

Любая КС-грамматика может быть как праворекурсивной, так и леворекурсивной, а также той и другой одновременно – по различным нетерминальным символам.

Некоторые алгоритмы левостороннего разбора для КС-языков не работают с леворекурсивными грамматиками, поэтому для них необходимо исключить левую рекурсию из правил вывода. Для этой цели существует специальный алгоритм устранения левой рекурсии. Любую грамматику можно привести к нелеворекурсивному или неправорекурсивному виду путем эквивалентных преобразований.

Заметим, что рекурсия лежит в основе построения языков на базе

правил грамматики в форме Бэкуса-Наура, поэтому полностью исключить рекурсию из правил вывода невозможно, можно только устранить какой-то один из её видов – левый или правый.

Алгоритм устранения левой рекурсии.

Алгоритм работает с множеством правил исходной грамматики P, множеством нетерминальных символов VN и двумя счетчиками i и j.

1. Обозначим нетерминальные символы грамматики как $VN=\{A_1,A_2,\ldots,A_n\}$, i:=1.
2. Рассмотрим правила для символа A_i. Если они не содержат левой рекурсии, то переносим их во множество P′ без изменений, а символ A_i добавляем во множество нетерминальных символов VN′. В противном случае перепишем эти правила в следующем виде: $A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p$, где $\forall j \mid 1 \le j \le P$ ни одна из цепочек β_j не начинается с символов A_k, таких, что $k \le i$. Вместо этого правила во множество P′ записываем два правила вида:

   $A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_p \mid \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots \mid \beta_p A_i'$,

   $A_i' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A_i' \mid \alpha_2 A_i' \mid \ldots \mid \alpha_m A_i'$

   Символы A_i и A_i' включаем во множество VN′. Теперь все правила для A_i начинаются либо с терминального символа, либо с нетерминального символа A_k, такого, что $k > i$.
3. Если i=n, то грамматика G′ построена, иначе i:=i+1, j:=1 $\Rightarrow$ на шаг 4.
4. Если для символа A_i во множестве правил P есть правила вида $A_i \to A_j\alpha$, $\alpha \in (VN \cup VT)^*$, то заменить их на правила вида $A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \ldots \mid \beta_m\alpha$, причём $A_j \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ – все правила для символа A_j. Поскольку правая часть правил $A_j \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ уже начинается с терминального символа или нетерминального A_k, $k>j$, то и правая часть правил для A_i будет удовлетворять этому условию.
5. Если j=i–1, то переход на шаг 2, иначе j:=j+1 и переход на шаг 4.
6. Целевым символом новой грамматики G′ становится символ A_k, соответствующий символу S исходной грамматики.

Пример.

Пусть дана грамматика для арифметических выражений:

G ({+,–,/,*,a,b,(,)}, {S,T,E}, P, S), где правила P имеют вид:

(1) $S \to S+T \mid S–T \mid T$ (2) $T \to T*E \mid T/E \mid E$ (3) $E \to (S) \mid a \mid b$.

Эта грамматика леворекурсивная. Выполним эквивалентные преобразования и построим нелеворекурсивную грамматику G′.

Шаг 1. Обозначим множество нетерминальных символов $VN=\{A_1,A_2,A_3\}$, i:=1. Тогда правила грамматики примут вид:

$A_1 \to A_1+A_2 \mid A_1–A_2 \mid A_2$

$A_2 \to A_2*A_3 \mid A_2/A_3 \mid A_3$

$A_3 \to (A_1) \mid a \mid b$.

Шаг 2. Правила для A_1: $A_1 \to A_1+A_2 \mid A_1–A_2 \mid A_2$ запишем в виде $A_1 \to A_1\alpha_1 \mid A_1\alpha_2 \mid \beta_1$, где α_1= +A_2, α_2= –A_2, β_1=A_2. Согласно алгоритму, запишем в P′ новые правила для A_1:

$A_1 \to A_2 \mid A_2A_1'$, $A_1' \to +A_2 \mid –A_2 \mid +A_2A_1' \mid –A_2A_1'$. Символы A_1 и A_1' заносим в VN′. Получим VN′={A_1, A_1'}.

Шаг 3. Поскольку i=1<3, i:=i+1=2, j:=1, переходим на следующий шаг.

Шаг 4. Для символа A_2 во множестве правил P нет правила вида $A_2 \to A_1\alpha$, поэтому на этом шаге никаких действий не выполняется.

Шаг 5. Так как j=1=i–1, переходим на шаг 2.

Шаг 2. Правила для A_2: $A_2 \to A_2*A_3 \mid A_2/A_3 \mid A_3$ запишем в виде $A_2 \to A_2\alpha_1 \mid A_2\alpha_2 \mid \beta_1$, где α_1=*A_3, α_2=/A_3, β_1=A_3. Согласно алгоритму, запишем в P′ новые правила для A_2:

$A_2 \to A_3 \mid A_3A_2'$, $A_2' \to *A_3 \mid /A_3 \mid *A_3A_2' \mid /A_3A_2'$. Символы A_2 и A_2' добавим в VN′. Получим VN′={A_1, A_1', A_2, A_2'}.

Шаг 3. Поскольку i=2<3, то i:=3, j:=1, переходим на следующий шаг.

Шаг 4. Для символа A_3 во множестве правил P нет правила вида $A_3 \to A_1\alpha$, поэтому на этом шаге никаких действий не выполняется.

Шаг 5. Так как j=1< i–1, то j:=j+1=2 и переходим на шаг 4.

Шаг 4. Для символа A_3 во множестве правил P нет правила вида $A_3 \to A_2\alpha$, поэтому на этом шаге никаких действий не выполняется.

Шаг 5. Так как j=2=i–1, переходим на шаг 2.

Шаг 2. Правила для A_3: $A_3 \to (A_1) \mid a \mid b$ не содержат левой рекурсии, поэтому поместим их в P′ без изменений. Символ A_3 добавим в VN′. Получим VN′={A_1, A_1', A_2, A_2', A_3}.

Шаг 3. Поскольку i=3, построение грамматики закончено.

В итоге получили нелеворекурсивную грамматику G ({+,–,/,*,a,b,(,)}, {A_1, A_1', A_2, A_2', A_3}, P, A_1), где правила P имеют вид:

$A_1 \to A_2 \mid A_2A_1'$,
$A_1' \to +A_2 \mid –A_2 \mid +A_2A_1' \mid –A_2A_1'$.

$A_2 \to A_3 \mid A_3A_2'$,
$A_2' \to *A_3 \mid /A_3 \mid *A_3A_2' \mid /A_3A_2'$.
$A_3 \to (A_1) \mid a \mid b$.

Грамматика называется грамматикой *в нормальной форме Грейбах*, если она не является леворекурсивной и в её множестве правил присутствуют только правила следующего вида:

1. $A \to a\alpha$, где $a \in VT$ $\alpha \in VN^*$.
2. $S \to \lambda$, если $\lambda \in L(G)$ и S не встречается в правых частях правил.

Нормальная форма Грейбах удобна для построения нисходящих левосторонних распознавателей.

3.3.3 Контрольные вопросы

1. Какого типа грамматики могут быть приведены к БНФ?
2. Каковы требования на правила грамматики в БНФ?

3. Обязательно ли наличие рекурсии в правилах грамматики? Какого вида рекурсия в них может использоваться? Может ли в правилах одной грамматики присутствовать рекурсия разных видов?
4. От какого вида рекурсии в правилах грамматики принято избавляться и для чего это делается?

3.4 Виды распознавателей КС-языков

3.4.1 Распознаватели КС-языков с возвратом

Распознаватели КС-языков с возвратом представляют собой самый простой тип распознавателей, основанный на модели недетерминированного МП-автомата. Как известно, при работе такого автомата возможны альтернативные варианты его поведения. Существуют два варианта реализации алгоритма работы недетерминированного МПА.

Первый вариант предполагает запоминание на каждом шаге работы всех возможных следующих состояний. Алгоритм моделирует работу автомата по одному из возможных переходов до тех пор, пока либо не будет достигнута конечная конфигурация автомата, либо возникнет ситуация, когда следующая конфигурация не определена. В последнем случае происходит возврат на несколько шагов назад в ту конфигурацию, из которой был возможен альтернативный вариант, и моделирование продолжается. Если какая-то из последовательностей шагов приводит к заключительной конфигурации, то цепочка считается принятой, а автомат заканчивает работу. Если все варианты работы перебраны, а конечная конфигурация не достигнута, то алгоритм завершается с ошибкой.

Во втором варианте алгоритм моделирования МПА на каждом шаге работы при возникновении неоднозначности с несколькими возможными следующими состояниями должен запускать свою новую копию для обработки каждого из этих состояний. Если хотя бы одна из копий приводит в заключительную конфигурацию, то цепочка распознана и работа всех остальных копий также завершается. Если ни одна из копий не достигнет конечной конфигурации, алгоритм завершается с ошибкой и цепочка не принимается.

Во втором варианте различные копии алгоритма должны выполняться параллельно, следовательно, необходимо наличие механизма управления параллельными процессами. Кроме того, количество копий алгоритма заранее неизвестно, их может быть много, в то время как количество одновременно выполняющихся процессов ограничено. Этим объясняется то, что большее распространение получил первый вариант алгоритма, который называется «разбором с возвратами».

Хотя МПА представляет собой односторонний распознаватель, алгоритм моделирования его работы предусматривает возврат к уже прочитанной цепочке символов.

Кроме того, любой практический алгоритм должен завершаться за конечное время. Алгоритм моделирования работы произвольного МПА в общем случае не удовлетворяет такому условию. Например, после считывания всей входной цепочки МПА может совершить произвольное число (может быть, и бесконечное) λ-переходов. В таком случае, если входная цепочка не принята, алгоритм может никогда не завершиться.

Для того чтобы избежать таких ситуаций, алгоритм работы МПА с возвратами строят не для произвольных автоматов, а для автоматов, удовлетворяющих некоторым заданным условиям. Обычно грамматику исходного языка подвергают сначала некоторым эквивалентным преобразованиям, приводя её к заранее заданному виду.

Вычислительная сложность алгоритмов:

Алгоритмы разбора с возвратами имеют экспоненциальную сложность, т.е. их вычислительные затраты экспоненциально зависят от длины входной цепочки символов. Обозначим эту цепочку $\alpha \mid \alpha \in VT^*$, её длина $|\alpha| = n$. Конкретный характер зависимости определяется вариантом реализации алгоритма.

В общем случае при первом варианте реализации время выполнения алгоритма для произвольной КС-грамматики имеет экспоненциальную, а необходимый объём памяти – линейную зависимость от длины входной цепочки: $T=O(e^n)$, $M=O(n)$. При втором варианте ситуация обратная – время имеет линейную зависимость, а требуемый объём памяти – экспоненциальную: $T=O(n)$, $M=O(e^n)$. В любом случае, непомерно большие вычислительные затраты на реализацию алгоритма существенно ограничивают возможности его применения. Для конкретных классов КС-языков существуют более эффективные алгоритмы распознавания.

Основные варианты разбора с возвратом – это нисходящий распознаватель и распознаватель на основе алгоритма «сдвиг-свёртка».

3.4.1.1 Нисходящий распознаватель с возвратом

Этот распознаватель моделирует работу МПА с одним состоянием q: $R(\{q\}, V, Z, \delta, q, S, \{q\})$. Автомат распознаёт цепочки КС-языка, задаваемого грамматикой $G(VT,VN,P,S)$. Входной алфавит автомата содержит терминальные символы грамматики $V=VT$, а алфавит магазинных символов $Z=VT \cup VN$.

Начальная конфигурация автомата (q,α,S), где входная цепочка $\alpha \in VT^*$, а в стеке находится целевой символ грамматики. Конечная (заключительная) конфигурация автомата (q,λ,λ).

Функция перехода строится на основе правил грамматики следующим образом:

1. Если правило $(A \to \mu) \in P$, то $(q,\mu) \in \delta(q,\lambda,A)$, $A \in VN$, $\mu \in (VT \cup VN)^*$.

2. $\forall a \in VT \ (q,\lambda) \in \delta(q,a,a)$.

Работу автомата можно описать следующим образом. Если на верхушке стека находится нетерминальный символ A, то его можно заменить на цепочку символов μ, если $(A \to \mu) \in P$, не сдвигая при этом считывающую головку автомата. Этот шаг работы алгоритма называется «подбор альтернативы». Если на верхушке стека находится терминальный символ, совпадающий с текущим входным символом, то его можно выбросить из стека, а считывающую головку передвинуть на один символ вправо. Данный шаг называется «выброс». Если в грамматике окажется более одного правила вида $(A \to \mu) \in P$ с различными μ, то функция будет содержать более одного следующего состояния, следовательно, у автомата будет несколько альтернатив.

Рассмотренный автомат строит левосторонние выводы для грамматики G, следовательно, эта грамматика не должна быть леворекурсивной. Ранее было показано, что к нелеворекурсивному виду может быть приведена произвольная грамматика, следовательно, алгоритм является универсальным. Поскольку цепочка входных символов считывается слева направо, а вывод является левосторонним, дерево строится сверху вниз. Такой распознаватель называется *нисходящим*.

На каждом шаге работы МП-автомата необходимо принимать решение о том, выполнять выброс или подбор альтернативы. Алгоритм должен иметь возможность выбирать поочерёдно все альтернативы, следовательно, необходимо хранить информацию о том, какие варианты уже использованы, чтобы вернуться к этому шагу и подобрать другие альтернативы. Такой алгоритм разбора *называется алгоритмом с подбором альтернатив*.

Существует множество способов моделирования работы данного МП-автомата. Рассмотрим один из примеров его реализации.

Алгоритм распознавателя с подбором альтернатив.

Для работы алгоритма используется МПА, построенный на основе исходной КС-грамматики G(VT,VN,P,S). Все правила из множества P представим в виде $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, т.е. для каждого нетерминального символа $A \in VN$ перенумеруем все возможные альтернативы. Входная цепочка имеет вид $\alpha = a_1 a_2 \ldots a_n$, $|\alpha| = n$, $a_i \in VT \ \forall i$. В алгоритме используется ещё одно дополнительное состояние b для обратного хода (back – возврат). Для хранения уже выбранных альтернатив используется дополнительный стек L_2, который может содержать символы входного языка автомата $a \in VT$ и символы вида A_j, где $A \in VN$ – это будет означать, что среди всех возможных правил для символа A была выбрана альтернатива с номером j.

Итак, алгоритм работает с двумя стеками: L_1 – стек МПА и L_2 – стек возвратов, причём в цепочку стека L_1 символы помещаются слева, а в

цепочку стека L_2 – справа.

Состояние алгоритма на каждом шаге определяется четырьмя параметрами: (Q,i,L_1,L_2), где Q – текущее состояние автомата (q или b), i – положение считывающей головки во входной цепочке ($1 < i \le n+1$), L_1 и L_2 – содержимое стеков.

Начальным состоянием алгоритма является $(q,1,S,\lambda)$, где S – целевой символ грамматики. Алгоритм начинает работу с начального состояния и циклически выполняет 6 шагов до тех пор, пока не перейдёт в конечное состояние или не обнаружит ошибку. Заключительная конфигурация имеет вид $(q, n+1, \lambda, \mu)$.

Алгоритм выполняет циклически следующие шаги:

Шаг 1 («Разрастание»):$(q, i, A\beta, \mu) \to (q, i, \gamma_1\beta, \mu A_1)$, где $\beta \in (VN \cup VT)^*$, μ – содержимое стека возвратов L_2, если $A \to \gamma_1$ – первая из всех альтернатив для символа A (заменяем нетерминал в стеке МПА по правилу вывода, а в стек возврата записываем выбранную альтернативу).

Шаг 2 («*Успешное сравнение*»): $(q, i, a\beta, \mu) \to (q, i+1, \beta, \mu a)$, если $a = a_i$, $a \in VT$ (текущий символ стека МПА совпадает с i-м во входной цепочке – тогда переписываем этот терминальный символ ***a*** в стек возвратов и переходим к рассмотрению следующего ***(i+1)*** символа).

Шаг 3 («*Завершение*»): Если в стеке МПА пусто, то: $(q, i, \lambda, \mu) \to (b, i, \lambda, \mu)$, если $i \ne n+1$, и разбор завершён, если $i = n+1$.

Шаг 4 («*Неуспешное сравнение*»): $(q, i, a\beta, \mu) \to (b, i, a\beta, \mu)$, если $a \ne a_i$, $a \in VT$ (текущий символ стека МПА отличен от i-го во входной цепочке – тогда переходим в состояние возврата).

Шаг 5 («*Возврат по входу*»): $(b, i, \beta, \mu a) \to (b, i-1, a\beta, \mu)$ $\forall a \in VT$ (возвращаемся к предыдущему символу входной цепочки, переписав терминальный символ из стека возврата в стек МПА) – если $i>1$.

Шаг 6 («*Другая альтернатива*»): Состояние алгоритма $(b, i, \gamma_j\beta, \mu A_j)$.

- Если существует другая альтернатива для символа $A \in VN$: $A \to \gamma_{j+1}$, то перейти $(b, i, \gamma_j\beta, \mu A_j) \to (q, i, \gamma_{j+1}\beta, \mu A_{j+1})$ (переходим к рассмотрению другой альтернативы для последнего примененного правила для нетерминала A – записываем номер очередной альтернативы в стек возврата вместо предыдущего и заменяем цепочку в стеке).
- Если $A \equiv S$, $i = 1$ и нет больше неиспользованных альтернатив для символа S, то сигнализировать об ошибке и прекратить выполнение.
- Если нет больше неиспользованных альтернатив для символа A, но $A \ne S$, то перейти $(b, i, \gamma_j\beta, \mu A_j) \to (b, i, A\beta, \mu)$ (возвращаем последний нетерминал из стека возврата в стек МПА, заменив им выведенную ранее из него цепочку).

В случае успешного завершения алгоритма на основе содержимого стека возврата можно построить цепочку вывода. Если в стеке содержится символ A_j, то в цепочку вывода помещают номер правила, соответствующего альтернативе $A \to \gamma_j$; при этом игнорируются все терминальные символы, находящиеся в стеке возврата.

Заметим, что в состоянии прямого хода алгоритма (q) его поведение на очередном шаге определяется содержимым стека L_1, а в состоянии обратного хода (b) – содержимым стека L_2.

Пример. Рассмотрим грамматику G ({+,–,/,*,a,b,(,)}, {S, R, T, F, E}, P, S), где правила P имеют вид:

$S \to T$ *[S1]* $\mid$ TR *[S2]*,
$R \to +T$ *[R1]* $\mid$ –T *[R2]* $\mid$ +TR *[R3]* $\mid$ –TR *[R4]*.
$T \to E$ *[T1]* $\mid$ EF *[T2]*,
$F \to *E$ *[F1]* $\mid$ /E *[F2]* $\mid$ *EF *[F3]* $\mid$ /EF *[F4]*.
$E \to (S)$ *[E1]* $\mid$ a *[E2]* $\mid$ b *[E3]*.

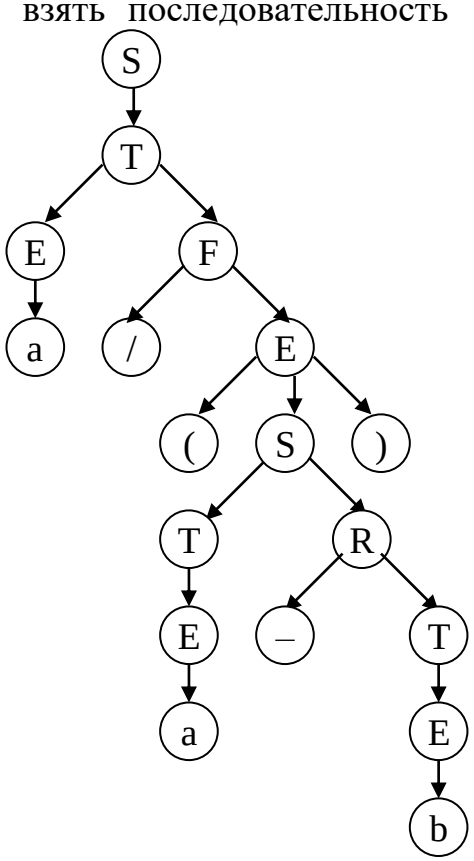
Здесь каждое правило сопровождается соответствующим символом *[Ai]*, который будет заноситься в стек возврата. Это нелеворекурсивная грамматика для арифметических выражений, которая была построена ранее. Рассмотрим процесс выполнения разбора цепочки a/(a–b). Состояния алгоритма будем записывать в фигурных скобках. Подчеркиваем одной чертой символы, с которыми работаем в настоящий момент; двумя чертами – не совпавший терминальный символ. В круглых скобках возле знака выводимости записываем номер соответствующего шага алгоритма.

1,2 $\{q,1,\underline{S},\lambda\} \vdash_{(1)} \{q,1,\underline{T},S_1\}$	раскроем нетерминал T
3 $\vdash_{(1)} \{q,1,\underline{E},S_1T_1\}$	раскроем нетерминал E
4 $\vdash_{(1)} \{q,1,\underline{(}S),S_1T_1E_1\}$	первый символ – не “(”
5 $\vdash_{(4)} \{b,1,\underline{(S)},S_1T_1\underline{E_1}\}$	выберем след. альтернативу для E
6 $\vdash_{(6,1)} \{q,1,\underline{a},S_1T_1E_2\}$	первый символ совпал, сдвиг головки
7 $\vdash_{(2)} \{q,2,\underline{\lambda},S_1T_1E_2a\}$	в стеке пусто, но i≠n+1 – возврат
8 $\vdash_{(3)} \{b,2,\lambda,S_1T_1E_2a\}$	возвращаем “a” из стека возврата в L_1
9 $\vdash_{(5)} \{b,1,\underline{a},S_1T_1\underline{E_2}\}$	выберем след. альтернативу для E
10 $\vdash_{(6,1)} \{q,1,\underline{b},S_1T_1E_3\}$	первый символ – не “b”
11 $\vdash_{(4)} \{b,1,\underline{b},S_1T_1\underline{E_3}\}$	вернем E из L_2 в L_1
12 $\vdash_{(6,3)} \{b,1,\underline{E},S_1\underline{T_1}\}$	выберем след. альтернативу для T
13 $\vdash_{(6,1)} \{q,1,\underline{E}F,S_1T_2\}$	раскроем нетерминал E
14 $\vdash_{(1)} \{q,1,\underline{(}S)F,S_1T_2E_1\}$	первый символ – не “(”
15 $\vdash_{(4)} \{b,1,\underline{(S)}F,S_1T_2\underline{E_1}\}$	выберем след. альтернативу для E
16 $\vdash_{(6,1)} \{q,1,\underline{a}F,S_1T_2E_2\}$	первый символ совпал, сдвиг головки
17 $\vdash_{(2)} \{q,2,\underline{F},S_1T_2E_2a\}$	раскроем нетерминал F
18 $\vdash_{(1)} \{q,2,\underline{*}E,S_1T_2E_2aF_1\}$	второй символ – не “*”
19 $\vdash_{(4)} \{b,2,\underline{*E},S_1T_2E_2a\underline{F_1}\}$	выберем след. альтернативу для F

20⊢(6,1) {q,2,/E,$S_1T_2E_2aF_2$} второй символ совпал, перепишем в L_2
21⊢(2) {q,3,E,$S_1T_2E_2aF_2$/} раскроем нетерминал E
22⊢(1) {q,3,(S),$S_1T_2E_2aF_2/E_1$} третий символ совпал, перепишем в L_2
23⊢(2) {q,4,S),$S_1T_2E_2aF_2/E_1$(} раскроем нетерминал S
24⊢(1) {q,4,T),$S_1T_2E_2aF_2/E_1(S_1$} раскроем нетерминал T
25⊢(1) {q,4,E),$S_1T_2E_2aF_2/E_1(S_1T_1$} раскроем нетерминал E
26⊢(1) {q,4,(S)),$S_1T_2E_2aF_2/E_1(S_1T_1E_1$} 4-й символ – не “(”
27⊢(4) {b,4,(S)),$S_1T_2E_2aF_2/E_1(S_1T_1E_1$} выберем след. альтернативу для E
28⊢(6,1) {q,4,a),$S_1T_2E_2aF_2/E_1(S_1T_1E_2$} 4-й символ совпал, переносим в L_2
29⊢(2) {q,5,),$S_1T_2E_2aF_2/E_1(S_1T_1E_2a$} 5-й символ не совпал
30⊢(4) {b,5,),$S_1T_2E_2aF_2/E_1(S_1T_1E_2a$} в стеке L_2 – терминал, вернём его
31⊢(5) {b,4,a),$S_1T_2E_2aF_2/E_1(S_1T_1E_2$} выберем след. альтернативу для E
32⊢(6,1) {q,4,b),$S_1T_2E_2aF_2/E_1(S_1T_1E_3$} 4-й символ не совпал
33⊢(4) {b,4,b),$S_1T_2E_2aF_2/E_1(S_1T_1E_3$} нет альтернатив для E
34⊢(6,3) {b,4,E),$S_1T_2E_2aF_2/E_1(S_1T_1$} выберем след. альтернативу для T
35⊢(6,1) {q,4,EF),$S_1T_2E_2aF_2/E_1(S_1T_2$} раскроем нетерминал E
36⊢(1) {q,4,(S)F),$S_1T_2E_2aF_2/E_1(S_1T_2E_1$} 4-й символ не совпал
37⊢(4) {b,4,(S)F),$S_1T_2E_2aF_2/E_1(S_1T_2E_1$} выберем след. альтернативу для E
38⊢(6,1) {q,4,aF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2$} 4-й символ совпал
39⊢(2) {q,5,F),$S_1T_2E_2aF_2/E_1(S_1T_2E_2a$} раскроем нетерминал F
40⊢(1) {q,5,*E),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_1$} 5-й символ не совпал
41⊢(4) {b,5,*E),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_1$} выберем след. альтернативу для F
42⊢(6,1) {q,5,/E),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_2$} 5-й символ не совпал
43⊢(4) {b,5,/E),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_2$} выберем след. альтерн. для F
44⊢(6,1) {q,5,*EF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_3$} 5-й символ не совпал
45⊢(4) {b,5,*EF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_3$} выберем след. альтерн. для F
46⊢(6,1) {q,5,/EF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_4$} 5-й символ не совпал
47⊢(4) {b,5,/EF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2aF_4$} для F нет альтернативы
48⊢(6,3) {b,5,F),$S_1T_2E_2aF_2/E_1(S_1T_2E_2a$} возврат по символу
49⊢(5) {b,4,aF),$S_1T_2E_2aF_2/E_1(S_1T_2E_2$} выберем след. альтернативу для E
50⊢(6,1) {q,4,bF),$S_1T_2E_2aF_2/E_1(S_1T_2E_3$} 4-й символ не совпал
51⊢(4) {b,4,bF),$S_1T_2E_2aF_2/E_1(S_1T_2E_3$} вернем E из L_2 в L_1
52⊢(6,3) {b,4,EF),$S_1T_2E_2aF_2/E_1(S_1T_2$} вернем T из L_2 в L_1
53⊢(6,3) {b,4,T),$S_1T_2E_2aF_2/E_1(S_1$} выберем след. альтернативу для S
54⊢(6,1) {q,4,TR),$S_1T_2E_2aF_2/E_1(S_2$} раскроем нетерминал T
55⊢(1) {q,4,ER),$S_1T_2E_2aF_2/E_1(S_2T_1$} раскроем нетерминал E
56⊢(1) {q,4,(S)R),$S_1T_2E_2aF_2/E_1(S_2T_1E_1$} 4-й символ не совпал
57⊢(4) {b,4,(S)R),$S_1T_2E_2aF_2/E_1(S_2T_1E_1$} выберем след. альтернативу для E
58⊢(6,1) {q,4,aR),$S_1T_2E_2aF_2/E_1(S_2T_1E_2$} 4-й символ совпал, переносим в L_2
59⊢(2) {q,5,R),$S_1T_2E_2aF_2/E_1(S_2T_1E_2a$} раскроем нетерминал R
60⊢(1) {q,5,+T),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_1$} 5-й символ не совпал
61⊢(4) {b,5,+T),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_1$} выберем след. альтернативу для R
62⊢(6,1) {q,5,–T),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2$} 5-й символ совпал, переносим в L_2
63⊢(2) {q,6,T),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2$–} раскроем нетерминал T

64⊢(1) {q,6,E),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1$} раскроем нетерминал E
65⊢(1) {q,6,(S)),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_1$} 6-й символ не совпал
66⊢(4) {b,6,(S)),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_1$} выберем след. альтер. для E
67⊢(6,1) {q,6,a),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_2$} 6-й символ не совпал
68⊢(4) {b,6,a),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_2$} выберем след. альтер. для E
69⊢(6,1) {q,6,b),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_3$} 6-й символ совпал
70⊢(2) {q,7,),$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_3$b} 7-й символ совпал
71⊢(2) {q,8,λ,$S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_3$b)}⊢(3) Разбор закончен stop(+)

Разбор закончен, стек МПА пуст. Если взять последовательность нетерминальных символов из стека возвратов, получим цепочку номеров альтернатив и можем построить дерево вывода.



В стеке возврата цепочка $S_1T_2E_2aF_2/E_1(S_2T_1E_2aR_2–T_1E_3$b. Удалив терминальные символы, получим $S_1T_2E_2F_2E_1S_2T_1E_2R_2T_1E_3$. Тогда цепочка вывода будет иметь вид: S⇒T⇒EF⇒aF⇒a/E⇒a/(S)⇒a/(TR)⇒a/(ER)⇒a/(aR)⇒a/(a–T)⇒a/(a–E)⇒a/(a–b).
Дерево вывода показано на рисунке справа.

Из рассмотренного примера видно, что *недостатком* алгоритма нисходящего разбора с возвратами является большое время работы. Для разбора достаточно короткой цепочки из 7 символов потребовалось выполнить 70 шагов.

Преимуществом алгоритма является его простота реализации и универсальность.

Сам по себе данный алгоритм не используется в компиляторах, но его основные принципы лежат в основе многих нисходящих распознавателей, строящих левосторонние выводы и работающих без использования возвратов.

3.4.1.2 Распознаватель на основе алгоритма «сдвиг-свёртка»

Этот распознаватель строится на основе расширенного МПА с одним состоянием q: R({q},V,Z,δ,q,λ,{q}). Автомат распознаёт цепочки КС-языка, задаваемого грамматикой G(VT,VN,P,S). Входной алфавит автомата содержит терминальные символы грамматики V=VT, а алфавит магазинных символов Z=VT∪VN.

Начальная конфигурация автомата (q,α,λ), т.е. считывающая головка находится в начале входной цепочки $\alpha \in VT^*$, а стек пуст. Конечная

конфигурация автомата (q,λ,S), т.е. в стеке находится целевой символ.

Функция перехода строится на основе правил P грамматики G:

1. Если правило $(A \to \mu) \in P$, то $(q,A) \in \delta(q,\lambda,\mu)$, $A \in VN$, $\mu \in (VT \cup VN)^*$.
2. $\forall a \in VT \ (q,a) \in \delta(q,a,\lambda)$.

Работу автомата можно описать следующим образом. Если на верхушке стека находится цепочка символов μ, то ее можно заменить на нетерминальный символ A, если $(A \to \mu) \in P$, не сдвигая считывающую головку автомата. Этот шаг работы алгоритма называется «свёртка». С другой стороны, если считывающая головка обозревает символ входной цепочки a, то его можно поместить в стек, а считывающую головку передвинуть на один символ вправо. Данный шаг называется «сдвиг» или «перенос». Алгоритм называется «сдвиг-свёртка» или «перенос-свёртка».

Данный расширенный автомат строит правосторонние выводы для грамматики G, читает цепочку входных символов слева направо, поэтому строит дерево снизу вверх. Такой распознаватель является восходящим.

Для моделирования такого автомата необходимо, чтобы грамматика не содержала цепных правил и λ-правил. Как было рассмотрено ранее, к такому виду может быть приведена любая КС-грамматика, поэтому алгоритм является универсальным.

Рассмотренный автомат имеет больше неоднозначностей, чем распознаватель, основанный на выборе альтернатив. На каждом шаге работы автомата должны быть решены следующие вопросы:

- что необходимо выполнить – сдвиг или свёртку;
- если выполнять свёртку, то какую цепочку μ выбрать для поиска правил (эта цепочка должна находиться в правой части правил);
- какое из правил выбрать для свёртки, если в грамматике окажется более одного правила вида $(A \to \mu) \in P$ с одинаковой правой частью и различными левыми частями A.

Чтобы промоделировать работу этого расширенного МПА, надо на каждом шаге запоминать все предпринятые действия, чтобы иметь возможность при необходимости вернуться к уже сделанному шагу и проделать все действия иначе. Таким образом должны быть перебраны все возможные варианты.

Рассмотрим один из возможных вариантов реализации алгоритма.

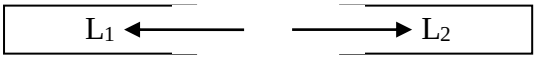
Распознаватель с возвратами на основе алгоритма «сдвиг-свёртка»

Для работы алгоритма используется расширенный МПА, построенный на основе исходной КС-грамматики G(VT,VN,P,S). Все правила из множества P перенумеруем слева направо и сверху вниз в порядке их записи в форме Бэкуса-Наура. Входная цепочка имеет вид $\alpha = a_1 a_2 \ldots a_n$, $|\alpha| = n$, $a_i \in VT \ \forall i$.

Аналогично алгоритму нисходящего распознавателя, в

рассматриваемом алгоритме используется дополнительное состояние b и стек возвратов L_2. Этот стек может содержать номера правил грамматики, использованных для свёртки, если на очередном шаге алгоритма выполнялась свёртка, или 0, если на очередном шаге выполнялся сдвиг.

В итоге алгоритм работает с двумя стеками, L_1 – стек МПА и L_2 – стек возвратов, причём первый содержит цепочку символов, а второй – целые числа от 0 до m, где m – количество правил грамматики. В цепочку стека L_1 символы помещаются справа, а числа в стек L_2 – слева.



Состояние алгоритма на каждом шаге определяется четырьмя параметрами: (Q,i,L_1,L_2), где Q – текущее состояние автомата (q или b), i – положение считывающей головки во входной цепочке α $(1 < i \le n+1)$, L_1 и L_2 – содержимое стеков.

Начальным состоянием алгоритма является $(q,1,\lambda,\lambda)$. Алгоритм начинает работу с начального состояния и циклически выполняет 5 шагов до тех пор, пока не перейдёт в конечное состояние или не обнаружит ошибку. Конечное состояние алгоритма $(q,n+1,S,\gamma)$.

Алгоритм выполняет циклически следующие шаги:

Шаг 1 («*Попытка свертки*»): $(q, i, \mu\beta, \gamma) \vdash (q, i, \mu A, j\gamma)$, где $\beta \in (VN \cup VT)^+$, γ – содержимое стека возвратов L_2, если $(A \to \beta) \in P$ – первое из всех возможных правил из множества P с номером j для подцепочки β, причём оно является первым подходящим правилом для цепочки $\mu\beta$, для которой правило вида $A \to \beta$ существует (заменяем цепочку в стеке МПА на нетерминал – «сворачиваем» её, а в стек возврата записываем номер использованного правила). Если удалось выполнить свёртку, то возвращаемся к шагу 1, иначе переходим к шагу 2.

Шаг 2 («*Перенос-сдвиг*»): Если $i < n+1$, то $(q, i, \mu, \gamma) \vdash (q, i+1, \mu a_i, 0\gamma)$, где $a_i \in VT$. Если $i = n+1$, то идти дальше, иначе перейти к шагу 1.

Шаг 3 («*Завершение*»): Если состояние алгоритма $(q, n+1, S, \gamma)$, то разбор завершён и алгоритм заканчивает работу, иначе перейти к шагу 4.

Шаг 4 («*Переход к возврату*»): $(q, n+1, \mu, \gamma) \vdash (b, n+1, \mu, \gamma)$.

Шаг 5 («*Возврат*»): При возврате возможны следующие варианты:

1) Если исходное состояние алгоритма $(b, i, \mu A, j\gamma)$ $(j>0)$:

- Перейти $(b, i, \mu A, j\gamma) \vdash (q, i, \mu' B, k\gamma)$, если $(A \to \beta) \in P$ – это правило с номером j и существует правило $(B \to \beta') \in P$ с номером k, $k>j$, такое, что $\mu\beta = \mu'\beta'$, после чего вернуться к шагу 1.
- Перейти $(b, i, \mu A, j\gamma) \vdash (b, n+1, \mu\beta, \gamma)$, если $i=n+1$, $(A \to \beta) \in P$ – это правило с номером j и не существует других правил из множества P с номером k, $k>j$, таких, что их правая часть является суффиксом цепочки $\mu\beta$, после этого вернуться к шагу 5.

- Перейти $(b, i, \mu A, j\gamma) \vdash (q, i+1, \mu\beta a_i, 0\gamma)$, где $a_i \in VT$, если $i \neq n+1$, $(A \to \beta) \in P$ – это правило с номером j и не существует других правил из множества P с номером k>j, таких, что их правая часть является правой подцепочкой из цепочки $\mu\beta$, после этого перейти к шагу 1.
- Иначе сигнализировать об ошибке и прекратить выполнение.

2) Если исходное состояние алгоритма $(b, i, \mu a, 0\gamma)$, $a \in VT$, то если i>1, тогда перейти в следующее состояние $(b, i–1, \mu, \gamma)$ и вернуться к шагу 5, иначе сигнализировать об ошибке и прекратить выполнение алгоритма.

При успешном завершении алгоритма на основе содержимого стека возврата можно построить цепочку вывода. Для этого достаточно удалить из стека все 0 и будет получена последовательность номеров правил, используемых при выводе цепочки.

Пример: Пусть дана грамматика для арифметических выражений:

G ({+,–,/,*,a,b,(,)}, {S,T,E}, P, S), где правила P имеют вид:

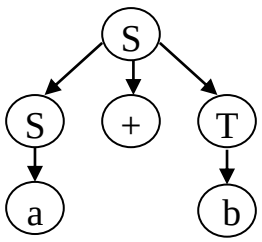
$S \to S+T$ *[1]* | $S–T$ *[2]* | $T*E$ *[3]* | T/E *[4]* | (S) *[5]* | a *[6]* | b *[7]*

$T \to T*E$ *[8]* | T/E *[9]* | (S) *[10]* | a *[11]* | b *[12]*

$E \to$ (S) *[13]* | a *[14]* | b *[15]*.

Рассмотрим разбор двух цепочек: ‘a+b’ и ‘a/(a–b)’.

1) $(q,1,\lambda,\lambda) \vdash_{(2)} (q,2,\underline{a},[0,\lambda]) \vdash_{(1)} (q,2,S,[6,0]) \vdash_{(2)} (q,3,S+,[0,6,0]) \vdash_{(2)} (q,4,S+\underline{b},[0,0,6,0]) \vdash_{(1)} (q,4,S+S,[7,0,0,6,0,0]) \vdash_{(4)} (b,4,S+\underline{S},[\underline{7},0,0,6,0,0]) \vdash_{(5)} (q,4,\underline{S+T},[12,0,0,6,0,0]) \vdash_{(1)} (q,4,S,[1,12,0,0,6,0,0]) \vdash_{(3)}$ stop



Алгоритм успешно завершён, в стеке возврата содержатся номера правил, которые участвовали в выводе цепочки: $L_2 = [1,12,0,0,6,0] = [1,12,6]$. $\Rightarrow$ цепочка вывода имеет вид $S \Rightarrow S+T \Rightarrow S+b \Rightarrow a+b$. Построим дерево вывода.

2) ‘a/(a–b)’:

	$(q,1,\lambda,\lambda)$	
$\vdash_{(2)}$	$(q,2,\underline{a},[0,\lambda])$	перенесли символ ”a”
$\vdash_{(1)}$	$(q,2,S,[6,0])$	выполнили свертку
$\vdash_{(2)}$	$(q,3,S/,[0,6,0])$	перенесли символ ”/”
$\vdash_{(2)}$	$(q,4,S/(,[0,0,6,0])$	перенесли символ ”(”
$\vdash_{(2)}$	$(q,5,S/(a,[0,0,0,6,0])$	перенесли символ ”a”
$\vdash_{(1)}$	$(q,5,S/(S,[6,0,0,0,6,0])$	выполнили свертку по №6
$\vdash_{(2)}$	$(q,6,S/(S–,[0,6,0,0,0,6,0])$	перенесли символ ”–”
$\vdash_{(2)}$	$(q,7,S/(S–b,[0,0,6,0,0,0,6,0])$	перенесли символ ”b”
$\vdash_{(1)}$	$(q,7,S/(S–S,[7,0,0,6,0,0,0,6,0])$	выполнили свёртку по №7
$\vdash_{(2)}$	$(q,8,S/(S–S),[0,7,0,0,6,0,0,0,6,0])$	перенесли символ ”)”
$\vdash_{(4)}$	$(b,8,S/(S–S),[0,7,0,0,6,0,0,0,6,0])$	перешли к возврату
$\vdash_{(5)}$	$(b,7,S/(S–S,[7,0,0,6,0,0,0,6,0])$	выполнили возврат
$\vdash_{(5)}$	$(q,7,S/(S–T,[12,0,0,6,0,0,0,6,0])$	заменили на другое правило
$\vdash_{(1)}$	$(q,7,S/(S,[2,12,0,0,6,0,0,0,6,0])$	выполнили свёртку S–T по №2

⊢(2)	(q,8,S/(S),[0,2,12,0,0,6,0,0,0,6,0])	перенесли символ ”)”
⊢(1)	(q,8,S/S,[5,0,2,12,0,0,6,0,0,0,6,0])	выполнили свёртку по №5
⊢(4)	(b,8,S/S,[5,0,2,12,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.1.1)	(q,8,S/T,[10,0,2,12,0,0,6,0,0,0,6,0])	заменили на другое правило
⊢(4)	(b,8,S/T,[10,0,2,12,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.1.1)	(q,8,S/E,[13,0,2,12,0,0,6,0,0,0,6,0])	заменили на другое правило
⊢(4)	(b,8,S/E,[13,0,2,12,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.1.2)	(b,8,S/(S),[0,2,12,0,0,6,0,0,0,6,0])	вернулись назад по 13 правилу
⊢(5.2)	(b,7,S/(S,[2,12,0,0,6,0,0,0,6,0])	возврат по символу ”)”
⊢(5.1.3)	(q,8,S/(S–T),[0,12,0,0,6,0,0,0,6,0])	развернули пр.2, сдвинули ”)”
⊢(4)	(b,8,S/(S–T),[0,12,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.2)	(b,7,S/(S–T,[12,0,0,6,0,0,0,6,0])	вернулись назад по 2 правилу
⊢(5.1.1)	(q,7,S/(S–E,[15,0,0,6,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,8,S/(S–E),[0,15,0,0,6,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(S–E),[0,15,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.2)	(b,7,S/(S–E,[15,0,0,6,0,0,0,6,0])	возврат по символу ”)”
⊢(5.1.3)	(q,8,S/(S–b),[0,0,0,6,0,0,0,6,0])	вернулись по 15 прав., сдвиг
⊢(4)	(b,8,S/(S–b),[0,0,0,6,0,0,0,6,0])	перешли к возврату
⊢(5.2)	(b,7,S/(S–b,[0,0,6,0,0,0,6,0])	возврат по символу ”)”
⊢(5.2)	(b,6,S/(S–,[0,6,0,0,0,6,0])	возврат по символу ”b”
⊢(5.2)	(b,5,S/(S,[6,0,0,0,6,0])	возврат по символу ”–”
⊢(5.1.1)	(q,5,S/(T,[11,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,6,S/(T–,[0,11,0,0,0,6,0])	перенесли символ ”–”
⊢(2)	(q,7,S/(T–b,[0,0,11,0,0,0,6,0])	перенесли символ ”b”
⊢(1)	(q,7,S/(T–S,[7,0,0,11,0,0,0,6,0])	выполнили свёртку по №7
⊢(2)	(q,8,S/(T–S),[0,7,0,0,11,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(T–S),[0,7,0,0,11,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(T–S,[7,0,0,11,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(q,7,S/(T–T,[12,0,0,11,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,8,S/(T–T),[0,12,0,0,11,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(T–T),[0,12,0,0,11,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(T–T,[12,0,0,11,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(q,7,S/(T–E,[15,0,0,11,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,8,S/(T–E),[0,15,0,0,11,0,0,0,6,0])	перенесли символ ”)”
⊢(5)	(b,8,S/(T–E),[0,15,0,0,11,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(T–E,[15,0,0,11,0,0,0,6,0])	возврат по символу ”)”
⊢(5.1.3)	(q,8,S/(T–b),[0,0,0,11,0,0,0,6,0])	
⊢(4)	(b,8,S/(T–b),[0,0,0,11,0,0,0,6,0])	
⊢(5)	(b,7,S/(T–b,[0,0,11,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(b,6,S/(T–,[0,11,0,0,0,6,0])	возврат по символу ”b”
⊢(5)	(b,5,S/(T,[11,0,0,0,6,0])	возврат по символу ”–”
⊢(5)	(q,5,S/(E,[14,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,6,S/(E–,[0,14,0,0,0,6,0])	перенесли символ ”–”
⊢(2)	(q,7,S/(E–b,[0,0,14,0,0,0,6,0])	перенесли символ ”b”

⊢(1)	(q,7,S/(E–S,[7,0,0,14,0,0,0,6,0])	выполнили свёртку по №7
⊢(2)	(q,8,S/(E–S),[0,7,0,0,14,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(E–S),[0,7,0,0,14,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(E–S,[7,0,0,14,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(q,7,S/(E–T,[12,0,0,14,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,8,S/(E–T),[0,12,0,0,14,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(E–T),[0,12,0,0,14,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(E–T,[12,0,0,14,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(q,7,S/(E–E,[15,0,0,14,0,0,0,6,0])	заменили на другое правило
⊢(2)	(q,8,S/(E–E),[0,15,0,0,14,0,0,0,6,0])	перенесли символ ”)”
⊢(4)	(b,8,S/(E–E),[0,15,0,0,14,0,0,0,6,0])	перешли к возврату
⊢(5)	(b,7,S/(E–E,[15,0,0,14,0,0,0,6,0])	возврат по символу ”)”
⊢(5.1.3)	(q,8,S/(E–b),[0,0,0,14,0,0,0,6,0])	
⊢(4)	(b,8,S/(E–b),[0,0,0,14,0,0,0,6,0])	
⊢(5)	(b,7,S/(E–b,[0,0,14,0,0,0,6,0])	возврат по символу ”)”
⊢(5)	(b,6,S/(E–,[0,14,0,0,0,6,0])	возврат по символу ”b”
⊢(5)	(b,5,S/(E,[14,0,0,0,6,0])	возврат по символу ”–”
⊢(5)	(q,6,S/(a–,[0,0,0,0,6,0])	
	дальше снова с b и свёртки…	…17 шагов…
⊢(5)	(b,5,S/(a,[0,0,0,6,0])	больше нет правил с ‘a’ справа
⊢(5)	(b,4,S/(,[0,0,6,0])	возврат по символу ”(”
⊢(5)	(b,3,S/,[0,6,0])	возврат по символу ”/”
⊢(5)	(b,2,S,[6,0])	вернулись назад по 6 правилу
⊢(5)	(q,2,T,[11,0])	заменили на другое правило
⊢(2)	(q,3,T/,[0,11,0])	перенесли символ ”/”
⊢(2)	(q,4,T/(,[0,0,11,0])	перенесли символ ”(”
⊢(2)	(q,5,T/(a,[0,0,0,11,0])	перенесли символ ”a”
⊢(1)	(q,5,T/(S,[6,0,0,0,11,0])	выполнили свёртку по №6
⊢(2)	(q,6,T/(S–,[0,6,0,0,0,11,0])	перенесли символ ”–”
⊢(2)	(q,7,T/(S–b,[0,0,6,0,0,0,11,0])	перенесли символ ”b”
⊢(1)	(q,7,T/(S–S,[7,0,0,6,0,0,0,11,0])	выполнили свёртку по №7
⊢(2)	(q,8,T/(S–S),[0,7,0,0,6,0,0,0,11,0])	перенесли символ ”)”
⊢(4)	(b,8,T/(S–S),[0,7,0,0,6,0,0,0,11,0])	перешли к возврату
⊢(5)	(b,7,T/(S–S,[7,0,0,6,0,0,0,11,0])	возврат по символу ”)”
⊢(5)	(q,7,T/(S–T,[12,0,0,6,0,0,0,11,0])	заменили на другое правило
⊢(1)	(q,7,T/(S,[2,12,0,0,6,0,0,0,11,0])	выполнили свёртку по №2
⊢(2)	(q,8,T/(S),[0,2,12,0,0,6,0,0,0,11,0])	перенесли символ ”)”
⊢(1)	(q,8,T/S,[5,0,2,12,0,0,6,0,0,0,11,0])	выполнили свёртку по №5
⊢(4)	(b,8,T/S,[5,0,2,12,0,0,6,0,0,0,11,0])	перешли к возврату
⊢(5)	(q,8,T/T,[10,0,2,12,0,0,6,0,0,0,11,0])	заменили на другое правило
⊢(4)	(b,8,T/T,[10,0,2,12,0,0,6,0,0,0,11,0])	перешли к возврату
⊢(5)	(q,8,T/E,[13,0,2,12,0,0,6,0,0,0,11,0])	заменили на другое правило
⊢(1)	(q,8,S,[4,13,0,2,12,0,0,6,0,0,0,11,0])	выполнили свёртку по №4
⊢(3)	stop +	

Алгоритм успешно завершён, в стеке возврата содержатся номера правил: L_2=[4,13,0,2,12,0,0,6,0,0,0,11,0]=[4,13,2,12,6,11]. Тогда цепочка вывода имеет вид: S⇒T/E⇒T/(S)⇒T/(S–T)⇒T/(S–b)⇒T/(a–b)⇒a/(a–b). Дерево вывода строится аналогично построенному ранее.

Преимущества и недостатки рассмотренного алгоритма аналогичны тем, которые мы видели в методе нисходящего разбора с возвратами: это также просто реализуемый универсальный алгоритм, время работы которого экспоненциально зависит от длины входной цепочки.

Как и алгоритм нисходящего распознавателя с возвратами, сам по себе алгоритм «сдвиг-свёртка» не используется в компиляторах, но его основные принципы лежат в основе многих восходящих распознавателей, строящих правосторонние выводы и работающих без возвратов.

Оба рассмотренных распознавателя имеют приблизительно одинаковые показатели. Выбор того или иного алгоритма для реализации простейшего распознавателя зависит от грамматики языка.

3.4.2 Табличные распознаватели КС-языков

Табличные распознаватели КС-языков основаны на иных принципах, нежели МП-автоматы. Они также получают на вход цепочку входных символов $\alpha=a_1a_2\ldots a_n \mid \alpha\in VT^*$, $|\alpha|=n$, а построение вывода основывается на правилах заданной КС-грамматики. Но цепочка вывода строится не сразу – сначала на основе входной цепочки порождается промежуточная таблица (или некое другое хранилище информации) размера n×n, а уже потом на её основе строится вывод.

Алгоритмы этого класса обладают полиномиальными характеристиками. Для произвольной КС-грамматики время выполнения алгоритма имеет кубическую, а требуемый объём памяти – квадратичную зависимость от длины входной цепочки: $T=O(n^3)$, $M=O(n^2)$.

Так же, как и алгоритмы с возвратами, табличные распознаватели универсальны – они могут использоваться для распознавания цепочек языка, задаваемого произвольной КС-грамматикой, хотя, быть может, её и требуется предварительно привести к некоторому заранее определённому виду. Табличные распознаватели являются самыми эффективными универсальными алгоритмами с точки зрения используемых вычислительных ресурсов. Примерами алгоритмов этого класса являются алгоритм Кока-Янгера-Касами и алгоритм Эрли.

Алгоритм Кока-Янгера-Касами

Для построения вывода по рассматриваемому алгоритму грамматика должна находиться в нормальной форме Хомского, следовательно, произвольную грамматику следует сначала преобразовать к БНФ.

Работа алгоритма состоит из двух этапов. На первом этапе по заданной цепочке символов строится таблица, на втором с помощью этой

таблицы осуществляется разбор цепочки. Процесс построения таблицы состоит из трёх вложенных циклов, чем и определяется кубическая зависимость времени работы алгоритма от длины входной цепочки. Итак:

Этап 1. Для заданной грамматики ***G(VT,VN,P,S)*** и цепочки символов $\alpha=a_1a_2\ldots a_n$, $\alpha\in VT^*$, $|\alpha|=n$ алгоритм строит треугольную таблицу T_{n*n} состоящую из нетерминальных символов и такую, что $\forall A\in VN$ $A\in T_{i,j}$ тогда и только тогда, когда $A\Rightarrow^+ a_ia_{i+1}\ldots a_{i+j-1}$. Например, существованию вывода $S\Rightarrow^+\alpha$ соответствует условие $S\in T_{1,n}$.

Рассмотрим алгоритм построения таблицы:

Шаг 1 Первый столбец таблицы:

В $T_{i,1}$ включаются все нетерминальные символы, для которых в грамматике G существует правило $A\to a_i$, т.е. $T_{i,1}=\{A\,|\,\exists(A\to a_i)\in P\}$ $\forall i=1,\ldots,n$. Очевидно: $A\in T_{i,1}\Leftrightarrow A\Rightarrow^+a_i$.

Шаг 2 Пусть вычислены $T_{i,k}$ $\forall i=1,\ldots,n$, $1\le k<j$. Тогда остальные столбцы $T_{i,j}=\{A\,|\,\exists k:\ 1\le k<j\ |\ (A\to BC)\in P,\ B\in T_{i,k},\ C\in T_{i+k,j-k}\}$.

После этого шага $A\in T_{i,j}\Leftrightarrow A\Rightarrow BC\Rightarrow^+ a_i\ldots a_{i+k-1}C\Rightarrow^+ a_i\ldots a_{i+k-1}a_{i+k}\ldots a_{i+k+j-k-1}=a_i\ldots a_{i+j-1}$.

Шаг 3 Повторять шаг 2 до тех пор, пока не найдем все $T_{i,j}$ $\forall i,j:\ 1\le i\le n$, $1\le j\le n-i+1$.

Результатом работы данного алгоритма является таблица T. Для проверки существования вывода исходной цепочки в заданной грамматике остается проверить условие $S\in T_{1,n}$.

Пример: Рассмотрим грамматику G в БНФ: G({a,b},{S,A},P,S), P ={S → AA | AS | b; A → SA | AS | a}.

Входная цепочка α='abaab', т.е. n=5. Построим таблицу $T_{i,j}$.

В первой колонке в i-й строке будут находиться те нетерминальные символы, из которых выводится i-й символ входной цепочки.

Для второго столбца (j=2): $\forall i\ 1\le i<4$ $T_{i,2}=\{X\,|\,\exists k:\ 1\le k<2\,|\,(X\to BC)\in P,\ B\in T_{i,k},\ C\in T_{i+k,2-k}\}$. Очевидно, что в таком случае единственно возможное k=1, следовательно, $B\in T_{i,1}$, а $C\in T_{i+1,1}$, т.е. B и C – нетерминалы из первого столбца, из i-й и i+1-й строк.

Далее: $\forall i\ 1\le i<3$ $T_{i,3}=\{X\,|\,\exists k:\ 1\le k<3\,|\,(X\to BC)\in P,\ B\in T_{i,k},\ C\in T_{i+k,3-k}\}$, т.е. k=1 или k=2 и либо $B\in T_{i,1}$, $C\in T_{i+1,2}$, либо $B\in T_{i,2}$, $C\in T_{i+2,1}$. Аналогично: $\forall i=1,2$ $T_{i,4}=\{X\,|\,\exists k:\ 1\le k<4\,|\,(X\to BC)\in P, B\in T_{i,k}, C\in T_{i+k,4-k}\}$, т.е. k=1, или k=2, или k=3. Либо $B\in T_{i,1}$, $C\in T_{i+1,3}$, либо $B\in T_{i,2}$, $C\in T_{i+2,2}$, либо $B\in T_{i,3}$, $C\in T_{i+3,1}\ldots$ В итоге получим таблицу T:

$T_{i,j}$:					
	A	A,S	A,S	A,S	A,S
	S	A	S	A,S	
	A	S	A,S		
	A	A,S			
	S				

Проверка показывает, что целевой символ грамматики $S\in T_{1,5}$, следовательно, цепочка α выводима в грамматике G.

Этап 2. Построение цепочки вывода.

Если вывод существует, то для получения цепочки вывода имеется специальная рекурсивная процедура R. Она выдаёт последовательность номеров правил, которые надо применить, чтобы получить цепочку вывода. Процедура R порождает левый разбор, соответствующий выводу $A \Rightarrow^{+} a_i a_{i+1} \ldots a_{i+j-1}$.

Опишем эту процедуру R(i,j,A), $A \in VN$:

1. Если j=1 и существует правило $(A \to a_i,) \in P$, то выдать номер правила.
2. Если j>1, то возьмем k ($1 \le k < j$) как наименьшее из чисел, для которых существует правило $\exists\ (A \to BC) \in P$, $B \in T_{i,k}$, $C \in T_{i+k,j-k}$ (таких правил может быть несколько). Пусть правило $A \to BC$ имеет номер m. Тогда нужно выдать этот номер m, а затем вызвать сначала R(i,k,B), потом R(i+k,j–k,C).

Для получения цепочки вывода нужно вызвать R(1,n,S).

На основании полученной последовательности номеров правил строится левосторонний вывод для заданной грамматики G и входной цепочки α. $\Rightarrow$ Данный алгоритм позволяет решить задачу разбора.

Вернёмся к нашему примеру. Поскольку для построения вывода нужно будет выдавать номера правил, удобнее переписать все правила грамматики G по отдельности, перенумеровав их:

Пример: Запишем грамматику G в виде:

(1) $S \to AA$ (4) $A \to SA$
(2) $S \to AS$ (5) $A \to AS$
(3) $S \to b$ (6) $A \to a$

Для получения всей последовательности номеров правил вывода нужно вызвать R(1,n,S), т.е. в нашем случае R(1,5,S).

1) R(1,5,S): i=1, j=5. Поскольку j>1, нужно взять такое минимальное k, что существует правило $\exists\ (S \to BC) \in P$, $B \in T_{1,k}$, $C \in T_{1+k,5-k}$. Раз k должно быть минимальным, начинаем проверку с наименьшего из возможных: k=1, т.е. должно $\exists\ (S \to BC) \in P$, $B \in T_{1,1}$='A', $C \in T_{2,4}$='A,S'. Такое правило есть: $S \to AA$ (либо $S \to AS$), его номер 1 (либо 2). Условимся брать первое по порядку подходящее правило. Тогда нужно выдать его номер **1**, а затем вызвать сначала R(1,1,A), потом R(2,4,A).

2) R(1,1,A): i=1, j=1. $\exists\ (A \to a_1$='a')$\in P$, это правило с №**6**.

3) R(2,4,A): i=2, j=4. Найдем минимальное k, такое что $\exists\ (A \to BC) \in P$, $B \in T_{2,k}$, $C \in T_{2+k,4-k}$. Проверим k=1, т.е. должно $\exists\ (A \to BC) \in P$, $B \in T_{2,1}$='S', $C \in T_{3,3}$='A,S'. Такое правило есть: $A \to SA$, его номер 4. Тогда нужно выдать этот номер **4**, а затем вызвать сначала R(2,1,S), потом R(3,3,A).

4) R(2,1,S): i=2, j=1. $\exists\ (S \to a_2$='b')$\in P$, это правило с №**3**.

5) R(3,3,A): i=3, j=3. Найдем минимальное k, такое что $\exists\ (A \to BC) \in P$, $B \in T_{3,k}$, $C \in T_{3+k,3-k}$. Возможно только k=1 или 2. Проверим k=1, т.е. должно $\exists\ (A \to BC) \in P$, $B \in T_{3,1}$='A', $C \in T_{4,2}$='A,S'. Такое правило есть: $A \to AS$,

его номер 5. Тогда нужно выдать этот номер **5**, а затем вызвать сначала R(3,1,A), потом R(4,2,S).

6) R(3,1,A): i=3, j=1. $\exists$ $(A \to a_3 = 'a') \in P$, это правило с №**6**.

7) R(4,2,S): i=4, j=2. Найдем минимальное k, такое что $\exists$ $(S \to BC) \in P$, $B \in T_{4,k}$, $C \in T_{4+k,2-k}$. Единственно возможно k=1, т.е. должно $\exists$ $(S \to BC) \in P$, $B \in T_{4,1} = 'A'$, $C \in T_{5,1} = 'S'$. Такое правило есть: $S \to AS$, его номер 2. Тогда нужно выдать этот номер **2**, а затем вызвать сначала R(4,1,A), потом R(5,1,S).

8) R(4,1,A): i=4, j=1. $\exists$ $(A \to a_4 = 'a') \in P$, это правило с №**6**.

9) R(5,1,S): i=5, j=1. $\exists$ $(S \to a_5 = 'b') \in P$, это правило с №**3**.

Процесс закончился. Получили последовательность использованных правил: 1,6,4,3,5,6,2,6,3. Построим вывод: $S \Rightarrow^{(1)} AA \Rightarrow^{(6)} aA \Rightarrow^{(4)} aSA \Rightarrow^{(3)} \Rightarrow abA \Rightarrow^{(5)} abAS \Rightarrow^{(6)} abaS \Rightarrow^{(2)} abaAS \Rightarrow^{(6)} abaaS \Rightarrow^{(3)} abaab$.

Поскольку рассмотренная грамматика не является однозначной, можно было выбирать другие номера правил и, соответственно, цепочка вывода могла получиться иной.

Алгоритм Эрли

Для заданной грамматики ***G(VT,VN,P,S)*** и цепочки символов $\alpha = a_1 a_2 \ldots a_n$, $\alpha \in VT^*$, $|\alpha| = n$ алгоритм строит последовательность «списков ситуаций» $I_0, I_1, \ldots, I_n$, которая организована несколько сложнее, чем простая таблица. Каждая ситуация, входящая в список I_j для входной цепочки α, представляет собой структуру специального вида. На основании полученного списка ситуаций можно построить всю цепочку вывода и получить номера применяемых правил.

В нашем курсе данный алгоритм подробно рассматриваться не будет.

Алгоритм Эрли также является универсальным и имеет полиномиальную сложность, как и предыдущий рассмотренный алгоритм. Для произвольной КС-грамматики он имеет такие же оценки. Для однозначной КС-грамматики время его работы имеет квадратичную зависимость, а для некоторых более узких классов его характеристики ещё улучшаются. В целом он обладает лучшими характеристиками среди всех универсальных алгоритмов, хотя и более сложен в реализации.

3.4.3 Распознаватели КС-языков без возвратов – основные принципы

Рассмотренные выше распознаватели КС-языков универсальны, но имеют неудовлетворительные характеристики; в то же время для языков программирования нужнее эффективность работы, чем универсальность. Как правило, компилятор имеет дело с языком, принадлежащим к какому-то более узкому классу. Например, грамматика синтаксических конструкций языков программирования должна быть однозначной, значит, она относится к классу детерминированных КС-языков. Для такого языка

удается построить распознаватель, имеющий определённые ограничения, но обладающий лучшими характеристиками, чем универсальный.

Проблема преобразования КС-грамматик алгоритмически неразрешима, т.е. процесс приведения грамматики к заданному виду не формализован и требует участия человека. Если какая-то грамматика не принадлежит к требуемому классу, то возможно, что её удастся привести к нему путём преобразований.

Существуют два принципиально различающихся класса распознавателей – нисходящие, порождающие цепочки левостороннего вывода и строящие дерево сверху вниз, и восходящие, порождающие цепочки правостороннего вывода и строящие дерево снизу вверх. Считывание входной цепочки обычно осуществляется слева направо.

Нисходящие распознаватели основаны на алгоритме с подбором альтернатив. В них используются методы, позволяющие однозначно выбрать ту или иную альтернативу на каждом шаге работы МПА.

Восходящие распознаватели основаны на алгоритме «сдвиг-свёртка». В них используются методы, позволяющие однозначно выбрать между выполнением переноса или свёртки на каждом шаге работы расширенного МПА, а в случае свёртки однозначно выбрать необходимое правило.

3.4.4 Контрольные вопросы

1. Какие существуют модели поведения распознавателей КС-языков?
2. В чём различие по трудоёмкости распознавателя с возвратами и распознавателя с параллельным выполнением копий алгоритма?
3. Каковы ограничения на правила грамматики для применения алгоритма нисходящего разбора с возвратами? Какова их причина?
4. Какого типа грамматики допускают разбор цепочек с помощью алгоритма нисходящего разбора с возвратами?
5. Каковы начальная и заключительная конфигурации алгоритма распознавателя с подбором альтернатив?
6. В какой конфигурации должен оказаться алгоритм распознавателя с подбором альтернатив, чтобы был сделан вывод о недопустимости рассмотренной им цепочки?
7. Какие решения должны приниматься на каждом шаге работы распознавателя с подбором альтернатив?
8. Какого типа грамматики допускают разбор цепочек с помощью алгоритма «сдвиг-свёртка»?
9. Каковы ограничения на правила грамматики для применения алгоритма «сдвиг-свёртка»? Какова их причина?
10. Каковы начальная и заключительная конфигурации алгоритма распознавателя «сдвиг-свёртка»?
11. В какой конфигурации должен оказаться алгоритм распознавателя «сдвиг-свёртка», чтобы был сделан вывод о недопустимости

рассмотренной им цепочки?

12. Какие решения должны приниматься на каждом шаге работы распознавателя «сдвиг-свёртка»?
13. Почему в алгоритме восходящего разбора «сдвиг-свёртка» используется расширенный МПА?
14. Какой вид правил грамматики требуется для табличного распознавателя Кока-Янгера-Касами?
15. Какова трудоёмкость алгоритма табличного распознавателя?

3.5 Специальные классы КС-языков и грамматик

3.5.1 Нисходящие распознаватели без возвратов

Основная возможность улучшения алгоритма нисходящего разбора состоит в том, чтобы на каждом шаге работы однозначно выбирать одну из всего множества возможных альтернатив. В таком случае возвратов на предыдущие шаги не требуется, и количество выполняемых шагов алгоритма будет иметь линейную зависимость от длины входной цепочки. Если алгоритм не заканчивается успешно, то входная цепочка не принимается, повторных итераций не производится.

3.5.1.1 Метод рекурсивного спуска

Самым простым способом является выбор одной из множества альтернатив на основании текущего входного символа. Необходимо найти альтернативу, где этот символ присутствует в начале цепочки, находящейся в правой части правила грамматики – на этом принципе основан метод рекурсивного спуска.

Алгоритм разбора по методу рекурсивного спуска

Для каждого нетерминального символа исходной грамматики на основании правил строится процедура разбора, на вход которой подаётся цепочка символов $\boldsymbol{\alpha}$ и положение считывающей головки в этой цепочке $\boldsymbol{i}$. Если для символа $A \in VN$ в грамматике задано несколько правил, то при разборе выбирается то правило, в котором первый (терминальный) символ правой части совпадает с текущим входным символом: $\alpha_i = a$, $(A \to a\beta) \in P$, $a \in VT$, $\beta \in (VN \cup VT)^*$. Если такого правила нет, цепочка не принимается. Если оно есть или для символа A в грамматике существует единственное правило $A \to \beta$, то фиксируется номер правила. Если $\alpha_i = a$ в найденном правиле, то считывающая головка передвигается – увеличивается номер i, а для каждого нетерминального символа из цепочки β рекурсивно вызывается процедура его разбора.

Для начала разбора входной цепочки необходимо вызвать процедуру разбора для символа S с параметром i=1. Если в итоге разбора входной цепочки считывающая головка остановится на символе с номером n+1, то разбор закончен, цепочка принята, а выданная последовательность

номеров правил представляет собой цепочку вывода.

Данный метод накладывает жёсткие ограничения на правила задающей язык грамматики. Для каждого нетерминального символа A грамматики G разрешаются только два вида правил:

1) $(A \to \beta) \in P$, $\beta \in (VN \cup VT)^*$, и это единственное правило для A;
2) $A \to a_1\beta_1 \mid a_2\beta_2 \mid \ldots \mid a_n\beta_n$, $\forall i\ a_i \in VT$, $\beta \in (VN \cup VT)^*$, и $a_i \neq a_j$, если $i \neq j$, т.е. все a_i различны.

Таким образом, для каждого нетерминала либо должно существовать единственное правило вывода, либо – если их несколько – все правые части правил вывода должны начинаться с разных терминальных символов. Таким условиям удовлетворяет незначительное количество реальных грамматик. Возможно, что грамматика может быть приведена к требуемому виду путём некоторых преобразований.

Проблема приведения произвольной грамматики к указанному виду алгоритмически неразрешима. Возможно только привести некоторые рекомендации, которые могут способствовать приведению грамматики к заданному виду (но не гарантируют этого).

1. Исключение пустых правил.
2. Исключение левой рекурсии.
3. Добавление новых нетерминальных символов. Например, если существует правило $A \to a\alpha_1 \mid a\alpha_2 \mid \ldots \mid a\alpha_n \mid b_1\beta_1 \mid b_2\beta_2 \mid \ldots \mid b_m\beta_m$, то заменяем его на два: $A \to aA' \mid b_1\beta_1 \mid b_2\beta_2 \mid \ldots \mid b_m\beta_m$, $A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$.
4. Замена нетерминальных символов в правилах на цепочки их выводов. Например, если есть правила
$A \to B_1 \mid B_2 \mid \ldots \mid B_n \mid b_1\beta_1 \mid b_2\beta_2 \mid \ldots \mid b_m\beta_m$,
$B_1 \to \alpha_{11} \mid \alpha_{12} \mid \ldots \mid \alpha_{1k}$,
…
$B_n \to \alpha_{n1} \mid \alpha_{n2} \mid \ldots \mid \alpha_{nk}$, то заменяем первое правило на
$A \to \alpha_{11} \mid \alpha_{12} \mid \ldots \mid \alpha_{1k} \mid \ldots \mid \alpha_{n1} \mid \alpha_{n2} \mid \ldots \mid \alpha_{nk} \mid b_1\beta_1 \mid b_2\beta_2 \mid \ldots \mid b_m\beta_m$

Алгоритм рекурсивного спуска эффективен и прост в реализации, но имеет очень ограниченную применимость.

Рассмотрим пример.

Дана грамматика G({a,b,c},{A,B,C,S},P,S), где правила P имеют вид:

$S \to aA$ (1)$|bB$ (2)

$A \to a$ (3)$|\ bA$ (4)$|\ cC$ (5)

$B \to b$ (6)$|\ aB$ (7)$|\ cC$ (8)

$C \to AaBb$ (9)

Правила грамматики удовлетворяют требованиям метода рекурсивного спуска. Построим для каждого нетерминала процедуру разбора. Будем использовать псевдокод.

```
Процедура Rule(num);
   Записать в вывод номер правила num;
Процедура Proc_S(str,k); {входная строка и номер текущего символа}
  Выбор str[k] из:
  a: Rule(1);
     вернуть Proc_A(str,k+1);
  b: Rule(2);
     вернуть Proc_B(str,k+1);
  иначе вернуть 0;
Процедура Proc_A(str,k);
 Выбор str[k] из:
 a: Rule(3);
    вернуть k+1;
 b: Rule(4);
    вернуть Proc_A(str,k+1);
 c: Rule(5);
    вернуть Proc_C(str,k+1);
 иначе вернуть 0;
Процедура Proc_C(str,k);
  Rule(9);
  i:=Proc_A(str,k);
  если i=0 вернуть 0;
  если str[i]≠'a' вернуть 0;
  i:=Proc_B(str,i+1);
  если i=0 вернуть 0;
  если str[i]≠'b' вернуть 0;
  вернуть i+1;
Процедура Proc_B(str,k);
 Выбор str[k] из:
 a: Rule(7);
    вернуть Proc_B(str,k+1);
 b: Rule(6);
    вернуть k+1;
 c: Rule(8);
    вернуть Proc_C(str,k+1);
 иначе вернуть 0;
```

Пусть цепочка α = ‘acbaabb’.

Разбор начинается с вызова Proc_S(α,1). Дальнейшие вызовы и вывод номеров использованных правил будут происходить в следующем порядке:

Proc_S(α,1) (символ ‘a’, вывод **1**) $\Rightarrow$ Proc_A(α,2) (‘c’, вывод **5**) $\Rightarrow$ Proc_C(α,3) (вывод **9**) $\Rightarrow$ Proc_A(α,3) (‘b’, вывод **4**) $\Rightarrow$ Proc_A(α,4) (‘a’, вывод **3**, в Proc_C верн. i=5) $\Rightarrow$ Proc_B(α,6) (‘b’, вывод **6**, в Proc_C верн. i=7) $\Rightarrow$ вернули 8=n+1 $\Rightarrow$ stop(+). Правила вывода 1, 5, 9, 4, 3, 6.

$S \Rightarrow_{(1)} aA \Rightarrow_{(5)} acC \Rightarrow_{(9)} acAaBb \Rightarrow_{(4)} acbAaBb \Rightarrow_{(3)} acbaaBb \Rightarrow_{(6)} acbaabb$.

Метод рекурсивного спуска позволяет выбирать альтернативу на основе текущего символа входной цепочки. Если же имеется возможность

обозревать не один, а несколько символов вперед от текущего положения входной головки, то область применения этого метода может быть расширена. Тогда можно искать подходящие правила на основе некоторого терминального символа, входящего в правую часть правила (т.е. этот символ не обязан быть первым). Выбор и в таком случае должен осуществляться однозначно, т.е. для каждого нетерминального символа левой части необходимо, чтобы в правой части разных правил не встречалось двух одинаковых терминальных символов.

Поскольку один и тот же терминальный символ может встречаться во входной цепочке неоднократно, то в зависимости от типа рекурсии требуется искать либо его крайнее левое, либо крайнее правое вхождение, т.е. метод требует анализа типа использованной в грамматике рекурсии.

Для применения метода рекурсивного спуска требуется осуществлять неформальный анализ правил исходной грамматики G. При наличии большого количества правил такой анализ практически нереализуем, поэтому данный метод хотя и нагляден, но слабо применим.

Существуют распознаватели, основанные на строго формализованном подходе. Подготовка данных (правил грамматики) может быть строго формализована и автоматизирована.

3.5.1.2 Разбор для LL(k)-грамматик

Логическим продолжением идеи рекурсивного спуска является попытка использовать для выбора единственной из множества альтернатив не один, а несколько символов входной цепочки. Сложность заключается в том, что эти несколько соседних символов цепочки могут быть получены с применением не одного, а нескольких правил.

Грамматика *обладает свойством LL(k)* (называется *LL(k)-грамматикой*) для k>0, если на каждом шаге вывода для однозначного выбора очередной альтернативы автомату с магазинной памятью необходимо знать один верхний символ стека и рассмотреть k символов входной цепочки справа от положения считывающей головки.

Существуют LL(1), LL(2), LL(3), … грамматики. Все они в совокупности образуют класс LL-грамматик. В этом обозначении (LL) первая L означает, что входная цепочка считывается в направлении слева направо, а вторая L – что выполняется левосторонний разбор. Число k показывает, сколько символов справа от считывающей головки нужно рассмотреть для однозначного выбора альтернативы.

Алгоритм разбора входных цепочек для LL(k)-грамматик называется *k-предсказывающим алгоритмом*.

Свойства LL(k)-грамматик.

- Всякая LL(k)-грамматика для любого k>0 является однозначной.
- Существует алгоритм проверки, является ли произвольная КС-

грамматика LL(k)-грамматикой для строго определённого числа k.

- Всякая грамматика, допускающая разбор по методу рекурсивного спуска, является LL(1)-грамматикой. Обратное не справедливо.

Проблемы LL(k)-грамматик:

- Не существует алгоритма, позволяющего проверить, является ли произвольная КС-грамматика LL(k)-грамматикой для любого числа k.
- Не существует алгоритма преобразования произвольной КС-грамматики к виду LL(k)-грамматики для некоторого k (либо доказывающего, что такое преобразование невозможно).

Для LL(k)-грамматик для любого k>1 не обязательно все k символов должны находиться в одной цепочке в правой части правила вывода. Если это так, то такая грамматика называется *сильно LL(k)-грамматикой*. Но обычно эти символы находятся в правых частях разных правил.

Особенности правил грамматики класса LL(1):

1) В правилах грамматики для одного нетерминального символа не может существовать двух или более правил с одинаковым первым терминальным символом в правой части.
2) В отличие от метода рекурсивного спуска допускаются правила вида $A \to B\alpha$ и пустые правила.
3) Правила грамматики не должны содержать левой рекурсии.

LL-грамматики позволяют построить распознаватели с линейной трудоемкостью.

Для построения распознавателей LL(k)-грамматик используются два специальных множества, определяемых следующим образом:

- ***FIRST(k,α)*** – множество терминальных цепочек, выводимых из $\alpha \in (VT \cup VN)^*$ и укороченных до k символов. Формально $FIRST(k,\alpha) = \{w \in VT^* \mid |w| \le k$ и $\alpha \Rightarrow^* w$ или $\alpha \Rightarrow^* wx$, $x \in (VT \cup VN)^*\}$, $\alpha \in (VT \cup VN)^*$, $k>0$.
- ***FOLLOW(k,A)*** – множество укороченных до k символов терминальных цепочек, которые могут непосредственно следовать за $A \in VN$ в цепочках вывода. Формально: для $A \in VN$ и $k > 0$ $FOLLOW(k,A) = \{w \in VT^* \mid S \Rightarrow^* \alpha A \gamma$ и $w \in FIRST(k,\gamma)$, $\alpha \in VT^*\}$.

Очевидно, что, если имеется цепочка терминальных символов $\alpha \in VT^*$, то $FIRST(k,\alpha)$ – это первые k символов этой цепочки.

Доказано, что грамматика G(VT,VN,P,S) является LL(k)-грамматикой тогда и только тогда, когда выполняется условие: $\forall (A \to \beta) \in P$ и $\forall (A \to \gamma) \in P$, $\beta \neq \gamma$ $FIRST(k,\beta w) \cap FIRST(k,\gamma w) = \varnothing$ для всех цепочек w таких, что $S \Rightarrow^* \alpha A w$.

На основе этих двух множеств строится k-предсказывающий алгоритм для МПА $R(\{q\},VT,V,\delta,q,S,\{q\})$, где $V=VT \cup VN$, S – целевой символ

грамматики G, а функция переходов автомата строится на основе управляющей таблицы M, которую строят на базе правил грамматики.

Таблица M для LL(k) (k>0) отображает множество $(V\cup\{\lambda\})\times VT^{*k}$ (последнее обозначает цепочки длины не более k символов) на множество, состоящее из следующих элементов:

- ***пар вида (β, i)***, где β – цепочка символов, помещаемая автоматом на верхушку стека, i – номер правила: $(A\to\beta)\in P(i)$, $A\in VN$, $\beta\in V^*$;
- ***«выброс»;***
- ***«допуск»;***
- ***«ошибка».***

Автомат имеет два стека (второй – для записи последовательности правил). Поскольку состояние распознавателя МПА q единственно, можно его не упоминать в конфигурациях; тогда конфигурация такого распознавателя будет иметь вид (α, L_1, L_2) – непрочитанная часть входной цепочки и содержимое обоих стеков.

Пусть аванцепочка (т.е. первые k символов входной цепочки) обозначена через w: w=FIRST(k,α), остаток входной цепочки – через α, символ на верхушке стека – через x.

Тогда <u>алгоритм распознавания</u> (построение δ по M) содержит шаги:

- $(\alpha, x\gamma, \mu)\vdash(\alpha, \beta\gamma, \mu\ i)$, $x\in VN$, $\gamma\in V^*$; если $M(x,w)=(\beta, i)$;
- $(a\alpha', x\gamma, \mu)\vdash(\alpha', \gamma, \mu)$, если $x=a\in VT$, $\alpha=a\alpha'$, $M(a,w)=$ «выброс»;
- (λ,λ,μ) – завершение работы (с положительным результатом), если $M(\lambda,\lambda)=$ «допуск»;
- иначе завершение с ошибкой.

Рассмотрим класс LL(k)-грамматик на примере <u>LL(1)-грамматик</u>.

Алгоритм работы распознавателя для LL(1)-грамматик на вход получает текущий терминальный символ цепочки «a» и верхний символ стека. Возможны различные варианты работы распознавателя.

<u>1.</u> Построение таблицы M, описанной выше, и распознавание с помощью рассмотренного алгоритма. При этом построение таблицы упрощается, таблица отображает множество $(V\cup\{\lambda\})\times(VT\cup\{\lambda\})$ на множество, состоящее из описанных элементов.

Построение таблицы M для LL(1):

1) $\forall A\in VN$, $\forall a\in VT$, если $(A\to\beta)\in P(i)$ и $a\in FIRST(1,\beta)$: $M(A,a)=(\beta,i)$; $\forall b\in FOLLOW(1,A)$: если $\lambda\in FIRST(1,\beta)$, то $M(A,b)=(\beta,i)$;
2) $\forall a\in VT$: $M(a,a)=$ «выброс»;
3) $M(\lambda,\lambda)=$ «допуск»;
4) для всех остальных $M(x,a)=$«ошибка»: $\forall x\in(V\cup\{\lambda\})$, $a\in(VT\cup\{\lambda\})$.

<u>2.</u> Распознавание без предварительного построения таблицы; рассмотренный алгоритм модифицируется с учетом того, что аванцепочка

состоит не более, чем из одного символа. В этом случае алгоритм можно описать так:

1) Если на верхушке стека находится нетерминальный символ A, то алгоритм должен выбрать альтернативу, для чего и проверяет два условия:

- Если $a \in FIRST(1,\beta)$, то в качестве альтернативы выбирается правило $A \to \beta$.
- Если $a \in FOLLOW(1,A)$, то в качестве альтернативы выбирается правило $A \to \lambda$.

Если не выполняется ни одно из этих условий, то цепочка не принадлежит языку, о чём выдается соответствующее сообщение.

2) Если на верхушке стека находится терминальный символ «a», то выполняется шаг «выброса», на котором работа алгоритма остается без изменений – при совпадении символа «a» с текущим символом цепочки символ из стека удаляется, а считывающая головка смещается вправо на одну позицию. В противном случае цепочка не принимается.

Первый из рассмотренных вариантов распознавания более предпочтителен в случаях, когда требуется разбирать большое количество цепочек. Второй вариант не требует дополнительной памяти для размещения таблицы, но несколько медленнее осуществляет разбор.

Для того чтобы определить, относится ли грамматика к типу LL(1), необходимо и достаточно проверить условие: для каждого нетерминального символа, для которого существует более одного правила вида $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ должно выполняться требование: $\forall\ i \neq j,\ n \geq i,j > 0$ $FIRST(1,\alpha_i FOLLOW(1,A)) \cap FIRST(1,\alpha_j FOLLOW(1,A)) = \varnothing$.

Если для символа A нет пустого правила, то это условие очевидным образом вырождается в стандартную проверку отсутствия пересечений множеств $FIRST(1, \alpha_i)$ для различных α_i.

Построение множества $FIRST(1,\alpha)$ выполняется очевидным образом, если цепочка α начинается с терминального символа. Иначе, если в α первый символ нетерминальный (т.е. $\alpha = Ax$, $x \in (VT \cup VN)^*$, $A \in VN$), то $FIRST(1,\alpha) = FIRST(1,A)$. После построения множества FIRST(1,A) строится FOLLOW(1,A).

1) Алгоритм построения множества FIRST(1,A):

Сначала требуется устранить из множества правил исходной грамматики пустые правила. Затем для всех нетерминальных символов полученной грамматики строятся множества FIRST(1,A). При построении используется метод последовательного приближения.

Шаг 1. Для всех нетерминалов $A \in VN$: $FIRST_0(1,A) = \{X \mid (A \to X\alpha) \in P,\ X \in (VT \cup VN),\ \alpha \in (VT \cup VN)^*\}$ – т.е. для каждого нетерминального символа A во множество заносим все символы, стоящие в начале правых частей правил для этого нетерминала; i:=0.

Шаг 2. Для всех $A \in VN$: $FIRST_{i+1}(1,A) = FIRST_i(1,A) \cup FIRST_i(1,B)$ для всех нетерминалов $B \in (FIRST_i(1,A) \cap VN)$ – если в $FIRST_i(1,A)$ есть нетерминальные символы B, то добавляем к нему $FIRST_i(1,B)$.

Шаг 3. Если $\exists A \in VN$: $FIRST_{i+1}(1,A) \neq FIRST_i(1,A)$, то i:=i+1 и вернуться к шагу 2 (т.е. после предыдущего шага множество $FIRST_i(1,A)$ хотя бы для одного нетерминала изменилось), иначе перейти на шаг 4.

Шаг 4. Для всех $A \in VN$: $FIRST(1,A) = FIRST_i(1,A) \setminus VN$ (исключаем из построенных множеств все нетерминальные символы).

2) Алгоритм построения множества FOLLOW(1,A):

Множества FOLLOW(1,A) также строятся для всех нетерминальных символов грамматики методом последовательного приближения.

Шаг 1. Для всех $A \in VN$: $FOLLOW_0(1,A) = \{X \mid \exists (B \to \alpha A X \beta) \in P, B \in VN, X \in (VT \cup VN), \alpha,\beta \in (VT \cup VN)^*\}$. Т.е. первоначально для каждого нетерминала A во множество $FOLLOW_0(1,A)$ вносим те символы, которые стоят непосредственно за A в правых частях правил; i:=0.

Шаг 2. $FOLLOW_0(1,S) = FOLLOW_0(1,S) \cup \{\lambda\}$ – вносим пустую цепочку во множество последующих символов для нетерминала S, это означает, что в конце разбора за целевым символом цепочка кончается.

Шаг 3. Для $\forall A \in VN$: $FOLLOW'_i(1,A) = FOLLOW_i(1,A) \cup FIRST(1,B)$, для всех нетерминальных символов $B \in (FOLLOW_i(1,A) \cap VN)$.

Шаг 4. Для $\forall A \in VN$ и для всех нетерминальных символов $B \in (FOLLOW'_i(1,A) \cap VN)$, для которых существует правило $\exists (B \to \lambda) \in P$: $FOLLOW''_i(1,A) = FOLLOW'_i(1,A) \cup FOLLOW'_i(1,B)$.

Шаг 5. Для $\forall A \in VN$ и $\forall B \in VN$, если $\exists (B \to \alpha A) \in P$, $\alpha \in (VT \cup VN)^*$: $FOLLOW_{i+1}(1,A) = FOLLOW''_i(1,A) \cup FOLLOW''_i(1,B)$.

Шаг 6. Если $\exists A \in VN$: $FOLLOW_{i+1}(1,A) \neq FOLLOW_i(1,A)$ (т.е. на последнем шаге были изменения во множестве $FOLLOW_i(1,A)$), то i:=i+1 и вернуться на шаг 3, иначе перейти на следующий шаг.

Шаг 7. Для $\forall A \in VN$: $FOLLOW(1,A) = FOLLOW_i(1,A) \setminus VN$ – исключаем из построенных множеств все нетерминальные символы.

Пример. Рассмотрим нелеворекурсивную грамматику для построения арифметических выражений: G ({+,–,/,*,a,b,(,)}, {S,R,T,F,E}, P, S), где P:

$S \to T \mid TR$

$R \to +T \mid -T \mid +TR \mid -TR$

$T \to E \mid EF$

$F \to *E \mid /E \mid *EF \mid /EF$

$E \to (S) \mid a \mid b$

Очевидно, что эта грамматика не является LL(1)-грамматикой – например, для символов R и F имеется по два правила, начинающихся с одного и того же терминального символа. Но можно получить из данной

грамматики эквивалентную ей LL(1)-грамматику. Если преобразовать её к грамматике G′, добавив пустые правила, то получим P′:

$S \to TR$ *(1)*
$R \to +TR$ *(2)* $| -TR$ *(3)* $| \lambda$ *(4)*
$T \to EF$ *(5)*
$F \to *EF$ *(6)* $| /EF$ *(7)* $| \lambda$ *(8)*
$E \to (S)$ *(9)* $|$ a *(10)* $|$ b *(11)*

Полученная грамматика является LL(1)-грамматикой. Проверим это, построив для неё множества FIRST и FOLLOW. При этом для построения множества FIRST нужна грамматика без пустых правил, поэтому будем брать за основу правила P грамматики G, а для множества FOLLOW – правила P′ грамматики G′.

Множество FIRST («1» не будем писать для краткости):

Сначала i:=0;	i:=1:	i:=2:
$FIRST_0(S) = \{T\}$	$FIRST_1(S) = \{T,E\}$	$FIRST_2(S) = \{T,E, (,a,b\}$
$FIRST_0(R) = \{+,–\}$	$FIRST_1(R) = \{+,–\}$	$FIRST_2(R) = \{+,–\}$
$FIRST_0(T) = \{E\}$	$FIRST_1(T) = \{E, (,a,b\}$	$FIRST_2(T) = \{E, (,a,b\}$
$FIRST_0(F) = \{*, /\}$	$FIRST_1(F) = \{*, /\}$	$FIRST_2(F) = \{*, /\}$
$FIRST_0(E) = \{(, a, b\}$	$FIRST_1(E) = \{(,a,b\}$	$FIRST_2(E) = \{(,a,b\}$

Поскольку на последнем шаге новых нетерминалов ни в одно множество не добавилось, последующие изменения невозможны. Формально мы должны выполнить ещё одну итерацию, получив в итоге $FIRST_3(F) = FIRST_2(F)$. После удаления из построенных множеств нетерминальных символов получим:

FIRST(S) = { (, a, b }; FIRST(R) = {+, – };
FIRST(T) = { (, a, b }; FIRST(F) = {*, / };
FIRST(E) = { (, a, b }

Теперь построим множество FOLLOW (используем правила P′):

Сначала i:=0; шаг 1	Шаг 2	Шаг 3
$FOLLOW_0(S) = \{)\}$	$FOLLOW_0(S) = \{),\lambda\}$	$FOLLOW'_0(S) = \{),\lambda\}$
$FOLLOW_0(R) = \{\varnothing\}$	$FOLLOW_0(R) = \{\varnothing\}$	$FOLLOW'_0(R) = \{\varnothing\}$
$FOLLOW_0(T) = \{R\}$	$FOLLOW_0(T) = \{R\}$	$FOLLOW'_0(T) = \{R,+,–\}$
$FOLLOW_0(F) = \{\varnothing\}$	$FOLLOW_0(F) = \{\varnothing\}$	$FOLLOW'_0(F) = \{\varnothing\}$
$FOLLOW_0(E) = \{F\}$	$FOLLOW_0(E) = \{F\}$	$FOLLOW'_0(E) = \{F,*,/\}$

Шаг 4. В построенных множествах содержатся только нетерминалы R и F. Хотя для них и существуют пустые правила в P, но в силу того, что множества FOLLOW для R и F пустые, ничего нового не добавится и $FOLLOW_0'' = FOLLOW_0'$ для всех нетерминалов.

Шаг 5. Проанализируем правила для $\forall A \in VN$ на наличие $(B \to \alpha A) \in P$

$FOLLOW_1(S) = \{),\lambda\}$ Т.к. нет правил вида $(B \to \alpha S)$– нет добавлений
$FOLLOW_1(R) = \{),\lambda\}$ Т.к. $\exists\ (S \to \alpha R) \in P$, добавили $FOLLOW''_0(S)$
$FOLLOW_1(T) = \{R,+,–\}$ Т.к. нет $(B \to \alpha T) \in P$ – нет добавлений
$FOLLOW_1(F) = \{R,+,–\}$ Т.к. $\exists\ (T \to \alpha F) \in P$, добавили $FOLLOW''_0(T)$
$FOLLOW_1(E) = \{F,*,/\ \}$ Т.к. нет $(B \to \alpha E) \in P$ – нет добавлений

i:=1; Шаг 3. Для $\forall\ A \in VN\ \forall\ B \in FOLLOW_1(A)$ нужно добавить FIRST(B)
$FOLLOW'_1(S) = \{),\lambda\}$ Нет нетерминалов $B \in FOLLOW_1(S)$
$FOLLOW'_1(R) = \{),\lambda\}$ Нет нетерминалов $B \in FOLLOW_1(R)$
$FOLLOW'_1(T) = \{R,+,–,\}$ Нужно добавить FIRST(R) – нет изменений
$FOLLOW'_1(F) = \{R,+,–\}$ Нужно добавить FIRST(R) – нет изменений
$FOLLOW'_1(E) = \{F,*,/\ \}$ Нужно добавить FIRST(F) – нет изменений

Шаг 4. Для $\forall\ A \in VN$ и $\forall\ B \in FOLLOW'_1(A)$ проверим на наличие $B \to \lambda$
$FOLLOW''_1(S) = \{),\lambda\}$ Нет нетерминалов $B \in FOLLOW_1(S)$
$FOLLOW''_1(R) = \{),\lambda\}$ Нет нетерминалов $B \in FOLLOW_1(R)$
$FOLLOW''_1(T) = \{R,+,–,), \lambda\}$ $\exists\ (R \to \lambda) \in P \Rightarrow$ добавили $FOLLOW'_1(R)$
$FOLLOW''_1(F) = \{R,+,–,), \lambda\}$ $\exists\ (R \to \lambda) \in P \Rightarrow$ добавили $FOLLOW'_1(R)$
$FOLLOW''_1(E) = \{F,*,/, R,+,–\}$ $\exists\ (F \to \lambda) \in P \Rightarrow$ добавили $FOLLOW'_1(F)$

Шаг 5. Проанализируем правила для $\forall\ A \in VN$ на наличие $(B \to \alpha A) \in P$
$FOLLOW_2(S) = \{),\lambda\}$ нет $(B \to \alpha S) \in P$
$FOLLOW_2(R) = \{),\lambda\}$ $\exists\ (S \to \alpha R) \in P \Rightarrow FOLLOW''_1(S)$ было
$FOLLOW_2(T) = \{R,+,–,), \lambda\}$ нет $(B \to \alpha T) \in P$
$FOLLOW_2(F) = \{R,+,–,), \lambda\}$ $\exists\ (T \to \alpha F) \in P$, но нет изменений
$FOLLOW_2(E) = \{F,*,/, R,+,–\}$ Т.к. нет $(B \to \alpha E) \in P$ – нет добавлений

i:=2; Шаг 3. Новый нетерминал появился только в $FOLLOW_2(E)$
$FOLLOW'_2(S) = \{),\lambda\}$
$FOLLOW'_2(R) = \{),\lambda\}$
$FOLLOW'_2(T) = \{R,+,–,), \lambda\}$
$FOLLOW'_2(F) = \{R,+,–,), \lambda\}$
$FOLLOW'_2(E) = \{F,*,/, R,+,–\}$ добавили FIRST(R) = {+, – } – нет измен.

Шаг 4. Проверка на пустые правила – новый нетерминал только для E
$FOLLOW''_2(S) = \{),\lambda\}$
$FOLLOW''_2(R) = \{),\lambda\}$
$FOLLOW''_2(T) = \{R,+,–,), \lambda\}$
$FOLLOW''_2(F) = \{R,+,–,), \lambda\}$
$FOLLOW''_2(E) = \{F,*,/, R,+,–,),\lambda\}$ $\exists\ (R \to \lambda) \in P$, добавили $FOLLOW'_1(R)$

Шаг 5.

$FOLLOW_3(S) = \{),\lambda\}$

$FOLLOW_3(R) = \{),\lambda\}$

$FOLLOW_3(T) = \{R,+,–,), \lambda\}$

$FOLLOW_3(F) = \{R,+,–,), \lambda\}$

$FOLLOW_3(E) = \{F,*,/, R,+,–,),\lambda\}$

Поскольку новых нетерминалов относительно построенного ранее $FOLLOW''_1$ не появилось – множество $FOLLOW_3$ совпадает с $FOLLOW''_2$

Множество $FOLLOW_3$ отличается от $FOLLOW_2$ только терминальными символами. Очевидно, что после выполнения ещё одной итерации согласно алгоритму получим $FOLLOW_4 = FOLLOW_3$. При выполнении 7 шага исключаются все нетерминальные символы. В итоге построенные множества сведём в одну таблицу для удобства пользования:

$A \in VN$	FIRST(A)	FOLLOW(A)
S	{ (, a, b }	{), λ}
R	{+, – }	{), λ}
T	{ (, a, b }	{+, –,), λ}
F	{*, / }	{+, –,), λ}
E	{ (, a, b }	{*,/, +, –,), λ}

Теперь рассмотрим для нашего примера оба варианта распознавания цепочек – с помощью таблицы M и без неё.

Начнём с построения таблицы M. Строки таблицы озаглавлены символами $V \cup \{\lambda\}$, столбцы – символами $VT \cup \{\lambda\}$. Построение таблицы было описано раньше; оно выполняется на основании множеств FIRST и FOLLOW и правил грамматики.

Например, для M(S,’a’): должно быть ‘a’$\in$FIRST(1,β); где $(S \to \beta) \in P(i)$, тогда M(S,’a’)=(β,i); при этом i=1, β = TR. Или для M(S,’)’): ‘)’$\notin$FIRST(1,TR); ‘)’$\in$FOLLOW(1,S), но для единственного правила вывода для S $(S \to TR) \in P(1)$ $\lambda \notin$FIRST(1,TR) $\Rightarrow$ M(S,’)’)=’ошибка’. Для M(R,’)’): ‘)’$\notin$FIRST(1,β), где $(R \to \beta) \in P(i)$ (из R может выводиться только первый терминальный + или – или λ), ‘)’$\in$FOLLOW(1,R), для $(R \to \lambda) \in P(4)$, $\lambda \in$FIRST(1,β) $\Rightarrow$ M(R,’)’)=(λ,4). Остальные клетки таблицы заполняются аналогично. Ячейки таблицы, которые соответствуют ситуации «ошибка», оставлены пустыми.

Рассмотрим те же цепочки: $\alpha_1 = 'a+b'$ и $\alpha_2 = 'a/(a–b)'$.

	a	b	(	)	+	–	*	/	λ
S	TR,1	TR,1	TR,1						
R				λ,4	+TR,2	–TR,3			λ,4
T	EF,5	EF,5	EF,5						
F				λ,8	λ,8	λ,8	*EF,6	/EF,7	λ,8
E	a,10	b,11	(S),9						
a	выброс								
b		выброс							
(			выброс						
)				выброс					
+					выброс				
–						выброс			
*****							выброс		
/								выброс	
λ									допуск

Поскольку в данном автомате только одно состояние q, не будем его писать. Для построения вывода в дополнительный стек записываем номера правил. Таким образом, конфигурацию автомата на каждом шаге будем записывать в виде трёх компонент – оставшаяся непрочитанной цепочка, содержимое стека и список использованных правил.

Сначала выполним разбор с помощью таблицы, анализируя текущий символ цепочки ‘a’ и верхний символ стека ‘x’ и используя значение M(x,a). {a+b,S,[λ]}⇒{a+b,TR,[1]}⇒{a+b,EFR,[1,5]}⇒{a+b,aFR,[1,5,10]} ⇒ (выброс) ⇒ {+b,FR,[1,5,10]} ⇒ {+b,FR,[1,5,10]} ⇒ {+b,R,[1,5,10,8]} ⇒ {+b,+TR,[1,5,10,8,2]} ⇒ {b,TR,[1,5,10,8,2]} ⇒ {b,EFR,[1,5,10,8,2,5]} ⇒ {b,bFR,[1,5,10,8,2,5,11]} ⇒ {λ,FR,[1,5,10,8,2,5,11]} ⇒ {λ,R,[1,5,10,8,2,5,11,8]} ⇒ {λ,λ,[1,5,10,8,2,5,11,8,4]}.

Теперь выполним разбор этой же цепочки по второму варианту.

1	{a+b, S, λ}	a∈FIRST(TR)⇒ выбираем S → TR *(1)*
2	{a+b, TR, [1]}	a∈FIRST(EF) ⇒ выбираем T → EF *(5)*
3	{a+b, EFR, [1,5]}	a∈FIRST(a) ⇒ выбираем E → a *(10)*
4	{a+b, aFR, [1,5,10]}	«выброс» – убираем «a»
5	{+b, FR, [1,5,10]}	+∈FOLLOW(F) ⇒ выбираем F → λ *(8)*
6	{+b, R, [1,5,10,8]}	+∈FIRST(+TR) ⇒ выбираем R → +TR *(2)*
7	{+b, +TR, [1,5,10,8,2]}	«выброс» – убираем «+»
8	{b, TR, [1,5,10,8,2]}	b∈FIRST(EF) ⇒ выбираем T → EF *(5)*
9	{b, EFR, [1,5,10,8,2,5]}	b∈FIRST(b) ⇒ выбираем E → b *(11)*

10	{b, bFR, [1,5,10,8,2,5,11]}	«выброс» – убираем «b»
11	{λ, FR, [1,5,10,8,2,5,11,8]}	λ∈FOLLOW(F) ⇒ выбираем F → λ *(8)*
12	{λ, R, [1,5,10,8,2,5,11,8,4]}	λ∈FOLLOW(R) ⇒ выбираем R → λ *(4)*
13	{λ, λ, [1,5,10,8,2,5,11,8,4]}	цепочка разобрана, стек пуст.

Оба варианта дали один результат – последовательность правил.

Замечание. Ели для одного нетерминала (например, A) существует несколько правил, то прежде чем для каждого из правил A→β проверять для текущего терминального символа выполнение x∈FIRST(1,β) полезно проверить, выполняется ли x∈FIRST(1,A). Только в этом случае будет выполняться x∈FIRST(1,β) для одного из правил A→β.

Запишем цепочку вывода по полученной последовательности номеров правил: $S \Rightarrow^{(1)} TR \Rightarrow^{(5)} EFR \Rightarrow^{(10)} aFR \Rightarrow^{(8)} aR \Rightarrow^{(2)} a+TR \Rightarrow^{(5)} a+EFR \Rightarrow^{(11)} a+bFR \Rightarrow^{(8)} a+bR \Rightarrow^{(4)} a+b$. Разбор цепочки выполнен за 13 шагов.

Теперь рассмотрим разбор цепочки α_2= ′a/(a–b)′.

1	{a/(a-b), S, λ}	a∈FIRST(TR)⇒ выбираем S → TR *(1)*
2	{a/(a–b), TR, [1]}	a∈FIRST(EF) ⇒ выбираем T → EF *(5)*
3	{a/(a–b), EFR, [1,5]}	a∈FIRST(a) ⇒ выбираем E → a *(10)*
4	{a/(a–b), aFR, [1,5,10]}	«выброс» – убираем «a»
5	{/(a–b), FR, [1,5,10]}	/∈FIRST(/EF) ⇒ выбираем F → /EF *(7)*
6	{/(a–b), /EFR, [1,5,10,7]}	«выброс» – убираем «/»
7	{(a–b), EFR, [1,5,10,7]}	(∈FIRST((S)) ⇒ выбираем E → (S) *(9)*
8	{(a–b), (S)FR, [1,5,10,7,9]}	«выброс» – убираем «(»
9	{a–b), S)FR, [1,5,10,7,9]}	a∈FIRST(TR)⇒ выбираем S → TR *(1)*
10	{a–b), TR)FR, [1,5,10,7,9,1]}	a∈FIRST(EF) ⇒ выбираем T → EF *(5)*
11	{a–b), EFR)FR, [1,5,10,7,9,1,5]}	a∈FIRST(a) ⇒ выбираем E → a *(10)*
12	{a–b), aFR)FR, [1,5,10,7,9,1,5,10]}	«выброс» – убираем «a»
13	{–b), FR)FR, [1,5,10,7,9,1,5,10]}	–∈FOLLOW(F) ⇒ выбираем F → λ *(8)*
14	{–b), R)FR, [1,5,10,7,9,1,5,10,8]}	–∈FIRST(–TR)⇒выбираем R→–TR *(3)*
15	{–b),–TR)FR,[1,5,10,7,9,1,5,10,8,3]}	«выброс» – убираем «–»
16	{b),TR)FR,[1,5,10,7,9,1,5,10,8,3]}	b∈FIRST(EF) ⇒ T → EF *(5)*
17	{b),EFR)FR,[1,5,10,7,9,1,5,10,8,3,5]}	b∈FIRST(b) ⇒ E → b *(11)*
18	{b),bFR)FR,[1,5,10,7,9,1,5,10,8,3,5,11]}	«выброс» – убираем «b»
19	{),FR)FR,[1,5,10,7,9,1,5,10,8,3,5,11]}	)∈FOLLOW(F)⇒ F → λ *(8)*
20	{),R)FR,[1,5,10,7,9,1,5,10,8,3,5,11,8]}	)∈FOLLOW(R) ⇒ R → λ *(4)*
21	{),)FR,[1,5,10,7,9,1,5,10,8,3,5,11,8,4]}	«выброс» – убираем «)»
22	{λ,FR,[1,5,10,7,9,1,5,10,8,3,5,11,8,4]}	λ∈FOLLOW(F) ⇒ F → λ *(8)*
23	{λ,R,[1,5,10,7,9,1,5,10,8,3,5,11,8,4,8]}	λ∈FOLLOW(R) ⇒ R → λ *(4)*
24	{λ,λ,[1,5,10,7,9,1,5,10,8,3,5,11,8,4,8,4]}	цепочка разобрана, стек пуст.

Получена цепочка вывода :S $\Rightarrow^{(1)}$TR $\Rightarrow^{(5)}$EFR $\Rightarrow^{(10)}$aFR $\Rightarrow^{(7)}$a/EFR $\Rightarrow^{(9)}$ a/(S)FR $\Rightarrow^{(1)}$ a/(TR)FR $\Rightarrow^{(5)}$ a/(EFR)FR $\Rightarrow^{(10)}$ a/(aFR)FR $\Rightarrow^{(8)}$ a/(aR)FR$\Rightarrow^{(3)}$ a/(a–TR)FR $\Rightarrow^{(5)}$ a/(a–EFR)FR $\Rightarrow^{(11)}$ a/(a–bFR)FR $\Rightarrow^{(8)}$ a/(a–bR)FR $\Rightarrow^{(4)}$ a/(a–b)FR $\Rightarrow^{(8)}$ a/(a–b)R $\Rightarrow^{(4)}$ a/(a–b) .

Попробуем рассмотреть неправильную цепочку, например, (+a)*b.

1	{(+a)*b, S, λ}	(∈FIRST(S), (∈FIRST(TR)⇒выбираем S → TR *(1)*
2	{(+a)*b, TR, [1]}	(∈FIRST(T), (∈FIRST(EF)⇒выбираем T → EF *(5)*
3	{(+a)*b, EFR, [1,5]}	(∈FIRST(E), (∈FIRST((S))⇒выбираем E → (S) *(9)*
4	{(+a)*b, (S)FR, [1,5,9]}	«выброс» – убираем «(»
5	{+a)*b, S)FR, [1,5,9]}	Нет правил вида S → β | +∈FIRST(β), и +∉FOLLOW(S) ⇒ цепочка не принята

Заметим, что в случае использования таблицы (вариант разбора 1) остановка произошла бы в той же конфигурации, т.к. M(S,'+')='ошибка'.

Из рассмотренных примеров видно, что алгоритму разбора на основе LL(1)-грамматик требуется значительно меньше шагов на принятие решения относительно входной цепочки, чем ранее рассмотренным алгоритмам, работающим с возвратами.

Алгоритм является эффективным, только строгие ограничения на правила грамматики сужают возможности его применения.

3.5.2 Восходящие распознаватели без возвратов

Основная возможность улучшения алгоритма восходящего разбора также состоит в том, чтобы на каждом шаге работы однозначно принимать решение, что выполнять, сдвиг или свёртку, а также какие цепочку и правило выбирать для свёртки. В таком случае возвратов не выполняется, и количество проделанных шагов алгоритма имеет линейную зависимость от длины входной цепочки. Если алгоритм не заканчивается успешно, то входная цепочка не принимается, повторных итераций не производится.

3.5.2.1 LR(k)-грамматики

При моделировании восходящих распознавателей без возвратов может использоваться аналогичный подход, который был положен в основу определения LL(k)-грамматик.

КС-грамматика *обладает свойством LR(k),* k≥0, если на каждом шаге вывода для принятия однозначного решения по вопросу о выполняемом действии в алгоритме «сдвиг-свёртка» расширенному МПА достаточно знать содержимое верхней части стека и рассмотреть первые k символов от текущего положения считывающей головки автомата во входной цепочке символов.

Грамматика называется *LR(k)-грамматикой*, если она обладает свойством LR(k).

Обозначение грамматики LR(k) имеет смысл, аналогичный рассмотренной ранее аббревиатуре LL(k). Отличие состоит в символе R, который обозначает, что в результате работы распознавателя получается правосторонний вывод. Остальные символы имеют тот же смысл, что и в обозначении LL(k)-грамматики.

В совокупности все LR(k)-грамматики для различных $k \geq 0$ образуют класс LR-грамматик. Поскольку алгоритм восходящего распознавателя моделирует работу расширенного МПА, возможность $k=0$ не является абсурдом, т.к. в таком случае все равно автомат в процессе работы рассматривает цепочки символов на вершине стека и, следовательно, результат его работы зависит от входной цепочки (т.к. в стеке находится именно она).

Класс LR-грамматик является более широким, чем класс LL. Это объясняется тем, что на каждом шаге работы расширенного МПА обрабатывается больше информации, чем при работе обычного МПА. Существует и строгое доказательство этого факта. Вообще, для каждого языка, заданного LL-грамматикой, может быть построена LR-грамматика, задающая тот же язык (при этом значения k в них не обязаны совпадать).

Свойства LR(k)-грамматик.

- Всякая LR(k)-грамматика для любого $k \geq 0$ является однозначной.
- Существует алгоритм проверки, является ли заданная КС-грамматика LR(k)-грамматикой для строго определённого числа k.
- Класс LR-грамматик полностью совпадает с классом детерминированных КС-языков.

Проблемы LR(k)-грамматик:

- Не существует алгоритма, позволяющего проверить, является ли заданная КС-грамматика LR(k)-грамматикой для любого числа k.
- Не существует алгоритма преобразования произвольной КС-грамматики к виду LR(k)-грамматики для некоторого k (либо доказывающего, что такое преобразование невозможно).

Класс LR-грамматик удобен для построения распознавателей детерминированных КС-языков, следовательно, для распознавания языков программирования.

Для формального определения LR(k)-свойства для КС-грамматик, нужно ввести ещё одно определение.

Грамматика является *пополненной*, если её целевой символ не встречается нигде в правых частях правил. Для приведения произвольной КС-грамматики к такому виду необходимо к множеству правил добавить $S' \to S$, а S' сделать целевым символом.

Понятие пополненной грамматики введено для того, чтобы появление символа S' на вершине стека означало окончание работы алгоритма.

Формальное определение LR(k)-свойства.

Если для произвольной КС-грамматики G в её пополненной грамматике G′ для двух произвольных цепочек вывода из условий:

1. $S' \Rightarrow^* \alpha A \omega \Rightarrow \alpha\beta\omega$
2. $S' \Rightarrow^* \gamma B x \Rightarrow \alpha\beta y$
3. FIRST(k,w)= FIRST(k,y)

следует, что $\alpha A y = \gamma B x$, т.е. $\alpha = \gamma$, A = B, x = y, то доказано, что грамматика G обладает LR(k)-свойством (так же, как и её пополненная грамматика G′).

Распознаватель для LR(k)-грамматик основан на специальной управляющей таблице T. Эта таблица состоит из двух частей, называемых «действия» и «переходы».

По строкам таблицы распределены все цепочки символов на верхушке стека, которые могут приниматься во внимание в процессе работы распознавателя; по столбцам в части «действия»– все возможные части входной цепочки длины не более k символов, которые могут следовать за считывающей головкой в процессе выполнения разбора; в части «переходы» – все терминальные и нетерминальные символы, которые могут появляться на верхушке стека в процессе выполнения действий.

Клетки управляющей таблицы в части «действия» содержат информацию о необходимых в каждой ситуации действиях, в частности:

- «сдвиг», если требуется выполнение сдвига (переноса текущего символа из входной цепочки в стек);
- «успех», если возможна свёртка к целевому символу грамматики S и разбор завершен;
- целое число («свёртка»), если возможно выполнение свёртки (число обозначает номер правила грамматики, по которому должна выполняться свёртка);
- «ошибка» – во всех других ситуациях.

Клетки управляющей таблицы T в части «переходы» служат для определения номера строки таблицы, которая будет использоваться для выполнения действия на очередном шаге. Эти клетки содержат данные:

- целое число – номер строки таблицы T;
- «ошибка» – во всех других ситуациях.

Для удобства работы используются два дополнительных символа начала и конца цепочки: $\perp_н$ и $\perp_к$. Тогда в начальном состоянии работы распознавателя символ $\perp_н$ находится на верхушке стека, а считывающая головка обозревает первый символ входной цепочки. В конечном состоянии в стеке должны находиться целевой символ и $\perp_к$, а считывающая головка должна обозревать символ $\perp_к$.

Алгоритм работы распознавателя:

Шаг 1 Занести в стек символ начала цепочки $\perp_н$ и начальную (нулевую) строку управляющей таблицы T (её номер), в конец цепочки поместить

символ $\perp_к$.

Шаг 2 Прочитать с вершины стека строку управляющей таблицы T (её номер). Выбрать из неё часть «действие» в соответствии с аванцепочкой.

Шаг 3 В соответствии с типом действия:

- «сдвиг»: если это не $\perp_к$, то прочитать и запомнить как «новый символ» очередной символ входной цепочки, сдвинув считывающую головку вправо на 1 символ; иначе остановка с ошибкой;
- «целое число, свёртка»: выбрать соответствующее числу правило (пусть это $A \to \beta$), убрать из стека $2 \cdot |\beta|$ символов, запомнить A как «новый символ»;
- «ошибка»: остановка с ошибкой;
- «успех»: свернуть к S, если текущий символ $\perp_к$, то завершить с успехом, иначе завершить с ошибкой.

Шаг 4 Прочитать с вершины стека строку таблицы T (её номер). Выбрать часть «переход» в соответствии с «новым символом».

Шаг 5 Если «переход» содержит «ошибка», то остановка алгоритма с ошибкой. Иначе (если «переход» содержит номер) – в стек занести «новый символ» и строку таблицы (выбранный номер). Вернуться на шаг 2.

Рассмотрим пример для LR(0)-грамматики.

Дана грамматика G({a,b}, {S}, {S→aSS|b}, S).

Эта грамматика определяет язык, в цепочках которого количество символов ‘a’ на один меньше, чем ‘b’, первый символ всегда ‘a’, в конце всегда пара ‘bb’. Исключение – минимальная цепочка ‘b’.

Поскольку символ S входит в правую часть правил, необходимо преобразовать грамматику G в пополненную G′. Правила P′ перенумеруем.

G′({a,b},{S,S′},P′,S′);

P′: (1) S′→S; (2) S→aSS, (3) S→b.

№ строки	Стек	Действие	Переход		
			S	a	b
0	$\perp_н$	сдвиг	1	2	3
1	S	успех, 1			
2	a	сдвиг	4	2	3
3	b	свёртка, 3			
4	aS	сдвиг	5	2	3
5	aSS	свёртка, 2			

Клетки таблицы в графе «переход», в которых должна содержаться «ошибка», оставлены пустыми и закрашены. Поскольку состояние распознавателя единственно, в записи конфигурации его можно опустить.

Стек лучше заполнять слева направо (для получения естественного порядка правил), поскольку выполняется правосторонний разбор.

Рассмотрим две цепочки: α_1=′aabbb′; α_2=′aabb′.

(aabbb$\perp_к$,{$\perp_н$,0}, λ) ⊢ (abbb$\perp_к$,{$\perp_н$,0; a,2}, λ) ⊢ (bbb$\perp_к$,{$\perp_н$,0; a,2; a,2}, λ) ⊢ (bb$\perp_к$, {$\perp_н$,0; a,2; a,2; b,3}, λ) ⊢ (bb$\perp_к$, {$\perp_н$,0; a,2; a,2; S,4}, 3λ) ⊢ (b$\perp_к$, {$\perp_н$,0; a,2; a,2; S,4; b,3}, 3) ⊢ (b$\perp_к$, {$\perp_н$,0; a,2; a,2; S,4; S,5}, 33) ⊢ (b$\perp_к$, {$\perp_н$,0; a,2; S,4}, 233) ⊢ ($\perp_к$, {$\perp_н$,0; a,2; S,4; b,3}, 233)

$\vdash$ ($\perp_к$, {$\perp_н$,0; a,2; S,4; S,5}, 3233) $\vdash$ ($\perp_к$, {$\perp_н$,0; S,1}, 23233) $\vdash$ ($\perp_к$, {$\perp_н$,0; S′}, 123233) алгоритм завершён успешно, цепочка принята.

$S' \Rightarrow^{(1)} S \Rightarrow^{(2)} aSS \Rightarrow^{(3)} aSb \Rightarrow^{(2)} aaSSb \Rightarrow^{(3)} aaSbb \Rightarrow^{(3)} aabbb$.

(aabb$\perp_к$,{$\perp_н$,0}, λ) $\vdash$(abb$\perp_к$,{$\perp_н$,0; a,2}, λ) $\vdash$ (bb$\perp_к$,{$\perp_н$,0; a,2; a,2}, λ) $\vdash$ (b$\perp_к$, {$\perp_н$,0; a,2; a,2; b,3}, λ) $\vdash$ (b$\perp_к$, {$\perp_н$,0; a,2; a,2; S,4}, 3λ) $\vdash$ ($\perp_к$, {$\perp_н$,0; a,2; a,2; S,4; b,3}, 3) $\vdash$ ($\perp_к$, {$\perp_н$,0; a,2; a,2; S,4; S,5}, 33) $\vdash$ ($\perp_к$, {$\perp_н$,0; a,2; S,4}, 233) $\vdash$ алгоритм завершился с ошибкой.

На практике LR(k)-грамматики для k>1 не применяются: для любой LR(k)-грамматики можно построить эквивалентную ей LR(1)-грамматику, которая работает значительно эффективнее в силу меньших размеров управляющей таблицы.

3.5.2.2 Грамматики предшествования

Ещё один распространенный класс КС-грамматик, для которых возможно построить восходящий распознаватель без возвратов, представляют грамматики предшествования. Распознаватель для них строится также на основе алгоритма «сдвиг-свёртка».

Суть таких грамматик состоит в том, что для каждой упорядоченной пары символов в грамматике устанавливается некоторое отношение, называемое *отношением предшествования*. В процессе разбора входной цепочки распознаватель сравнивает текущий символ с одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из отношений предшествования существует между этими двумя символами, и в зависимости от найденного отношения выполняется либо сдвиг, либо свёртка. 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. При отсутствии какого-либо отношения выдается сигнал об ошибке.

Таким образом, задача состоит в том, чтобы определить отношения предшествования между символами грамматики. В случае удачи 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) эти отношения могут быть установлены.

Выделяют следующие типы грамматик предшествования:

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

Наиболее распространены первый и последний типы грамматик.

Грамматикой простого предшествования называют такую КС-грамматику G(VN,VT,P,S), в которой различные правила имеют разные правые части, и для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех заданных отношений предшествования.

Для примера более подробно рассмотрим грамматики операторного предшествования.

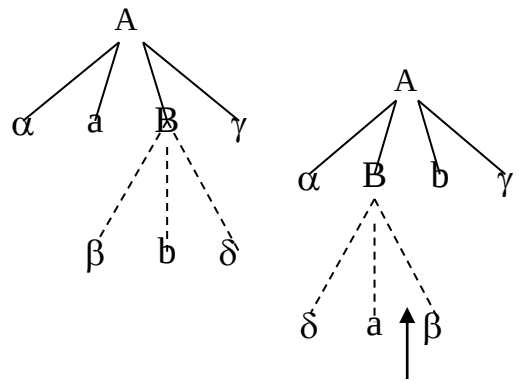
Грамматики, в которых все правила таковы, что в любой правой части никакие два нетерминала не являются смежными, и, следовательно, стоящий между ними терминал можно представить как оператор (хотя и не обязательно в арифметическом смысле), называются *операторными* грамматиками.

Пусть так же, как и в предыдущем разделе, кроме терминального алфавита VT имеются специальные символы $\{\perp_н, \perp_к\}$ (или $\{\vdash, \dashv\}$), которые ограничивают цепочку слева и справа соответственно.

Будем использовать следующие обозначения цепочек: $\beta \in VN \cup \{\lambda\}$ (т.е. нетерминал или пусто), $\alpha, \gamma, \delta \in V^*$ ($V=VT \cup VN$). Определим отношения предшествования $\{\doteq, <\cdot, \cdot>\}$ на множестве $VT \cup \{\perp_к, \perp_н\}$:

1. $\mathbf{a \doteq b}$ (a имеет такое же старшинство, как b), если $A \to \alpha a \beta b \gamma$;
2. $\mathbf{a <\cdot\ b}$ (a имеет меньшее старшинство, чем b), если $A \to \alpha a B \gamma$, $B \Rightarrow^+ \beta b\ \delta$;

Схематичное представление данного правила показано на первом рисунке.



3. $\mathbf{a \cdot> b}$ (a имеет большее старшинство, чем b), если $A \to \alpha B b \gamma$, $B \Rightarrow^+ \delta a \beta$;

Схематичное представление правила показано на втором рисунке.

4. $\perp_н <\cdot\ \mathbf{a}$, если $S \Rightarrow^+ \beta a \delta$;
5. $\mathbf{a \cdot>}\ \perp_к$, если $S \Rightarrow^+ \delta a \beta$.

Если между любыми двумя операторами из $VT \cup \{\perp_к, \perp_н\}$ возможно не более одного такого отношения, то соответствующая операторная грамматика называется грамматикой *операторного предшествования* (ОП)

Пример.

Дана грамматика построения АВ с операциями сложения и умножения и скобками: G({x,+,*,(,)}, {S,T,R}, P, S), где правила P: $S \to S+T_{(1)} | T_{(2)}$, $T \to T*R_{(3)} | R_{(4)}$, $R \to (S)_{(5)} | x_{(6)}$. Здесь вместо символа x может быть использовано любое целое число.

В соответствии с приведенными выше правилами определения отношений предшествования построим таблицу, в которую занесём отношения между всеми операторами данной грамматики:

	+	*	(	)	x	$\perp_к$
$\perp_н$	<·	<·	<·		<·	
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≝	<·	
)	·>	·>		·>		·>
x	·>	·>		·>		·>

Рассмотрим работу алгоритма разбора на базе грамматик ОП.

Цепочка α считается принятой, если за конечное число шагов произошел переход от начальной конфигурации к заключительной: $(\perp_н \alpha \perp_к, \lambda, \lambda) \vdash\!\!\!\vdash^* (\perp_к, \perp_н S, \mu) \Rightarrow stop(+)$.

Алгоритм:

1) Считывающая головка устанавливается на первый символ цепочки α_1, в стек заносится $\perp_н$, i=1.
2) Верхний символ стека (или ближний к верху стека терминальный символ) сравнивается с α_i.
3) Если отношение <· или ≝, то выполняется сдвиг, i++, возврат на шаг 2.
4) Если отношение ·>, то выполняется свёртка. При этом если нет подходящего правила, то алгоритм заканчивается с ошибкой, иначе основа (символы, связанные отношением ≝) удаляется из стека и сворачивается по найденному правилу. Далее возврат на шаг 2.
5) Если на шаге 2 отношение не найдено $\Rightarrow$ завершение с ошибкой.
6) Если получена заключительная конфигурация – цепочка разобрана.

Дополнительные требования к правилам грамматики: (1) среди них не должно быть пустых правил; (2) не должно быть правил с одинаковыми правыми частями.

Вернёмся к примеру. $S \to S+T \mid T,\ T \to T*R \mid R,\ R \to (S) \mid x$.

Сначала устраним цепные правила. Получим: $S \to S+T \mid T*R \mid (S) \mid x$, $T \to T*R \mid (S) \mid x$, $R \to (S) \mid x$. Можно избавиться от лишних нетерминалов и оставить только один нетерминальный символ S. Правила примут вид:

$S \to S+S_{(1)} \mid S*S_{(2)} \mid (S)_{(3)} \mid x_{(4)}$. Разбирается цепочка $\perp_н x*(x+x) \perp_к$.

$[x*(x+x) \perp_к, \perp_н, \lambda] \vdash\!\!\!\vdash \{\perp_н <\cdot x \Rightarrow$ сдвиг$\}$ $[*(x+x) \perp_к, \perp_н x, \lambda] \vdash\!\!\!\vdash \{x \cdot\!> * \Rightarrow$ свёртка$\}$ $[*(x+x) \perp_к, \perp_н S, 4\lambda] \vdash\!\!\!\vdash \{\perp_н <\cdot * \Rightarrow$ сдвиг$\}$ $[(x+x)\perp_к, \perp_н S*, 4] \vdash\!\!\!\vdash \{* <\cdot(\Rightarrow$ сдвиг$\}$ $[x+x)\perp_к, \perp_н S*(, 4] \vdash\!\!\!\vdash \{(<\cdot x \Rightarrow$ сдвиг$\}$ $[+x)\perp_к, \perp_н S*(x, 4] \vdash\!\!\!\vdash \{x\cdot\!> + \Rightarrow$ свёртка$\}$ $[+x)\perp_к, \perp_н S*(S, 44] \vdash\!\!\!\vdash \{(<\cdot + \Rightarrow$ сдвиг$\}$ $[x)\perp_к,$

$\perp_н$S*(S+, 44] ├─ {+<·x ⇒ сдвиг} []$\perp_к$, $\perp_н$S*(S+x, 44] ├─ {x ·>) ⇒ свёртка} []$\perp_к$, $\perp_н$S*(S+S, 444] ├─ {+ ·>) ⇒ свёртка} []$\perp_к$, $\perp_н$S*(S, 1444] ├─ {(≐,) ⇒ сдвиг} [$\perp_к$, $\perp_н$S*(S), 1444] ├─ {) ·>$\perp_к$ ⇒ свёртка} [$\perp_к$, $\perp_н$S*S, 21444] ├─ {* ·>$\perp_к$ ⇒ свёртка} [$\perp_к$, $\perp_н$S, 231444] ├─ stop (+).

Рассмотрим другой способ разбора. При вычислении значения выражения сначала выполняются операции с большим приоритетом.

В той же цепочке $\perp_н$ x*(x+x) $\perp_к$ подставим вместо символов x конкретные числа: $\perp_н$ 3*(2+7) $\perp_к$ – и запишем под ней знаки отношений:

$\perp_н$	3	*	(	2	+	7	)	$\perp_к$
	<·	·>	<·	<·	·>	<·	·>	·>

Рассмотрим процесс выполнения действий. Сначала будет выбрано из цепочки число 3 и сохранено в стеке. В результате останется:

$\perp_н$	*	(	2	+	7	)	$\perp_к$
	<·	<·	<·	·>	<·	·>	·>

$\perp_н$	*	(	)	$\perp_к$
	<·	<·	≐	·>

Теперь следует выбрать число 2 и сохранить его в стеке; останется:

$\perp_н$	*	(	+	7	)	$\perp_к$
	<·	<·	<·	<·	·>	·>

Теперь так же выбирается число 7 и сохраняется в стеке:

$\perp_н$	*	(	+	)	$\perp_к$
	<·	<·	<·	·>	·>

Старшей операцией осталось сложение; его нужно применить к двум верхним элементам стека, результат остается в стеке, а символ операции «+» удаляем из цепочки. В итоге получится:

$\perp_н$	*	$\perp_к$
	<·	·>

Скобки имеют одинаковое старшинство и просто отбрасываются:

Производится умножение двух верхних элементов стека, результат остается там же, а знак операции удаляется:

$\perp_н$	$\perp_к$

Отношения предшествования между оставшимися символами отсутствуют, поэтому происходит остановка. Поскольку вся цепочка разобрана, результат её выполнения находится в стеке.

Вместо выполнения арифметических операций можно было бы породить код и только после этого уже вычислять значение выражения. Именно это бы и выполнил компилятор.