Лекция 4

Синтаксический анализ

§17. Постановка задачи синтаксического анализа

Определение. Порождающую грамматику $G = \langle N, T, P, S \rangle$ называют *контекстно-свободной* (КС-грамматикой), если каждое правило вывода из множества P имеет вид
$A \to \gamma$,
где $A \in N$ – нетерминал, а $\gamma \in (N \cup T)^\star$ – цепочка в объединённом алфавите грамматики.

Будем говорить, что uxv выводится за один шаг из uAv (и записывать это как $uAv \Rightarrow uxv$), если $A \to x$ – правило вывода, и u и v – произвольные цепочки из $(N \cup T)^\star$.

Если $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n$, будем говорить, что из u_1 выводится u_n, и записывать это как $u_1 \Rightarrow^\star u_n$.

Для отношения $\Rightarrow^\star$ справедливы утверждения:
1. $u \Rightarrow^\star u$ для любой цепочки u;
2. если $u \Rightarrow^\star v$ и $v \Rightarrow^\star w$, то $u \Rightarrow^\star w$.

Аналогично, $\Rightarrow^+$ означает «выводится за один или более шагов».

Определение. Цепочка u называется *сентенциальной формой* в грамматике $G = \langle N, T, P, S \rangle$, если $S \Rightarrow^{\star} u$.

Определение. Сентенциальная форма u называется *левой сентенциальной формой*, если на каждом шаге вывода u из аксиомы грамматики S осуществляется подстановка самого левого нетерминала. При этом такой вывод называется *левосторонним*.

Аналогично, *правая сентенциальная форма* и *правосторонний вывод*.

Определение. *Предложение* – это сентенциальная форма, не содержащая нетерминалов.

Определение. *Упорядоченным графом* называется пара $\langle V, E\rangle$, где V обозначает множество вершин, а E – множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $\langle\langle x, y_1\rangle, \langle x, y_2\rangle, \ldots \langle x, y_n\rangle\rangle$.

Этот элемент указывает, что из вершины x выходят n дуг, причем первой из них считается дуга, входящая в вершину y_1, второй – дуга, входящая в вершину y_2, и т.д.

Определение. *Дерево вывода* в КС-грамматике $G = \langle N, T, P, S\rangle$ – это упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup T \cup \{\varepsilon\}$. При этом:

1. корень дерева помечен аксиомой S;
2. каждый лист помечен либо терминалом, либо ε;
3. каждая внутренняя вершина помечена нетерминалом;
4. если N – нетерминал, которым помечена нелистовая вершина, и $X_1, \ldots, X_n$ – метки её прямых потомков в указанном порядке, то $N \to X_1 \ldots X_n$ – правило из множества P.

Задача синтаксического анализа. Дано:

$G = \langle N, T, P, S \rangle$ – КС-грамматика языка;

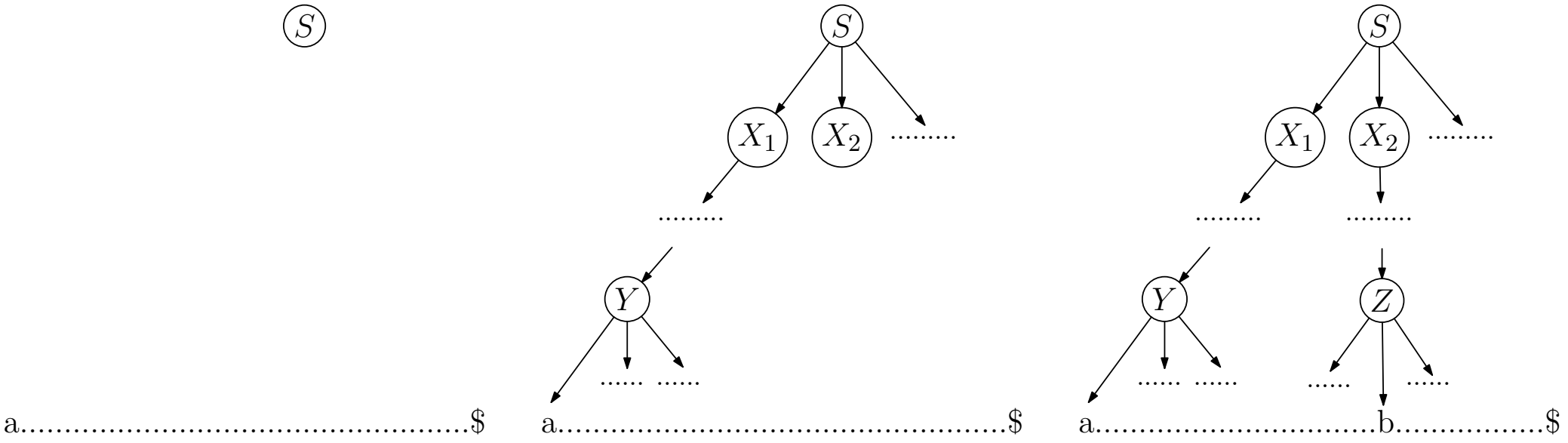
$u \in T^\star$ – цепочка.

Требуется определить, является ли цепочка u предложением в грамматике G. В случае положительного ответа на этот вопрос следует построить дерево вывода цепочки u в грамматике G.

Замечание. Следует понимать, что реальный синтаксический анализатор вовсе не обязан строить настоящее дерево вывода. Достаточно, чтобы он в процессе работы формировал некие данные, по которым можно построить дерево вывода.

§18. Алгоритм предсказывающего синтаксического разбора

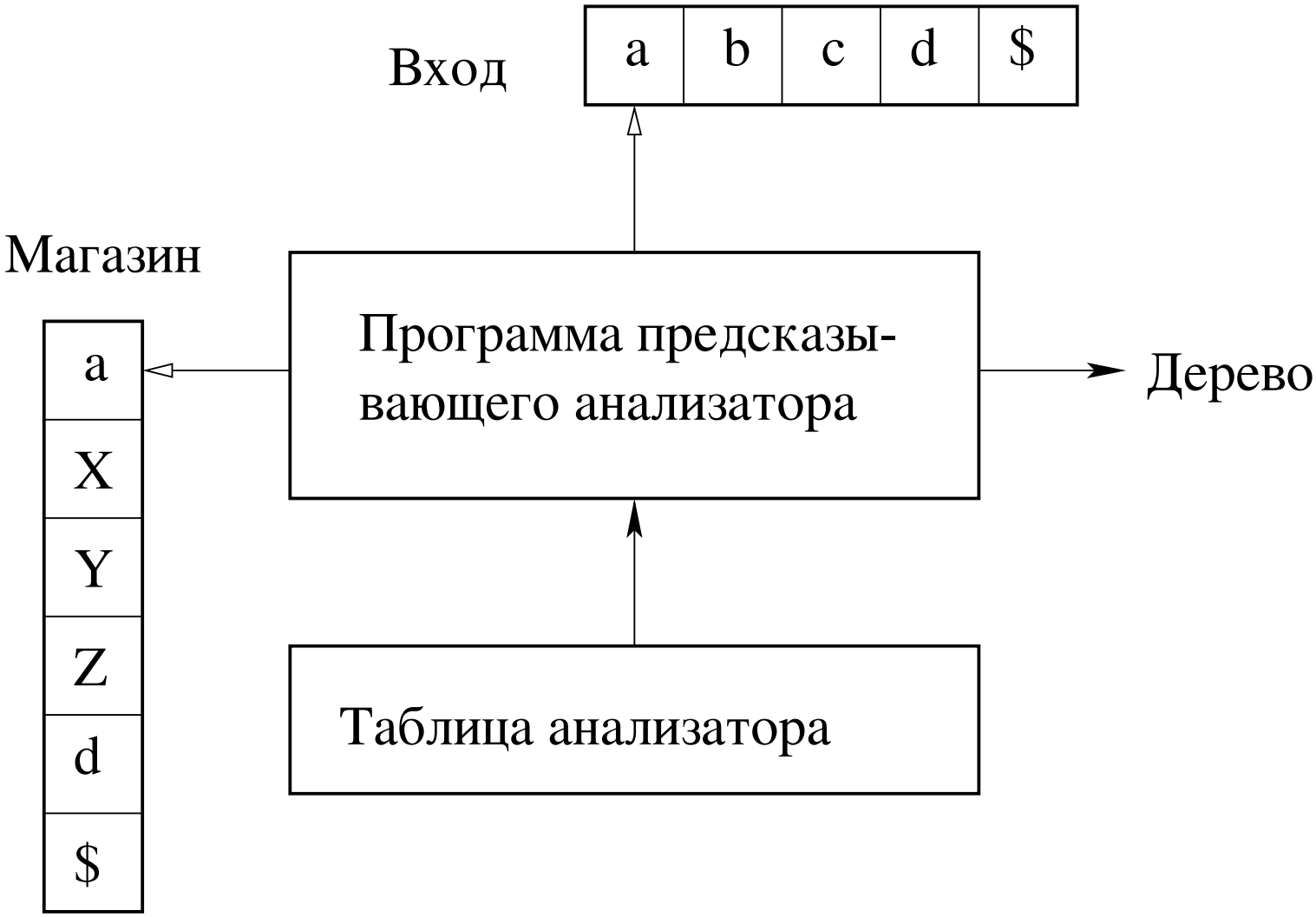
Последовательность формирования дерева вывода в процессе предсказывающего синтаксического разбора (сверху-вниз):



Здесь \$ – терминальный символ, обозначающий конец предложения (правый концевой маркер).

Основная проблема предсказывающего разбора – определение правила вывода, которое нужно применить к нетерминалу.

Структура предсказывающего анализатора:



Предсказывающий анализатор для грамматики $G = \langle N, T, P, S \rangle$ получает на вход цепочку $u \in T^\star$, в конце которой обязательно стоит правый концевой маркер \$.

Магазин содержит последовательность символов $v \in (N \cup T)^\star$ с символом \$ на дне. В начале магазин содержит аксиому грамматики S на верхушке и \$ на дне.

Пока будем считать, что на выходе анализатора получается последовательность применённых правил вывода.

Таблица анализатора задаёт функцию $\delta : N \times T \longrightarrow (N \cup T)^\star \cup \{\mathrm{ERROR}\}$. (В ячейках таблицы располагаются либо правые части правил грамматики, либо индикаторы ошибки.)

Программа предсказывающего анализатора работает следующим образом. Она рассматривает X – символ на верхушке магазина и a – текущий входной символ. При этом имеются три возможности.

1. Если $X = a =$ \$, анализатор останавливается и сообщает об успешном окончании разбора.

2. Если $X = a \neq$ \$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.

3. Если X – нетерминал, программа заглядывает в таблицу $\delta(X, a)$, где хранится либо правило для X, либо ошибка. Если, например, $\delta(X, a) = UVW$, анализатор заменяет X на верхушке магазина на WVU (U оказывается на верхушке) и отправляет на выход правило $X \to UVW$. Если $\delta(X, a) =$ ERROR, анализатор переходит в режим восстановления при синтаксических ошибках.

```
TopDownParse(N, T (содержит $), S, δ, цепочка u):
    Result ←пустая последовательность
    поместить в магазин $S
    a ←первый символ из u
    do:
        X ←верхний символ магазина
        if X ∈ T:
            if X = a:
                удалить X из магазина
                a ←очередной символ из u
            else:
                error()
        else if δ(X, a) = Y1Y2...Yk:
            удалить X из магазина
            поместить Yk...Y2Y1 в магазин (Y1 сверху)
            добавить правило X → Y1Y2...Yk в Result
        else:
            error()
    while X ≠$
    return Result
```

Пример. Объектные Рефал-выражения

$T = \{\text{symbol}, \text{'('}, \text{')'}, \$\}$, $N = \{\text{expr}, \text{term}\}$, $S = \text{expr}$,
$P =$

```
expr ::= term expr | ε.
term ::= symbol | '(' expr ')'.
```

Функция δ:

	symbol	'('	')'	$
expr	term expr	term expr	ε	ε
term	symbol	'(' expr ')'	ERROR	ERROR

§19. Множества FIRST и FOLLOW

Пусть дана грамматика $G = \langle N, T, P, S \rangle$. Будем обозначать нетерминальные символы грамматики буквами X и Y, терминальные символы – буквой a, а цепочки – буквами u и v.

Множества FIRST и FOLLOW связаны с грамматикой языка и позволяют построить таблицу предсказывающего разбора.

Определим FIRST(u) как множество терминалов, с которых начинаются цепочки, выводимые из u. Если $u \Rightarrow^\star \varepsilon$, то ε также принадлежит FIRST(u).

$$\text{FIRST}\,(u) = \{a \in T \mid u \Rightarrow^\star av\} \cup \{\varepsilon \mid u \Rightarrow^\star \varepsilon\}$$

Определим FOLLOW(X) как множество терминалов a таких, что существует вывод вида $S \Rightarrow^\star uXav$. Если X может быть самым правым символом некоторой сентенциальной формы, то \$ принадлежит FOLLOW(X).

$$\text{FOLLOW}\,(X) = \{a \in T \mid S\$ \Rightarrow^\star uXav\}$$

Определим функцию $\mathcal{F} : (N \cup T)^\star \longrightarrow 2^T$, работающую в предположении, что для каждого нетерминального символа грамматики определено некоторое множество FIRST, и вычисляющую множество FIRST для любой цепочки, то есть $\mathsf{FIRST}(u) = \mathcal{F}[\![u]\!]$.

$$
\begin{array}{lcl}
\mathcal{F}[\![au]\!] & = & \{a\} \\
\mathcal{F}[\![Xu]\!] & = & \begin{cases} \mathsf{FIRST}(X), & \text{если } \varepsilon \notin \mathsf{FIRST}(X) \\ (\mathsf{FIRST}(X) \setminus \{\varepsilon\}) \cup \mathcal{F}[\![u]\!], & \text{если } \varepsilon \in \mathsf{FIRST}(X) \end{cases} \\
\mathcal{F}[\![\varepsilon]\!] & = & \{\varepsilon\}
\end{array}
$$

Запишем алгоритм построения множеств FIRST для нетерминальных символов:

Шаг 1. Пусть $\mathsf{FIRST}(X) = \emptyset$ для любого $X \in N$.

Шаг 2. Для каждого правила $X \to u$ добавить $\mathcal{F}[\![u]\!]$ в $\mathsf{FIRST}(X)$.

Шаг 3. Повторять шаг 2 до тех пор, пока множества FIRST не перестанут изменяться.

Запишем алгоритм построения множеств FOLLOW для всех нетерминальных символов:

Шаг 1. Положить FOLLOW(X)=$\emptyset$ для любого $X \in N$.

Шаг 2. Поместить \$ в FOLLOW(S), где S – аксиома.

Шаг 3. Если есть правило вывода $X \to uYv$, то все символы из FIRST(v), за исключением ε, добавить к FOLLOW(Y).

Шаг 4. Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X): если есть правило вывода $X \to uY$ или $X \to uYv$, где FIRST(v) содержит ε (т.е. $v \Rightarrow^\star \varepsilon$), то все символы из FOLLOW(X) добавить к FOLLOW(Y).

Пример. Арифметические выражения

$T = \{\text{'+'}, \text{'*'}, \text{n}, \text{'('}, \text{')'}, \$\}$, $N = \{\text{E}, \text{E}', \text{T}, \text{T}', \text{F}\}$, $S = \text{E}$,
$P =$

$$
\begin{array}{lll}
\text{E} & ::= & \text{T}\ \ \text{E}'. \\
\text{E}' & ::= & \text{'+'}\ \ \text{T}\ \ \text{E}'\ \ |\ \ \varepsilon. \\
\text{T} & ::= & \text{F}\ \ \text{T}'. \\
\text{T}' & ::= & \text{'}*\text{'}\ \ \text{F}\ \ \text{T}'\ \ |\ \ \varepsilon. \\
\text{F} & ::= & \text{n}\ \ |\ \ \text{'('}\ \ \text{E}\ \ \text{')'}.
\end{array}
$$

Множества FIRST и FOLLOW:

FIRST(E) =FIRST(T)=FIRST(F)={'(',n}
FIRST(E′)={'+',ε}
FIRST(T′)={'*',ε}
FOLLOW(E)=FOLLOW(E′)={')',$}
FOLLOW(T)=FOLLOW(T′)={'+',')',$}
FOLLOW(F)={'+','*',')',$}

§20. LL(1)-грамматики

Определение. Грамматика G является LL(1) тогда и только тогда, когда для любых двух правил вида $A \to u \,|\, v$ выполняется следующее:

1. $\mathrm{FIRST}\,(u) \cap \mathrm{FIRST}\,(v) = \emptyset$;
(Поясняющий пример 1. Предположим, что $a \in FIRST\,(u) \cap FIRST\,(v)$. Тогда невозможно решить, по какому правилу раскрывать A.)

(Поясняющий пример 2. Предположим, что $\varepsilon \in FIRST\,(u) \cap FIRST\,(v)$ и $a \in FOLLOW(A)$. Тогда A нужно раскрыть, как ε, но непонятно, каким из двух правил при этом нужно воспользоваться.)

2. если $v \Rightarrow^\star \varepsilon$, то $\mathrm{FIRST}\,(u) \cap \mathrm{FOLLOW}\,(A) = \emptyset$.
(Поясняющий пример. Предположим, что $a \in FIRST\,(u) \cap FOLLOW(A)$. Тогда невозможно определить, то ли A надо раскрыть как u, то ли A раскрывается как ε, и символ a расположен после A.)

(Примечание. В поясняющих примерах A находится на верхушке магазина, a – текущий входной символ.)

Напомним, что программа предсказывающего разбора работает на основе таблицы, задаваемой функцией $\delta : N \times T \longrightarrow (N \cup T)^\star \cup \{\text{ERROR}\}$. Такая таблица может быть построена только для LL(1)-грамматик.

Основная трудность при использовании предсказывающего анализа – это составление для входного языка LL(1)-грамматики.

Грамматики языков программирования часто бывают «почти LL(1)», то есть отличаются от LL(1)-грамматики тем, что содержат левую рекурсию и нуждаются в левой факторизации.

Удаление левой рекурсии

Определение. КС-грамматика *леворекурсивна*, если в ней имеется нетерминал A такой, что существует вывод $A \Rightarrow^{+} Au$, где u – некоторая цепочка.

Мы не будем рассматривать алгоритм удаления левой рекурсии, ограничившись рассмотрением простейшего случая.

Пусть в грамматике есть правила $A \to Av_1 \,|\, Av_2 \,|\, \ldots \,|\, Av_n \,|\, u_1 \,|\, u_2 \,|\, \ldots \,|\, u_m$.

Тогда для удаления левой рекурсии выполним преобразование:
$A \to u_1 A' \,|\, u_2 A' \,|\, \ldots \,|\, u_m A'$.
$A' \to v_1 A' \,|\, v_2 A' \,|\, \ldots \,|\, v_n A' \,|\, \varepsilon$.

Левая факторизация

Пусть в грамматике имеются два правила для нетерминала A:
$A \to uv_1 \,|\, uv_2$.

Основная идея левой факторизации заключается в том, что, когда неясно, какую из двух альтернатив надо использовать для развертки нетерминала A, нужно переделать правила для A так, чтобы отложить решение до тех пор, пока не будет досточно информации, чтобы принять правильное решение.

Для левой факторизации надо преобразовать грамматику:
$A \to uA'$.
$A' \to v_1 \,|\, v_2$.

То есть если входная цепочка начинается с непустой цепочки, выводимой из u, то в исходной грамматике мы не знаем, разворачивать ли A по uv_1 или по uv_2. Однако в факторизованной грамматике можно отложить решение, развернув $A \to uA'$. Тогда после анализа того, что выводимо из u, можно развернуть $A' \to v_1$ или $A' \to v_2$.

§21. Построение таблиц предсказывающего анализатора

Алгоритм построения таблиц работает для любой КС-грамматики, так как реально строит функцию $\delta' : N \times T \longrightarrow 2^{(N \cup T)^\star} \cup \{\text{ERROR}\}$. То есть в ячейке таблицы в общем случае могут находиться несколько правил.

Алгоритм построения таблиц предсказывающего анализатора

Пусть для начала $\delta'(X, a) = \text{ERROR}$ для любых $X \in N$ и $a \in T$.

Затем проходим по всем правилам грамматики, и для каждого правила $X \to u$ выполняем следующее:

а) перебираем все $a \in \text{FIRST}(u)$ и добавляем u к $\delta'(X, a)$.
б) если $\varepsilon \in \text{FIRST}(u)$, перебираем все $b \in \text{FOLLOW}(X)$ и добавляем u к $\delta'(X, b)$.

Замечание. Добавление u к $\delta'(X, a)$ заключается в том, что если $\delta'(X, a) = \text{ERROR}$, то $\delta'(X, a)$ становится равным $\{u\}$, а если $\delta'(X, a) = A$, то $\delta'(X, a)$ становится равным $A \cup \{u\}$.

Пример. Грамматика арифметических выражений

$$
\begin{aligned}
E &::= T\ E'. \\
E' &::= \text{'+'}\ T\ E' \mid \varepsilon. \\
T &::= F\ T'. \\
T' &::= \text{'*'}\ F\ T' \mid \varepsilon. \\
F &::= n \mid \text{'('}\ E\ \text{')'}.
\end{aligned}
$$

Инициализируем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
E′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
E ::= T E′

Имеем
FIRST(T E′) = {'(', n},
FOLLOW(E) = {')', $}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$E' ::= \text{'+'}\ T\ E'$

Имеем
$\text{FIRST('+' T E')} = \{\text{'+'}\}$,
$\text{FOLLOW(E')} = \{\text{')'}, \$\}$.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$E' ::= \varepsilon$

Имеем
FIRST(ε) = {ε},
FOLLOW(E') = {')', \$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	\$
E	ERROR	ERROR	T E'	T E'	ERROR	ERROR
E'	'+' T E'	ERROR	ERROR	ERROR	$\underline{\varepsilon}$	$\underline{\varepsilon}$
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
T ::= F T′

Имеем
FIRST(F T′) = {'(',n},
FOLLOW(T) = {'+',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
T′ ::= '*' F T′

Имеем
FIRST('*' F T′) = {'*'},
FOLLOW(T′) = {'+',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ERROR	'*' F T′	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$T' ::= \varepsilon$

Имеем
FIRST(ε) = {ε},
FOLLOW(T') = {'+',')',\$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	\$
E	ERROR	ERROR	T E'	T E'	ERROR	ERROR
E'	'+' T E'	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T'	F T'	ERROR	ERROR
T'	$\underline{\varepsilon}$	'*' F T'	ERROR	ERROR	$\underline{\varepsilon}$	$\underline{\varepsilon}$
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
F ::= n

Имеем
FIRST(n) = {n},
FOLLOW(F) = {'+','*',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ε	'*' F T′	ERROR	ERROR	ε	ε
F	ERROR	ERROR	n	ERROR	ERROR	ERROR

Рассматриваем правило
F ::= '(' E ')'

Имеем
FIRST('(' E ')') = {'('},
FOLLOW(F) = {'+','*',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ε	'*' F T′	ERROR	ERROR	ε	ε
F	ERROR	ERROR	n	'(' E ')'	ERROR	ERROR

§23. Расширенная БНФ

Определение. Говорят, что КС-грамматика $G = \langle N, T, E, S \rangle$ записана в расширенной форме Бэкуса-Наура (РБНФ), если её правила имеют вид $X \to R$, где $X \in N$ – нетерминальный символ, а R – дерево, синтаксис которого задаётся грамматикой

$$
\begin{array}{rcll}
R & ::= & \varepsilon & \text{; пустое дерево} \\
 & | & a & \text{; терминальный символ} \\
 & | & X & \text{; нетерминальныйсимвол} \\
 & | & R\,R & \text{; конкатенация поддеревьев} \\
 & | & R\,\text{'|'}\,R & \text{; альтернатива} \\
 & | & R^{\text{'*'}} & \text{; вхождение 0 и более раз} \\
 & | & R^{\text{'+'}} & \text{; вхождение 1 и более раз} \\
 & | & R^{\text{'?'}}. & \text{; вхождение 0 или 1 раз}
\end{array}
$$

Множество таких синтаксических деревьев мы будем обозначать как RHS.

При текстовой записи правил наивысший приоритет имеют операции '*', '+' и '?', за ними следует конкатенация, а наименьший приоритет имеет альтернатива. При этом круглые скобки служат для изменения приоритета.

Пример. Арифметические выражения.

$$\begin{aligned}
\text{Expr} &::= (\text{'+'} \mid \text{'}-\text{'})^{?}\ \text{Term}\ ((\text{'+'} \mid \text{'}-\text{'})\ \ \text{Term})^{\star}. \\
\text{Term} &::= \text{Factor}\ ((\text{'*'} \mid \text{'/'})\ \ \text{Factor})^{\star}. \\
\text{Factor} &::= \text{Number} \mid \text{'('}\ \text{Expr}\ \text{')'}.
\end{aligned}$$

В книгах Н. Вирта используется альтернативная запись РБНФ, называемая синтаксической нотацией Вирта:

Метасимвол	Значение
=	Разделяет левую и правую части правила.
.	Конец правила.
|	Альтернатива
[ ]	Вхождение 0 или 1 раз.
{ }	Вхождение 0 или более раз.
( )	Изменение приоритета операций.

Приоритет операций – такой же, как и в РБНФ.

Пример. Арифметические выражения.

```
Expr   = ["+"|"-"] Term { ("+"|"-") Term }.
Term   = Factor { ("*"|"/") Factor }.
Factor = Number | "(" Expr ")".
```

§24. Множества FIRST для РБНФ

Пусть дана грамматика $G = \langle N, T, E, S \rangle$, правила которой заданы в РБНФ. Будем обозначать нетерминальные символы грамматики буквами X и Y, терминальные символы – буквой a, а выражения РБНФ – буквами u и v.

Определим функцию $\mathcal{F} : \mathsf{RHS} \longrightarrow 2^T$, работающую в предположении, что для каждого нетерминального символа грамматики определено некоторое множество FIRST, и вычисляющую множество FIRST для любого выражения РБНФ, то есть $\mathsf{FIRST}\,(u) = \mathcal{F}[\![u]\!]$.

$$
\begin{array}{lcl}
\mathcal{F}[\![a]\!] & = & \{a\} \\
\mathcal{F}[\![X]\!] & = & \mathsf{FIRST}\,(X) \\
\mathcal{F}[\![uv]\!] & = & \begin{cases} \mathcal{F}[\![u]\!], & \text{если } \varepsilon \notin \mathcal{F}[\![u]\!] \\ (\mathcal{F}[\![u]\!] \setminus \{\varepsilon\}) \cup \mathcal{F}[\![v]\!], & \text{если } \varepsilon \in \mathcal{F}[\![u]\!] \end{cases} \\
\mathcal{F}[\![u \,|\, v]\!] & = & \mathcal{F}[\![u]\!] \cup \mathcal{F}[\![v]\!] \\
\mathcal{F}[\![u^\star]\!] & = & \mathcal{F}[\![u]\!] \cup \{\varepsilon\} \\
\mathcal{F}[\![u^+]\!] & = & \mathcal{F}[\![u]\!] \\
\mathcal{F}[\![u^?]\!] & = & \mathcal{F}[\![u]\!] \cup \{\varepsilon\} \\
\mathcal{F}[\![\varepsilon]\!] & = & \{\varepsilon\}
\end{array}
$$

Алгоритм построения множеств FIRST
для всех нетерминальных символов грамматики:

Шаг 1. Пусть FIRST $(X) = \emptyset$ для любого $X \in N$.

Шаг 2. Для каждого правила $X \to u$ добавить $\mathcal{F}[\![u]\!]$ в FIRST (X).

Шаг 3. Повторять шаг 2 до тех пор, пока множества FIRST не перестанут изменяться.

§25. Рекурсивный спуск

Мы будем рассматривать метод рекурсивного спуска применительно к LL(1)-грамматикам, записанным в РБНФ.

В синтаксическом анализаторе, написанном методом рекурсивного спуска, каждому нетерминалу грамматики соответствует отдельная функция, в которой закодирован эффект применения соответствующего нетерминалу правила (мы будем считать, что такое правило единственно).

Цель этой функции – анализ последовательности токенов, которые по её запросу выдаёт лексический анализатор, и проверка соответствия этой последовательности правилу грамматики:

```
X() {
    PARSE(правило);
}
```

Функция, соответствующая нетерминалу X грамматики, составляется с учётом того, что:

- существует глобальная переменная Sym, в которую ДО вызова функции помещается токен, соответствующий первому символу цепочки, в которую раскрывается нетерминал X;

- функция должна либо полностью потребить последовательность токенов, в которую раскрывается нетерминал X, либо сгенерировать сообщение об ошибке;

- после завершения функции переменная Sym должна содержать токен, непосредственно следующий за раскрытым нетерминалом X.

Код функции, соответствующая нетерминалу X грамматики, порождается из правила грамматики согласно следующим шаблонам:

1. Пустое дерево

PARSE(ε) $\Rightarrow$

```
    ; /* Ничего не делать. */
```

2. Терминальный символ

PARSE(a) $\Rightarrow$

```
    {
        if (Sym == a)
            Sym = NextToken();
        else
            ReportError();
    }
```

3. Нетерминальный символ

PARSE(X) $\Rightarrow$
    X ( ) ;

4. Конкатенация поддеревьев

PARSE($R_1\, R_2$) $\Rightarrow$
    { PARSE(R_1) ; PARSE(R_2) ; }

5. Альтернатива ($\varepsilon \notin$ FIRST (R_1))

PARSE($R_1 \,|\, R_2$) $\Rightarrow$
    {
        if ($Sym \in$ FIRST (R_1))
            PARSE(R_1) ;
        else
            PARSE(R_2) ;
    }

6. Вхождение 0 или более раз

```
PARSE( R* ) ⇒
        {
            while (Sym ∈ FIRST(R)))
                PARSE( R );
        }
```

7. Вхождение 1 или более раз

```
PARSE( R+ ) ⇒
        {
            do
                PARSE( R );
            while (Sym ∈ FIRST(R)))
        }
```

8. Вхождение 0 или 1 раз

PARSE($R^?$) $\Rightarrow$

```
{
    if  ( Sym ∈ FIRST (R)) )
          PARSE( R ) ;
}
```

Пример. Арифметические выражения.

1. Имеем правило для нетерминала Expr:

Expr ::= ('+'|'−')$^{?}$ Term (('+'|'−') Term)*.

Создаём функцию Expr:

```
Expr() {
    PARSE( ('+'|'−')? Term (('+'|'−') Term)* );
}
```

2. Правило для Expr представляет собой конкатенацию трёх поддеревьев:

```
Expr() {
    PARSE( ('+'|'−')? );
    PARSE( Term );
    PARSE( (('+'|'−') Term)* );
}
```

3. Раскрываем PARSE(('+'|'−')$^?$):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    PARSE( Term );
    PARSE( (('+'|'−') Term)* );
}
```

4. Раскрываем PARSE(Term):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    Term();
    PARSE( (('+'|'−') Term)* );
}
```

5. Раскрываем PARSE((('+'|'−') Term)*):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

6. Раскрываем PARSE('+'|'−'):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−')) {
        if (Sym ∈ FIRST('+'))
            PARSE( '+' );
        else
            PARSE( '−' );
    }
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

7. Раскрываем PARSE('+') и PARSE('−'):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−')) {
        if (Sym ∈ FIRST('+')) {
            if (Sym == '+') Sym = NextToken();
            else ReportError();
        } else {
            if (Sym == '−') Sym = NextToken();
            else ReportError();
        }
    }
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

8. Упрощаем функцию:

```
Expr() {
      if (Sym == '+' || Sym == '-')
            Sym = NextToken();
      Term();
      while (Sym ∈ FIRST(('+'|'-') Term))
            PARSE( ('+'|'-') Term );
}
```

9. Раскрываем PARSE(('+'|'−') Term):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term)) {
        PARSE( '+'|'−' );
        PARSE( Term );
    }
}
```

10. Раскрываем PARSE('+'|'−'):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term)) {
        if (Sym ∈ FIRST('+'|'−')) {
            if (Sym ∈ FIRST('+')) {
                if (Sym == '+') Sym = NextToken();
                else ReportError();
            } else {
                if (Sym == '-') Sym = NextToken();
                else ReportError();
            }
        }
        PARSE( Term );
    }
}
```

11. Упрощаем функцию и раскрываем PARSE(Term):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym == '+' || Sym == '-') {
        Sym = NextToken();
        Term();
    }
}
```

12. Имеем правило для нетерминала Term:
Term ::= Factor (('*'|'/') Factor)*.

Создаём функцию Term:

```
Term() {
    PARSE( Factor (('*'|'/') Factor)* );
}
```

13. После раскрытия PARSE(Factor (('*'|'/') Factor)*) и упрощения получаем:

```
Term() {
    Factor();
    while (Sym == '*' || Sym == '/') {
        Sym = NextToken();
        Factor();
    }
}
```

14. Имеем правило для нетерминала Factor:

Factor ::= Number|'(' Expr ')'.

Создаём функцию Factor:

```
Factor() {
    PARSE( Number|'(' Expr ')' );
}
```

15. Раскрываем PARSE(Number|'(' Expr ')'):

```
Factor() {
    if (Sym ∈ FIRST(Number))
        PARSE( Number );
    else
        PARSE( '(' Expr ')' );
}
```

15. Раскрываем PARSE(Number):

```
Factor() {
    if (Sym ∈ FIRST(Number)) {
        if (Sym == Number)
            Sym = NextToken();
        else
            ReportError();
    } else
        PARSE( '(' Expr ')' );
}
```

16. Упрощаем функцию:

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else
        PARSE( '(' Expr ')' );
}
```

17. Раскрываем PARSE('(' Expr ')'):

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else {
        PARSE( '(' );
        PARSE( Expr );
        PARSE( ')' );
    }
}
```

19. Раскрывая все вхождения PARSE, в итоге получаем:

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else {
        if (Sym == '(')
            Sym = NextToken();
        else
            ReportError();

        Expr();

        if (Sym == ')')
            Sym = NextToken();
        else
            ReportError();
    }
}
```

§22. Алгоритм Эрли

Earley, J. *An Efficient Context-Free Parsing Algorithm* [PhD Thesis]. – Carnegie-Mellon University, 1968.

Алгоритм Эрли подходит для любой КС-грамматики и для заданного предложения языка позволяет найти все левые выводы. Алгоритм Эрли относится к алгоритмам, работающих «сверху-вниз».

Оценка времени: $O(n)$ – для LL(1)-грамматик, $O\left(n^2\right)$ – для непротиворечивых грамматик, $O\left(n^3\right)$ – для всех остальных КС-грамматик.

Работа алгоритма основана на построении последовательности множеств, называемых *множествами Эрли*. Для входной цепочки $a_1 a_2 \ldots a_n$ конструируются $n+1$ множеств: начальное множество S_0 и по одному множеству S_i для каждого входного символа a_i.

Определение. *Элемент множества Эрли* (Earley item), принадлежащий множеству S_k – это тройка $\langle p, i, j \rangle$, в которой:
p – это правило грамматики;
i – позиция в правой части правила, показывающая, какая часть правила уже обработана;
$j \leq k$ – номер множества Эрли, в котором началось распознавание по правилу p.

Элемент $\langle p, i, j \rangle$ принято записывать в виде $[A \to u \bullet v,\ j]$, где $A \to uv$ – это правило p, а знак $\bullet$ задаёт позицию i.

В начале работы алгоритма Эрли в множество S_0 записывается единственный элемент $[S' \to \bullet S,\ 0]$, в котором S – это аксиома грамматики, а $S' \to S$ – вспомогательное правило, добавляемое в грамматику для удобства описания алгоритма.

В процессе работы алгоритма Эрли одни элементы порождают другие. Для того чтобы можно было восстановить последовательность применяемых правил, необходимо строить *дерево потомков*, в узлах которого находятся элементы Эрли, а дуги отражают тот факт, что один элемент породил другой элемент.

Алгоритм Эрли работает за $n+1$ шагов.

На каждом шаге происходит вычисление множества S_i и инициализация множества S_{i+1}:

```
S_{i+1} = ∅;
Q = S_i;
loop {
  Q' = ∅;
  for (каждый элемент x ∈ Q) {
    SCANNER(x);          /* Может добавлять в S_{i+1} */
    PREDICTOR(x, Q');    /* Может добавлять в Q' */
    COMPLETER(x, Q');    /* Может добавлять в Q' */
  }

  if (Q' ⊆ S_i) break;

  S_i = S_i ∪ Q';
  Q = Q';
}
```

Пусть $a_1 a_2 \ldots a_n$ – входная цепочка, A и B обозначают нетерминальные символы, b обозначает терминальный символ, u обозначает цепочку, составленную из терминальных и нетерминальных символов.

SCANNER(x). Если $x = [A \to \ldots \bullet b \ldots, j]$ и $b = a_{i+1}$,
добавить $[A \to \ldots b \bullet \ldots, j]$ в S_{i+1}.

PREDICTOR(x,Q). Если $x = [A \to \ldots \bullet B \ldots, j]$, то для каждого правила вида $B \to u$ добавить в Q элемент $[B \to \bullet u, i]$.
При этом, если $B \Rightarrow^\star \varepsilon$, то в Q также нужно добавить элемент $[A \to \ldots B \bullet \ldots, j]$.

COMPLETER(x,Q). Если $x = [A \to \ldots \bullet, j]$, то для каждого элемента $[B \to \ldots \bullet A \ldots, k] \in S_j$ добавить в Q элемент $[B \to \ldots A \bullet \ldots, k]$.

Пример. Грамматика $E \to E\,\text{'+'}\,E \mid n$ и входная цепочка $n\,\text{'+'}\,n$.

S_0

Q_1	S'	$\to$	$\bullet E$	, 0
Q_2	E	$\to$	$\bullet E\,\text{'+'}\,E$	, 0
	E	$\to$	$\bullet n$	, 0

n

S_1

Q_1	E	$\to$	$n\bullet$	, 0
Q_2	S'	$\to$	$E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 0

'+'

S_2

Q_1	E	$\to$	$E\,\text{'+'}\bullet\,E$	, 0
Q_2	E	$\to$	$\bullet E\,\text{'+'}\,E$	, 2
	E	$\to$	$\bullet n$	, 2

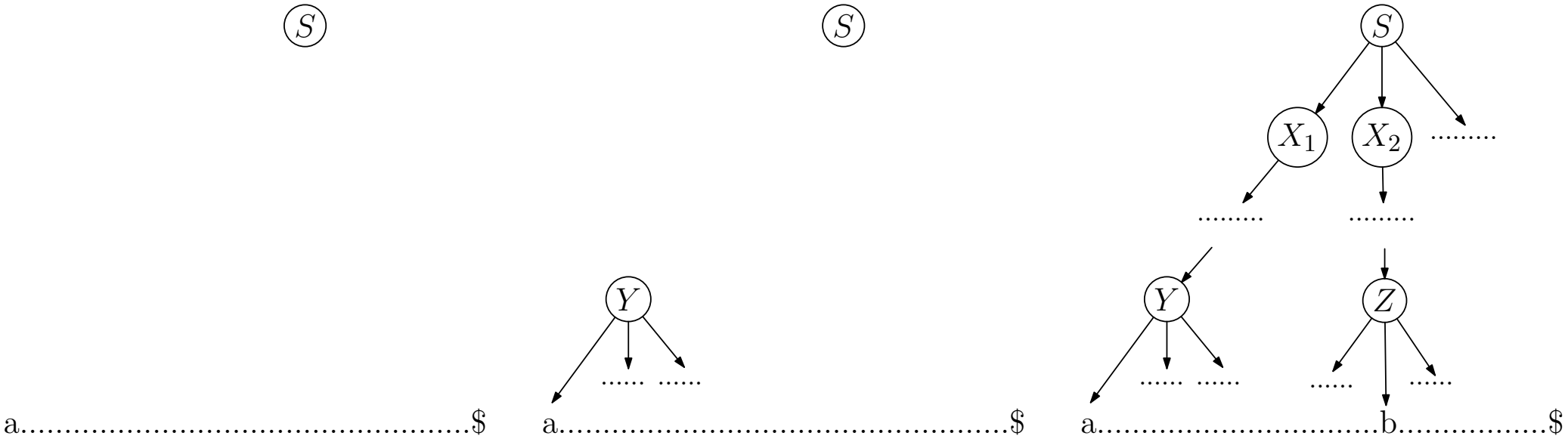
n

S_3

Q_1	E	$\to$	$n\bullet$	, 2
Q_2	E	$\to$	$E\,\text{'+'}\,E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 2
Q_3	S'	$\to$	$E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 0

§26. Синтаксический разбор типа «перенос–свёртка»

Последовательность формирования дерева вывода в процессе синтаксического разбора типа «перенос–свёртка» (снизу-вверх):



ПС-алгоритм (алгоритм синтаксического разбора типа «перенос–свёртка») строит правосторонний вывод предложения языка в обратном порядке.

Пример. Арифметические выражения.

$T = \{'+', '*', \mathsf{n}, '(', ')', \$\}$, $N = \{\mathsf{E}, \mathsf{T}, \mathsf{F}\}$, $S = \mathsf{E}$,

$P =$

$\mathsf{E} ::= \mathsf{E}\ '+'\ \mathsf{T} \mid \mathsf{T}.$

$\mathsf{T} ::= \mathsf{T}\ '*'\ \mathsf{F} \mid \mathsf{F}.$

$\mathsf{F} ::= \mathsf{n} \mid '('\ \mathsf{E}\ ')'.$

Для цепочки $\mathsf{n} * (\mathsf{n} + \mathsf{n})$ ПС-алгоритм строит вывод:

$\underline{\mathsf{n}} * (\mathsf{n} + \mathsf{n}) \Leftarrow \underline{\mathsf{F}} * (\mathsf{n} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{n}} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{F}} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{T}} + \mathsf{n}) \Leftarrow$
$\Leftarrow \mathsf{T} * (\mathsf{E} + \underline{\mathsf{n}}) \Leftarrow \mathsf{T} * (\mathsf{E} + \underline{\mathsf{F}}) \Leftarrow \mathsf{T} * \underline{(\mathsf{E} + \mathsf{T})} \Leftarrow \mathsf{T} * \underline{(\mathsf{E})} \Leftarrow \underline{\mathsf{T} * \mathsf{F}} \Leftarrow \underline{\mathsf{T}} \Leftarrow \mathsf{E}.$

Определение. *Основа* правой сентенциальной формы u – это сочетание правила $X \to v$ и позиции цепочки v в u, такое, что вхождение v в u может быть заменено нетерминалом X для получения предыдущей правой сентенциальной формы в правостороннем выводе u.

После основы в правой сентенциальной форме идут только терминальные символы. В примере↑ все основы подчёркнуты.

ПС-алгоритм работает с двумя структурами данных:
– стек, в который помещаются символы грамматики ($N \cup T$), и дно которого помечено символом \$;
– входной буфер, содержащий нераспознанный суффикс входной цепочки и оканчивающийся символом \$.

В начале работы ПС-алгоритма стек содержит \$, а входной буффер – u\$, где u – распознаваемая цепочка.

ПС-алгоритм выполняет четыре действия:
1. *перенос* – символ в начале входного буфера переносится в стек;
2. *свёртка* – основа на вершине стека заменяется нетерминалом;
3. *допуск* – успешное окончание разбора (во входном буфере – \$, в стеке – \$$S$, где S – аксиома грамматики);
4. *ошибка* – вызов подпрограммы реакции на ошибку во входном потоке.

Основные проблемы ПС-алгоритма – поиск основы в стеке и выбор правила, по которому нужно осуществлять свёртку. В общем случае ПС-алгоритм осуществляет возвраты (см. [Ахо, Ульман]).

Пример. Разбор цепочки $n * (n + n)$.

№	Стек	Вход	Действие
1	$	n*(n+n)$	Перенос
2	$n	*(n+n)$	Свёртка по правилу F → n
3	$F	*(n+n)$	Свёртка по правилу T → F
4!	$T	*(n+n)$	Перенос ×3
7	$T*(n	+n)$	Свёртка по правилу F → n
8	$T*(F	+n)$	Свёртка по правилу T → F
9	$T*(T	+n)$	Свёртка по правилу E → T
10	$T*(E	+n)$	Перенос ×2
12	$T*(E+n	)$	Свёртка по правилу F → n
13	$T*(E+F	)$	Свёртка по правилу T → F
14!	$T*(E+T	)$	Свёртка по правилу E → E '+' T
15	$T*(E	)$	Перенос
16	$T*(E)	$	Свёртка по правилу F → '(' E ')'
17!	$T*F	$	Свёртка по правилу T → T '*' F
18	$T	$	Свёртка по правилу E → T
19	$E	$	Допуск

§27. Недетерминированные SLR-распознаватели

Определение. *Активный префикс* (viable prefix) – это префикс правосентенциальной формы, который не выходит за правый конец основы этой сентенциальной формы.

Если ПС-алгоритм работает правильно (то есть не пошёл по ложному пути), то его стек содержит знак \$, выше которого располагается активный префикс.

Пример. На 14-ом шаге разбора цепочки $n * (n + n)$ (см. пример в §26) в стеке содержится цепочка
\$T*(E+T
в которой T*(E+T является активным префиксом.

Определение. *SLR-ситуация* (SLR item) – это пара $\langle p, i \rangle$, в которой:

p – правило грамматики;

i – позиция в правой части правила, показывающая, какая часть правила уже обработана.

Далее мы будем называть SLR-ситуации просто *ситуациями*.

Ситуацию $\langle p, i \rangle$ принято записывать в виде $X \to u \bullet v$ или $[X \to u \bullet v]$, где $X \to uv$ – это правило p, а знак $\bullet$ задаёт позицию i.

Для каждого правила $X \to x_1 x_2 \dots x_n$ можно построить $(n + 1)$ ситуаций:
$X \to \bullet x_1 x_2 \dots x_n$,
$X \to x_1 \bullet x_2 \dots x_n$,
$X \to x_1 x_2 \bullet \dots x_n$,
...,
$X \to x_1 x_2 \dots \bullet x_n$,
$X \to x_1 x_2 \dots x_n \bullet$.

Для правила вида $X \to \varepsilon$ возможна единственная ситуация $X \to \bullet$.

Определение. *Расширенная грамматика* для грамматики $G = \langle N, T, P, S \rangle$ – это грамматика $G' = \langle N', T, P', S' \rangle$, в которой $N' = N \cup \{S'\}$, $P' = P \cup \{S' \to S\}$.

Основные проблемы ПС-алгоритма – поиск основы в стеке и выбор правила, по которому нужно осуществлять свёртку – для некоторых КС-грамматик можно решить с помощью распознавателей активных префиксов.

Определение. *Недетерминированный SLR-распознаватель активных префиксов* для расширенной грамматики $G' = \langle N', T, P', S' \rangle$ – это конечный автомат, состояниями которого являются все возможные SLR-ситуации для P', а переходы между ними задаются условиями:
1) $[X \to u \bullet xv] \overset{x}{\mapsto} [X \to ux \bullet v]$, где $x \in N' \cup T$;
2) $[X \to u \bullet Yv] \overset{\varepsilon}{\mapsto} [Y \to \bullet w]$, где $Y \in N'$.
При этом начальным состоянием автомата является ситуация $[S' \to \bullet S]$, а заключительными состояниями являются все ситуации вида $[X \to u\bullet]$.

Пример.

Правила грамматики:

$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & L = R \\
S & \to & R \\
L & \to & \star R \\
L & \to & v \\
R & \to & L
\end{array}
$$

Примеры выводимых предложений:

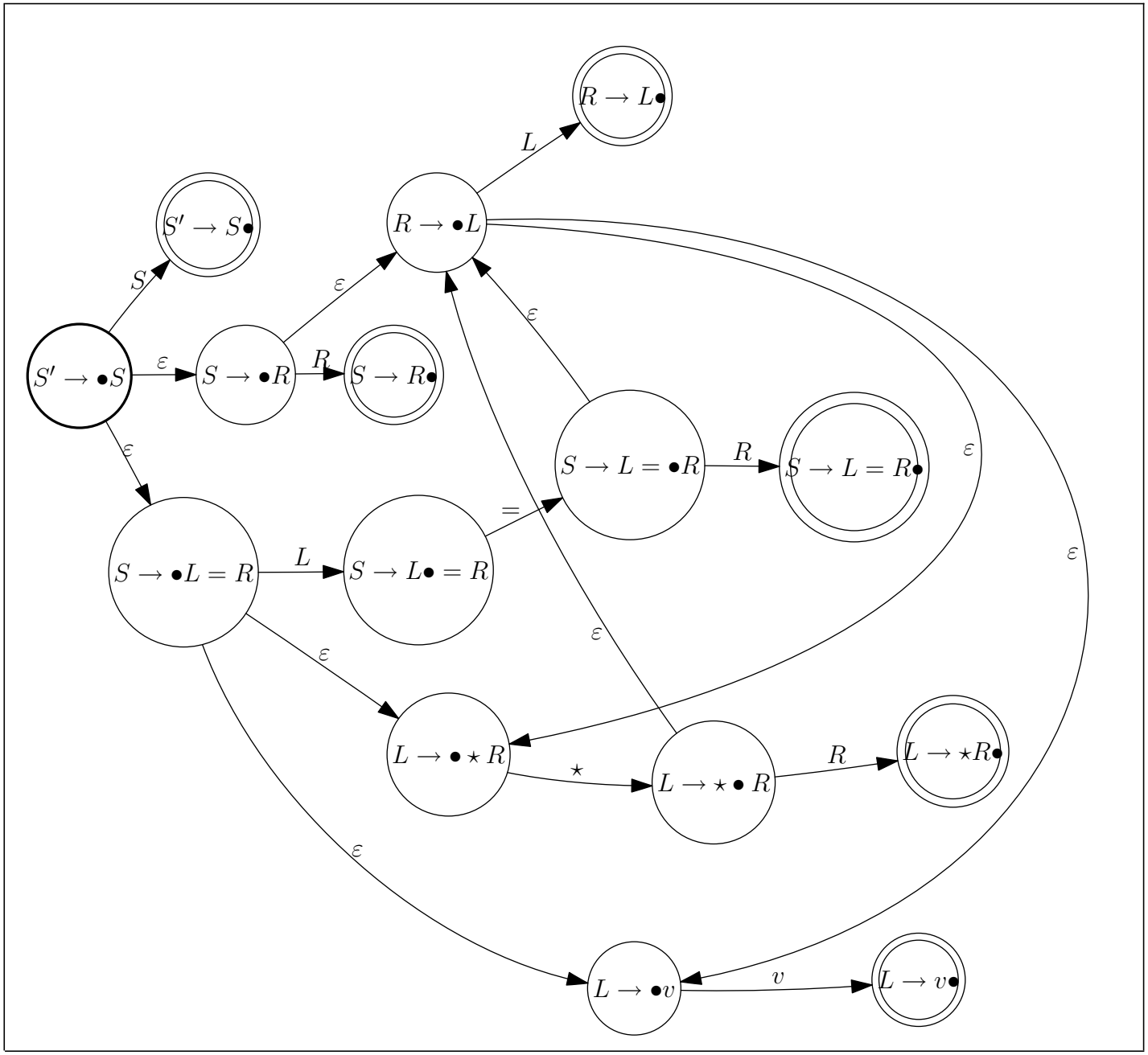
v

$\star v$

$\star \star v$

$v = v$

$\star \star v = \star v$



Если «натравить» SLR-распознаватель на активный префикс, лежащий в стеке ПС-алгоритма, то в результате мы можем получить некоторую ситуацию, которая является подсказкой для ПС-алгоритма:
– заключительная ситуация вида $[X \to u\bullet]$ говорит о том, что нужно выполнить свёртку по правилу $X \to u$;
– ситуация вида $[X \to u \bullet xv]$ означает, что нужно выполнить перенос.

Пример. Пусть выполняется синтаксический анализ цепочки $\star\star v = \star v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс $L = \star R$.

Распознавание активного префикса:
$[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{L}{\mapsto} [S \to L\bullet = R] \overset{=}{\mapsto} [S \to L = \bullet R] \overset{\varepsilon}{\mapsto} [R \to \bullet L] \overset{\varepsilon}{\mapsto}$
$\overset{\varepsilon}{\mapsto} [L \to \bullet \star R] \overset{\star}{\mapsto} [L \to \star \bullet R] \overset{R}{\mapsto} [L \to \star R\bullet]$.

$[L \to \star R\bullet]$ – заключительное состояние, означающее свёртку основы $\star R$ по правилу $L \to \star R$.

Если распознавание активного префикса оказалось невозможным, это означает, что обнаружена ошибка.

Пример. Пусть выполняется синтаксический анализ цепочки $\star = \star v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс $\star =$.

Распознавание активного префикса:
$[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{\varepsilon}{\mapsto} [L \to \bullet \star R] \overset{\star}{\mapsto} [L \to \star \bullet R] \mapsto ?$

Активный префикс не может быть распознан – обнаружена ошибка.

Если в результате распознавания активного префикса мы получили более одной ситуации, это означает, что КС-грамматика слишком сложна для SLR-распознавателя.

Пример. Пусть выполняется синтаксический анализ цепочки $v = v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс L.

Распознавание активного префикса:
1. $[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{L}{\mapsto} [S \to L\bullet = R]$ – перенос;
2. $[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet R] \overset{\varepsilon}{\mapsto} [R \to \bullet L] \overset{L}{\mapsto} [R \to L\bullet]$ – свёртка по правилу $R \to L$.

Имеем так называемый конфликт «перенос/свёртка».

Кроме конфликтов «перенос/свёртка» бывают ещё конфликты «свёртка/свёртка».

§28. Детерминированные SLR-распознаватели

Если выполнить детерминизацию недетерминированного SLR-распознавателя, то получится *детерминированный SLR-распознаватель активного префикса*, состояниями которого являются не отдельные ситуации, а множества ситуаций.

При этом начальным состоянием детерминированного SLR-распознавателя является состояние, содержащее ситуацию $[S' \to \bullet S]$, а заключительными состояниями являются состояния, содержащие заключительные состояния соответствующего недетминированного SLR-распознавателя.

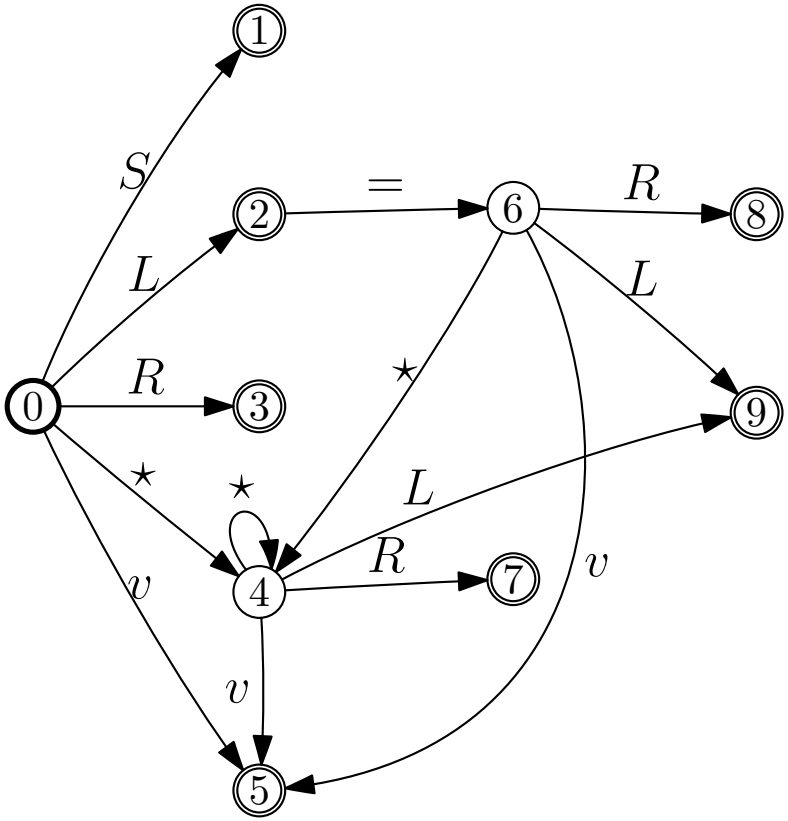
На практике используются только детерминированные SLR-распознаватели, потому что они позволяют распознавать активные префиксы за время, пропорциональное длине префикса.

Пример.

Правила грамматики:

$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & L = R \\
S & \to & R \\
L & \to & \star R \\
L & \to & v \\
R & \to & L
\end{array}
$$

(В состоянии 2 – конфликт «перенос/свёртка».)



0 :
$$
\begin{array}{lcl}
S' & \to & \bullet S \\
S & \to & \bullet L = R \\
S & \to & \bullet R \\
L & \to & \bullet \star R \\
L & \to & \bullet v \\
R & \to & \bullet L
\end{array}
$$

1 :
$$
S' \to S\bullet
$$

2 :
$$
\begin{array}{lcl}
S & \to & L\bullet = R \\
R & \to & L\bullet
\end{array}
$$

3 :
$$
S \to R\bullet
$$

4 :
$$
\begin{array}{lcl}
L & \to & \star \bullet R \\
R & \to & \bullet L \\
L & \to & \bullet \star R \\
L & \to & \bullet v
\end{array}
$$

5 :
$$
L \to v\bullet
$$

6 :
$$
\begin{array}{lcl}
S & \to & L = \bullet R \\
R & \to & \bullet L \\
L & \to & \bullet \star R \\
L & \to & \bullet v
\end{array}
$$

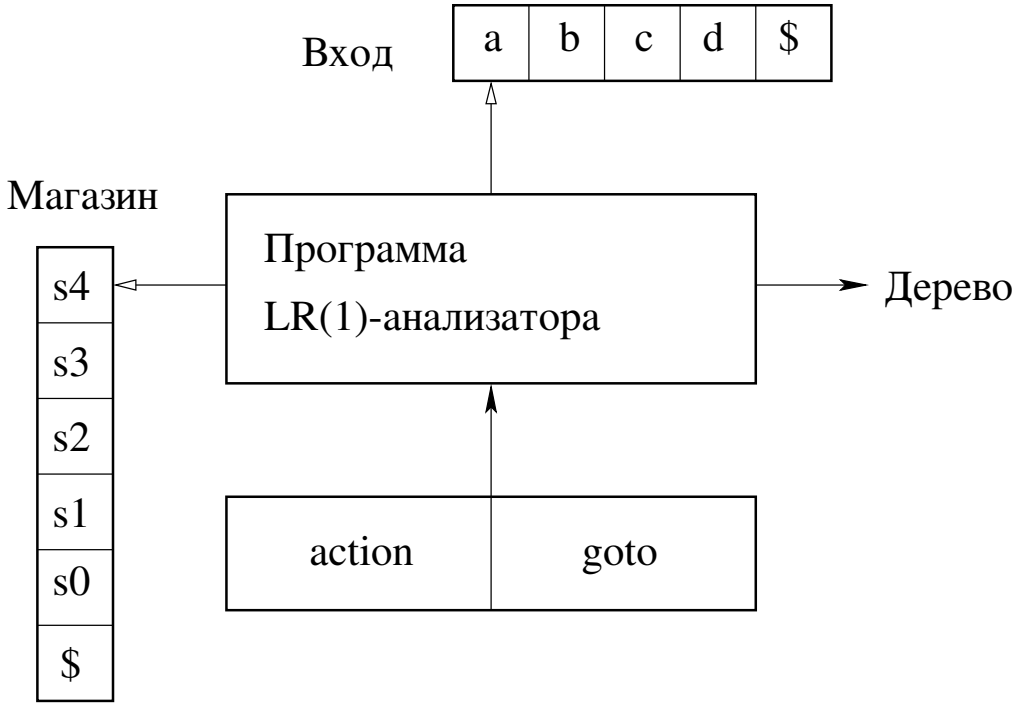
7 :
$$
L \to \star R\bullet
$$

8 :
$$
S \to L = R\bullet
$$

9 :
$$
R \to L\bullet
$$

§29. Алгоритм LR(1)-анализа

Алгоритм LR(1)-анализа является вариантом ПС-алгоритма, работающим за линейное время. В стеке LR(1)-анализатора содержится последовательность состояний, через которые прошёл SLR-распознаватель (или более сложный LR-распознаватель) в процессе распознавания активного префикса. LR(1)-анализатор управляется таблицами распознавателя, не содержащими конфликтов.



action:

$State \times T \longrightarrow$ перенос $State$
| свёртка $X \to u$
| допуск
| ошибка

goto:

$State \times N' \longrightarrow State$
| ошибка

```
LR1Parse(N, T (содержит $), S,
         s0, action, goto, цепочка x) {
  Result = пустая последовательность;
  поместить в стек $, а затем s0;
  a = первый символ из x;
  loop {
      s = состояние на вершине стека;
      if (action[s,a] == "перенос s'") {
          поместить в стек s';
          a = следующий символ из x;
      } else if (action[s,a] == "свёртка X → u") {
          снять со стека |u| состояний.
          s' = состояние на вершине стека;
          поместить в стек goto[s',X];
          добавить правило X → u в Result;
      } else if (action[s,a] == "допуск")
          return Result;
      else error();
  }
}
```

Таблица action[s, a], где s – номер состояния SLR-распознавателя, а $a \in T \cup \{\$\}$, задаётся следующим образом:
– action[s,a] $\ni$ «перенос s'», если $s \overset{a}{\mapsto} s'$;
– action[s,a] $\ni$ «свёртка $X \to u$», если $[X \to u\bullet] \in s$ и $a \in \mathrm{FOLLOW}(X)$;
– action[s,\$] $\ni$ «допуск», если $[S' \to S\bullet] \in s$;
– action[s,a] $\ni$ «ошибка», если ни одна из вышеприведённых альтернатив не сработала.

Таблица goto[s,X], где s – номер состояния SLR-распознавателя, а $X \in N'$, задаётся следующим образом:
– goto[s,X] $= s'$, если $s \overset{X}{\mapsto} s'$;
– goto[s,X] $=$ «ошибка» в противном случае.

Пример.

FOLLOW $(S') = \{\$\}$,
FOLLOW $(S) = \{\$\}$,
FOLLOW $(L) = \{=, \$\}$,
FOLLOW $(R) = \{=, \$\}$

№	action				goto			
	$=$	$\star$	v	\$	S'	S	L	R
0	ош.	пер.4	пер.5	ош.	ош.	1	2	3
1	ош.	ош.	ош.	доп.	ош.	ош.	ош.	ош.
2	пер.6, св.$R \to L$	ош.	ош.	св.$R \to L$	ош.	ош.	ош.	ош.
3	ош.	ош.	ош.	св.$S \to R$	ош.	ош.	ош.	ош.
4	ош.	пер.4	пер.5	ош.	ош.	ош.	9	7
5	св.$L \to v$	ош.	ош.	св.$L \to v$	ош.	ош.	ош.	ош.
6	ош.	пер.4	пер.5	ош.	ош.	ош.	9	8
7	св.$L \to \star R$	ош.	ош.	св.$L \to \star R$	ош.	ош.	ош.	ош.
8	ош.	ош.	ош.	св.$S \to L = R$	ош.	ош.	ош.	ош.
9	св.$R \to L$	ош.	ош.	св.$R \to L$	ош.	ош.	ош.	ош.