LL(k)-грамматики

Использование для выбора одной из множества альтернатив не одного, а нескольких символов входной цепочки

Грамматика обладает свойством LL(k), k > 0, если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые *k* символов от текущего положения считывающей головки во входной цепочке символов

LL – Left Left

Цепочка символов читается слева направо

Левосторонний вывод

LL(k)-грамматики

- Алгоритм разбора входных цепочек для LL(k)-грамматик – *k-предсказывающий алгоритм*
- Всякая LL(k)-грамматика для любого $k > 0$ является однозначной
- Любая грамматика, допускающая разбор по методу рекурсивного спуска, является LL(1)-грамматикой (но не наоборот!)
- Никакая леворекурсивная грамматика не может быть LL(k)-грамматикой

FIRST и FOLLOW

Множество FIRST(A) - множество терминальных символов, которыми начинаются цепочки, выводимые из A в грамматике G(VT, VN, P, S),

т.е. $FIRST(A) = \{ a \in \boldsymbol{VT} \mid A \to a\alpha, A \in \boldsymbol{VN}, \alpha \in (\boldsymbol{VT} \cup \boldsymbol{VN})^* \}$

Множество FOLLOW(A) - множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике G (VT, VN, P, S), т.е.

$$FOLLOW(A) = \{ a \in \boldsymbol{VT} \mid S \to A\beta, \beta \to a\gamma, A \in \boldsymbol{VN}, \alpha, \beta, \gamma \in (\boldsymbol{VT} \cup \boldsymbol{VN})^* \}$$

Пример

$E \to TE'$

$E' \to \mathbf{+}TE' \mid \varepsilon$

$T \to FT'$

$T' \to \mathbf{*}FT' \mid \varepsilon$

$F \to \mathbf{(}E\mathbf{)} \mid \mathbf{i}$

1. FIRST(F) = FIRST(T) = FIRST(E) = {**(**, **i**}
2. FIRST(E') = {**+**, ε}
3. FIRST(T') = {*, ε}
4. FOLLOW(E) = FOLLOW(E') = {**)**, $\perp$}
5. FOLLOW(T) = FOLLOW(T') = {**+**, **)**, $\perp$}
6. FOLLOW(F) = {**+**, *, **)**, $\perp$}

Условие применимости

$$FIRST^{+}(A \to \beta) = \begin{cases} FIRST(\beta), \text{если } \varepsilon \notin FIRST(\beta) \\ FIRST(\beta) \cup FOLLOW(A), \text{иначе} \end{cases}$$

$$A \to \beta_1 | \beta_2 | \cdots | \beta_n$$

$$FIRST^{+}(A \to \beta_i) \cap FIRST^{+}(A \to \beta_j) = \varnothing \quad \forall 1 \le i, j \le n, i \ne j$$

LL(1)-грамматики

G принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных правил $A \to \alpha \mid \beta$ выполняется:

1. Не существует такого терминала *a*, для которого и α, и β порождают строку, начинающуюся с *a*.

2. Пустую строку может порождать не более чем одно из правил α или β.

3. Если $\beta \Rightarrow^* \varepsilon$, то α не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A). Аналогично, если $\alpha \Rightarrow^* \varepsilon$, то β не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A).

Таблица предиктивного синтаксического анализа

$T[A, a]$ – двумерный массив

A – нетерминал

a – терминал или символ, маркирующий конец входного потока

Алгоритм построения таблицы

Вход: грамматика G.

Выход: таблица синтаксического анализа T.

Метод: для каждого правила грамматики $A \to \alpha$ выполняем следующие действия.

1. Для каждого терминала a из FIRST(A) добавляем $A \to \alpha$ в ячейку $T\ [A, a]$.
2. Если $\varepsilon \in$ FIRST(A), то для каждого терминала b из FOLLOW(A) добавляем $A \to \alpha$ в $T[A, b]$. Если $\varepsilon \in$ FIRST(A) и $\perp \in$ FOLLOW(A), то добавляем $A \to \alpha$ также и в T[A, $\perp$].

Если после выполнения этих действий ячейка $T[A, a]$ осталась без правила, устанавливаем ее значение равным `error` (это значение обычно представляется пустой записью таблицы).

Пример

0	Goal → Expr	6	Term' → × Factor Term' |
1	Expr → Term Expr'	7	÷ Factor Term' |
2	Expr' → + Term Expr' |	8	ε
3	- Term Expr' |	9	Factor → (Expr) |
4	ε	10	num |
5	Term → Factor Term'	11	name

	Expr	Expr'	Term	Term'	Factor
FIRST	(, name,num	+, -, ε	(, name,num	×, ÷, ε	(, name, num

	Expr	Expr'	Term	Term'	Factor
FOLLOW	⊥,)	⊥,)	⊥, +, -,)	⊥, +, -,)	⊥, +, -, ×, ÷,)

Пример (продолжение)

0	Goal → Expr	6	Term' → × Factor Term' |
1	Expr → Term Expr'	7	÷ Factor Term' |
2	Expr' → + Term Expr' |	8	ε
3	- Term Expr' |	9	Factor → (Expr) |
4	ε	10	num |
5	Term → Factor Term'	11	name

	⊥	+	-	×	÷	(	)	name	num
Goal						0		0	0
Expr						1		1	1
Expr'	4	2	3				4		
Term						5		5	5
Term'	8	8	8	6	7		8		
Factor						9		11	10

Таблица предиктивного синтаксического анализа

Пример (продолжение)

a + b × c

Правило	Стек	Входная цепочка символов
	⊥Goal	
0	⊥Expr	name + name × name
1	⊥Expr’ Term	name + name × name
5	⊥Expr’ Term’ Factor	name + name × name
11	⊥Expr’ Term’ name	name + name × name
→	⊥Expr’ Term’	name + name × name
8	⊥Expr’	name + name × name
2	⊥Expr’ Term +	name + name × name
→	⊥Expr’ Term	name + name × name
5	⊥Expr’ Term’ Factor	name + name × name
11	⊥Expr’ Term’ name	name + name × name
→	⊥Expr’ Term’	name + name × name
6	⊥Expr’ Term’ Factor ×	name + name × name
→	⊥Expr’ Term’ Factor	name + name × name
11	⊥Expr’ Term’ name	name + name × name
→	⊥Expr’ Term’	name + name × name
8	⊥Expr’	name + name × name
4	⊥	name + name × name

Пример (продолжение)

x + ÷ y

Правило	Стек	Входная цепочка символов
	⊥Goal	
0	⊥Expr	name + ÷ name
1	⊥Expr' Term	name + ÷ name
5	⊥Expr' Term' Factor	name + ÷ name
11	⊥Expr' Term' name	name + ÷ name
→	⊥Expr' Term'	name + ÷ name
8	⊥Expr'	name + ÷ name
2	⊥Expr' Term +	name + ÷ name
→	⊥Expr' Term	name + ÷ name

синтаксическая ошибка

```
word = NextWord();
push the start symbol onto Stack;
focus = top of the Stack;
for(;;){
   if(focus = ⊥ && word = ⊥){
     // success
     break;
   }else if(focus ∈ T || focus = ⊥){ // focus is a terminal
     if(focus matches word){
       pop Stack;
       word = NextWord();
     }else{
       error;
     }
   }else{ // focus is a nonterminal
     if(Table[focus,word] is A → B1B2...Bk){
        pop Stack;
        for(i = k; i >= 1; i--){
           if(Bi != ε)
             push Bi onto the Stack;
        }
     }else{
        error;
     }
     focus = top of the Stack;
   }
}
```

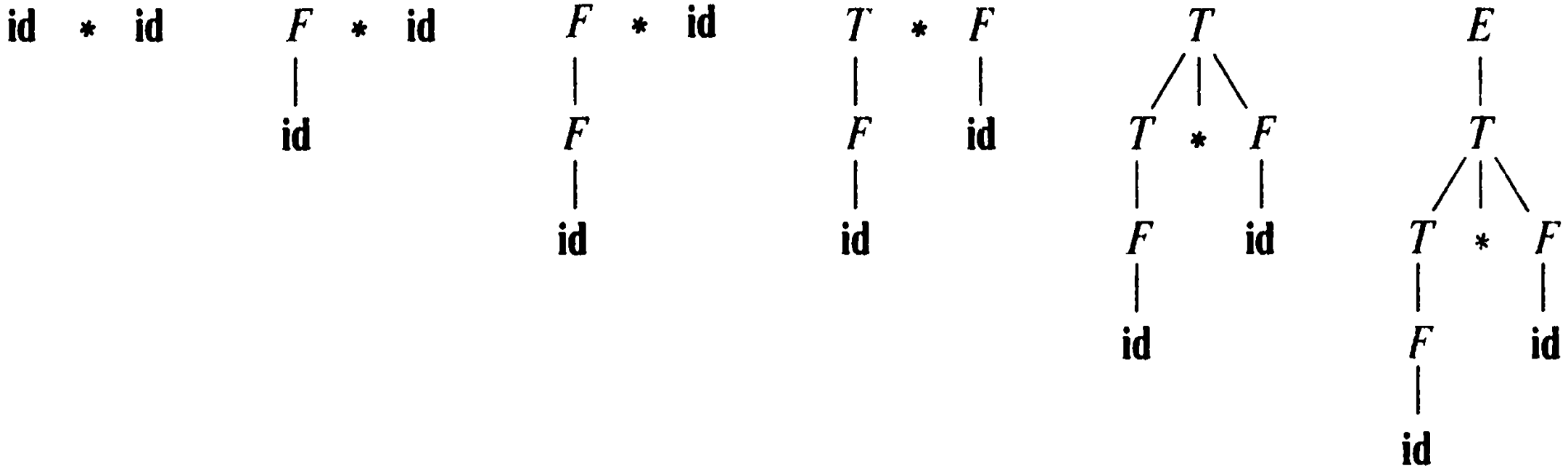
Алгоритм LL(1)-разбора

Литература

1. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p. (Раздел 3.3)
2. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М.: Изд. дом Вильямс, 2008. – 1184 с. (Раздел 4.4)

Восходящий синтаксический анализ

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх)



Алгоритм «сдвиг-свертка»

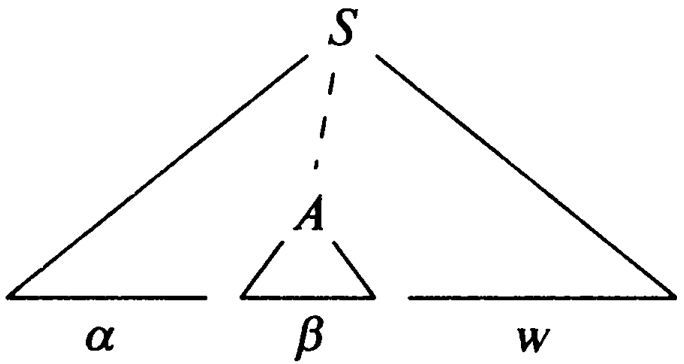
Свертка представляет собой шаг, обратный порождению

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Основа строки — это подстрока, которая соответствует телу правила и свертка которой представляет собой один шаг правого порождения в обратном порядке

Алгоритм «сдвиг-свертка»

Правая сентенциальная форма	Основа	Сворачивающее правило
id * id	id	$F \rightarrow id$
F * id	F	$T \rightarrow F$
T * id	id	$F \rightarrow id$
T * F	T * F	$T \rightarrow T * F$
T	T	$E \rightarrow T$



Если $S \Rightarrow^* \alpha A \omega \Rightarrow \alpha\beta\omega$, то правило $A \rightarrow \beta$ в позиции после α является основой

Алгоритм «сдвиг-свертка»

1. Сдвиг (shift). Перенос очередного входного символа на вершину стека.
2. Свертка (reduce). Правая часть сворачиваемой строки должна располагаться на вершине стека. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка.
3. Принятие (accept). Объявление об успешном завершении синтаксического анализа.
4. Ошибка (error). Обнаружение синтаксической ошибки и вызов подпрограммы восстановления после ошибки.

Пример

Стек	Вход	Действие
$\perp$	**id * id**$\perp$	Сдвиг
$\perp$**id**	***** **id**$\perp$	Свертка (F $\rightarrow$ **id**)
$\perp$F	***** **id**$\perp$	Свертка (T $\rightarrow$ F)
$\perp$T	***** **id**$\perp$	Сдвиг
$\perp$T *****	**id**$\perp$	Сдвиг
$\perp$T ***** **id**	$\perp$	Свертка (F $\rightarrow$ **id**)
$\perp$T ***** F	$\perp$	Свертка (T $\rightarrow$ T ***** F)
$\perp$T	$\perp$	Свертка (E $\rightarrow$ T)
$\perp$E	$\perp$	Принятие

Конфликты в процессе использования алгоритма «сдвиг-свертка»

1. *Конфликт «сдвиг/свертка»* : анализатор не может принять решение о том, следует ли выполнить сдвиг или свертку

2. *Конфликт «свертка/свертка»* : анализатор не может принять решение о том, какая свертка должна быть выполнена

LR(k)-анализ

LR(k)-анализ – наиболее распространенная концепция восходящего анализа

L – сканирование входного потока слева направо

R – построение правого порождения в обратном порядке

k – количество предпросматриваемых символов входного потока, необходимое для принятия решения

$k = 0$ $k = 1$

Пункты (ситуации)

LR-анализатор принимает решение о выборе «сдвиг/свертка», поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа мы находимся

Состояния представляют собой множества «пунктов»

LR(0)-пункт грамматики **G** — это правило грамматики **G** с точкой в некоторой позиции правой части

$$A \to XYZ$$

$$A \to \cdot XYZ$$

$$A \to X \cdot YZ$$

$$A \to XY \cdot Z$$

$$A \to XYZ \cdot$$

$$A \to \varepsilon$$

$$A \to \cdot$$

Канонический набор пунктов LR(0)

Один набор множеств LR(0)-пунктов, именуемый *каноническим набором LR(0)*, обеспечивает основу для построения детерминированного автомата (*LR(0)-автомата*), который используется для принятия решений в процессе синтаксического анализа.

- Расширенная грамматика
- Множество CLOSURE
- Множество GOTO

Расширенная грамматика

Расширенная грамматика представляет собой грамматику с новым целевым символом S' и правилом $S' \to S$

Замыкание множества пунктов

Если I — множество пунктов грамматики **G**, то $CLOSURE(I)$ представляет собой множество пунктов, построенное из I согласно двум правилам.

1. Изначально в $CLOSURE(I)$ добавляются все пункты из I.
2. Если $\mathrm{A} \to \alpha \bullet \mathrm{B}\beta$ входит в $CLOSURE(I)$, а $\mathrm{B} \to \gamma$ является правилом, то в $CLOSURE(I)$ добавляется пункт $\mathrm{B} \to \bullet \gamma$, если его там еще нет. Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в $CLOSURE\ (I)$.

Пример

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

1. $CLOSURE(I) = \{E' \rightarrow \bullet E\}$
2. $CLOSURE(I) = \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T\}$
3. $CLOSURE(I) = \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F\}$
4. $CLOSURE(I) = \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet(E), F \rightarrow \bullet id\}$

```
SetOfItems CLOSURE (I) {
    J = I;
    repeat
        for ( каждый пункт A → α · Bβ из J )
            for ( каждая продукция B → γ из G )
                if ( B → ·γ не входит в J )
                    Добавить B → ·γ в J;
    until больше нет пунктов для добавления в J за один проход;
    return J;
}
```

Пункты

1. *Базисные пункты*, или пункты ядра (kernel items): начальный пункт $S' \rightarrow \bullet S$ и все пункты, у которых точки расположены не у левого края.

2. *Небазисные* (nonkernel) пункты, у которых точки расположены слева, за исключением $S' \rightarrow \bullet S$

Множество GOTO

$GOTO(I, X)$, где I — множество пунктов, а X — грамматический символ

$GOTO(I, X)$ определяется как замыкание множества всех пунктов $[\mathrm{A} \rightarrow \alpha \mathrm{X} \bullet \beta]$, таких, что $[A \rightarrow \ \alpha \bullet \mathrm{X}\beta]$ находится в I

Пример

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

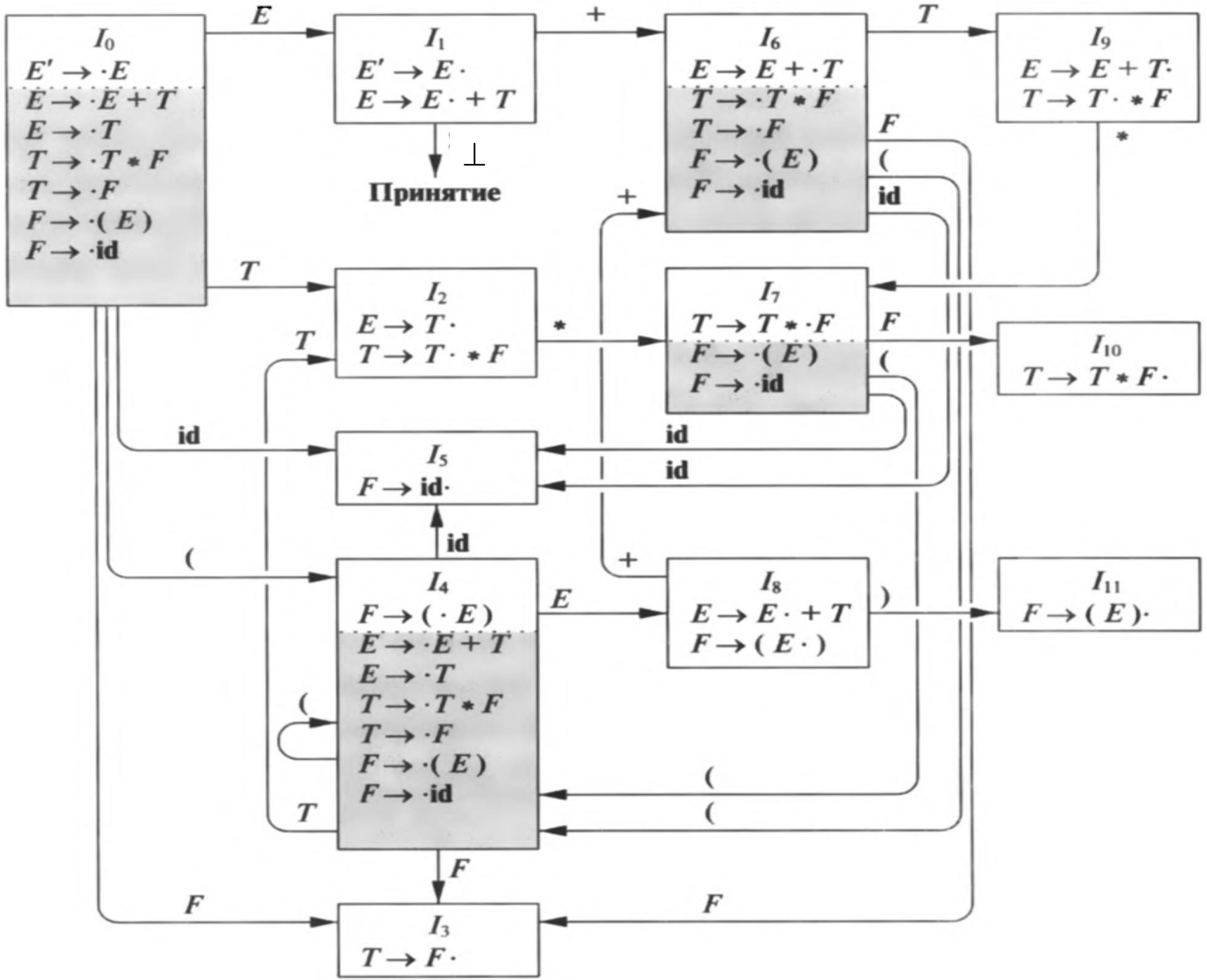
$$
\begin{aligned}
E &\rightarrow E + \cdot T \\
T &\rightarrow \cdot T * F \\
T &\rightarrow \cdot F \\
F &\rightarrow \cdot (E) \\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}
$$

$$I = \{E' \rightarrow \mathrm{E} \bullet, \mathrm{E} \longrightarrow \mathrm{E} \bullet + \mathrm{T}\}$$

$$GOTO(I, +) = ?$$

Алгоритм построения канонического набора пунктов

```
void items (G') {
    C = {CLOSURE ({[S' → ·S]})};
    repeat
        for ( каждое множество пунктов I в C )
            for ( каждый грамматический символ X )
                if ( множество GOTO (I, X) не пустое и не входит в C )
                    Добавить GOTO (I, X) в C;
    until нет новых множеств пунктов для добавления в C за один проход;
}
```


I0
E′ → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E′ → E·
E → E· + T
⊥
Принятие
I2
E → T·
T → T· * F
I3
T → F·
I4
F → (· E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I7
T → T * ·F
F → ·(E)
F → ·id
I8
E → E· + T
F → (E·)
I9
E → E + T·
T → T· * F
I10
T → T * F·
I11
F → (E)·
E
T
F
+
*
(
)
id

Управляющая таблица для LR-анализа

Управляющая таблица T состоит из двух частей: *«действия»* и *«переходы»*.

Строки: все цепочки символов на верхушке стека, которые могут приниматься во внимание в процессе работы распознавателя.

Столбцы:

- Часть «*действия*»: «сдвиг», «свертка», «успех», «ошибка»;
- Часть «*переходы*»: все терминальные и нетерминальные символы грамматики, которые могут появляться на верхушке стека при выполнении действий.

Пример

Исходная грамматика:

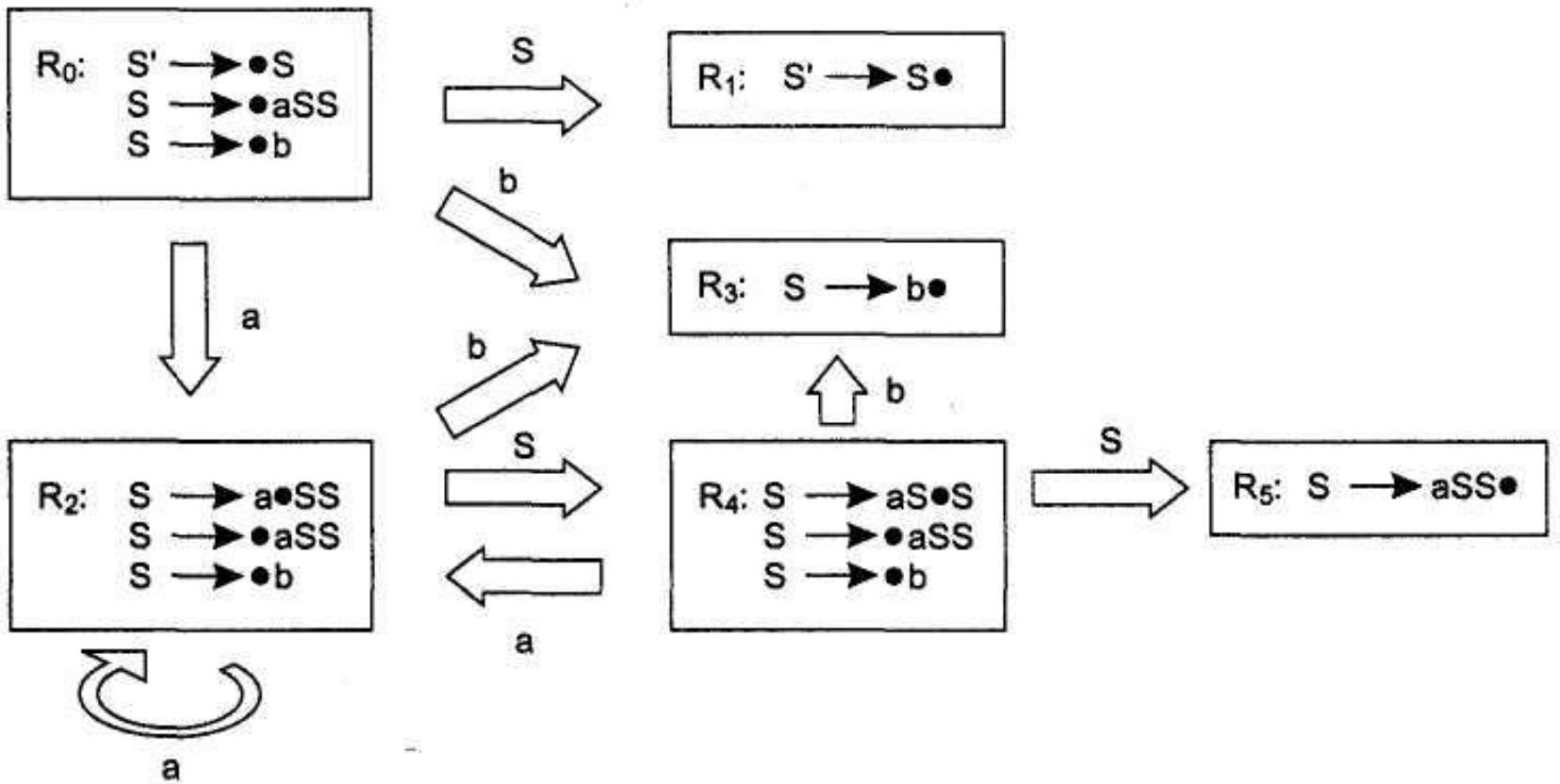
G({a,b}, {S}, P, S)

P: $S \rightarrow aSS \mid b$

Расширенная грамматика:

G′ ({a,b}, {S, S′}, P′, S′}

P′: $S' \rightarrow S$, $S \rightarrow aSS \mid b$



Пример (продолжение)

№	Стек	Действия	Переходы		
			S	a	b
0	$\perp_н$	сдвиг	1	2	3
1	S	успех, 1			
2	a	сдвиг	4	2	3
3	b	свертка, 3			
4	aS	сдвиг	5	2	3
5	aSS	свертка, 2			