Теория и реализация языков программирования

В.А.Серебряков, М.П.Галочкин,
Д.Р.Гончар, М.Г.Фуругян

Оглавление

Глава 1.

Синтаксический анализ

1.0.1. LR(1)-анализаторы

В названии LR(1) символ L указывает на то, что входная цепочка читается слева-направо, R — на то, что строится правый вывод, наконец, 1 указывает на то, что анализатор видит один символ непрочитанной части входной цепочки.

LR(1)-анализ привлекателен по нескольким причинам:

- LR(1)-анализ — наиболее мощный метод анализа без возвратов типа сдвиг-свёртка;
- LR(1)-анализ может быть реализован довольно эффективно;
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования;
- класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы предсказывающими анализаторами (сверху-вниз типа LL(1)).

Схематически структура LR(1)-анализатора изображена на рис. 1.1. Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таб-

лицы анализа (LR(1)-таблицы), которая имеет две части — *функцию действий* (*Action*) и *функцию переходов* (*Goto*). Управляющая программа одна и та же для всех LR(1)-анализаторов, разные анализаторы отличаются только таблицами анализа.

Анализатор читает символы на входной ленте по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2 \dots X_mS_m$ (S_m — верхушка магазина). Каждый X_i — символ грамматики (терминальный или нетерминальный), а S_i — символ *состояния*.

Заметим, что символы грамматики (либо символы состояний) не обязательно должны размещаться в магазине. Однако, их использование облегчает понимание поведения LR-анализатора.

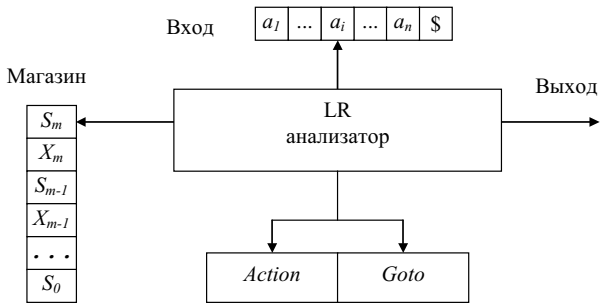

Рис. 1.1.

Элемент функции действий $Action[S_m, a_i]$ для символа состояния S_m и входа $a_i \in T \cup \{\$\}$, может иметь одно из четырех значений:

1) `shift` S (сдвиг), где S — символ состояния,
2) `reduce` $A \to \gamma$ (свертка по правилу грамматики $A \to \gamma$),
3) `accept` (допуск),
4) `error` (ошибка).

Элемент функции переходов $Goto[S_m, A]$ для символа состояния S_m и входа $A \in N$, может иметь одно из двух значений:

1) S, где S — символ состояния,
2) `error` (ошибка).

Конфигурацией LR(1)-анализатора называется пара, первая компонента которой — содержимое магазина, а вторая — непросмотренный вход:

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$$

Эта конфигурация соответствует правой сентенциальной форме

$$X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$$

Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются *активными* префиксами. Основа сентенциальной формы всегда располагается на верхушке магазина. Таким образом, активный префикс — это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.

Когда анализатор начинает работу, в магазине находится только символ начального состояния S_0, на входной ленте — анализируемая цепочка с маркером конца.

Каждый очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m следующим ниже образом.

Пусть LR(1)-анализатор находится в конфигурации

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$$

Анализатор может проделать один из следующих шагов:

1. Если $Action[S_m, a_i] =$ `shift` S, то анализатор выполняет сдвиг, переходя в конфигурацию

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$$

То есть, в магазин помещаются входной символ a_i и символ состояния S, определяемый $Action[S_m, a_i]$. Текущим входным символом становится a_{i+1}.

2. Если $Action[S_m, a_i] =$ `reduce` $A \to \gamma$, то анализатор выполняет свертку, переходя в конфигурацию

$$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$$

где $S = Goto[S_{m-r}, A]$ и r — длина γ, правой части правила вывода.
Анализатор сначала удаляет из магазина $2r$ символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние S_{m-r}. Затем анализатор помещает в магазин A — левую часть правила вывода, и S — символ состояния, определяемый $Goto[S_{m-r}, A]$. На шаге свертки текущий входной символ не меняется. Для LR(1)-анализаторов последовательность символов грамматики $X_{m-r+1} \dots X_m$, удаляемых из магазина , всегда соответствует γ — правой части правила вывода, по которому делается свертка. После осуществления шага свертки генерируется выход LR(1)-анализатора, то есть исполняются семантические действия, связанные с правилом, по которому делается свертка, например, печатаются номера правил, по которым делается свертка.
Заметим, что функция $Goto$ таблицы анализа, построенная по грамматике G, фактически представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы G.

3. Если $Action[S_m, a_i] =$ `accept`, то разбор успешно завершен.

4. Если $Action[S_m, a_i] =$ `error`, то анализатор обнаружил ошибку, и выполняются действия по диагностике и восстановлению.

`Пример 1.1.` Рассмотрим грамматику арифметических выражений $G = (\{E, T, F\}, \{id, +, *\}, P, E)$ с правилами:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to id$

На рис. 1.2 изображены функции *Action* и *Goto*, образующие LR(1)-таблицу для этой грамматики. Элемент `Si` функции *Action* означает сдвиг и помещение в магазин состояния с номером i, `Rj` — свертку по правилу номер j, `acc` — допуск, пустая клетка — ошибку. Для функции *Goto* символ `i` означает помещение в магазин состояния с номером i, пустая клетка — ошибку.

На входе $id+id*id$ последовательность состояний магазина и входной ленты показаны на рис. 1.3. Например, в первой строке LR-анализатор находится в нулевом состоянии и «видит» первый входной символ *id*. Действие `S6` в нулевой строке и столбце *id* в поле *Action* (рис. 1.2) означает сдвиг и помещение символа состояния 6 на верхушку магазина. Это и изображено во второй строке: первый символ *id* и символ состояния 6 помещаются в магазин, а *id* удаляется со входной ленты.

Состояния	*Action*				*Goto*		
	id	+	*	$	E	T	F
0	S6				1	2	3
1		S4		acc			
2		R2	S7	R2			
3		R4	R4	R4			
4	S6					5	3
5		R1	S7	R1			
6		R5	R5	R5			
7	S6						8
8		R3	R3	R3			

Рис. 1.2.

Текущим входным символом становится +, и действием в состоянии 6 на вход + является свертка по $F \to id$. Из магазина удаляются два символа (один символ состояния и один символ грамматики). Затем анализируется нулевое состояние. Поскольку *Goto* в нулевом состоянии по символу F — это 3, F и 3 помещаются в магазин. Теперь имеем конфигурацию, со-

Активный префикс	Магазин	Вход	Действие
	0	$id + id * id\$$	сдвиг
id	0 id 6	$+id * id\$$	$F \to id$
F	0 F 3	$+id * id\$$	$T \to F$
T	0 T 2	$+id * id\$$	$E \to T$
E	0 E 1	$+id * id\$$	сдвиг
$E+$	0 E 1 + 4	$id * id\$$	сдвиг
$E + id$	0 E 1 + 4 id 6	$*id\$$	$F \to id$
$E + F$	0 E 1 + 4 F 3	$*id\$$	$T \to F$
$E + T$	0 E 1 + 4 T 5	$id\$$	сдвиг
$E + T*$	0 E 1 + 4 T 5 * 7	$id\$$	сдвиг
$E + T * id$	0 E 1 + 4 T 5 * 7 id 6	$\$$	$F \to id$
$E + T * F$	0 E 1 + 4 T 5 * 7 F 8	$\$$	$T \to T * F$
$E + T$	0 E 1 + 4 T 5	$\$$	$E \to E + T$
E	0 E 1		допуск

Рис. 1.3.

ответствующую третьей строке. Остальные шаги определяются аналогично.

1.0.2. Конструирование LR(1)-таблицы

Рассмотрим алгоритм конструирования таблицы, управляющей LR(1) - анализатором.

Пусть $G = (N, T, P, S)$ — КС-грамматика. *Пополненной* грамматикой для данной грамматики G называется КС-грамматика

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S'),$$

то есть эквивалентная грамматика, в которой введен новый начальный символ S' и новое правило вывода $S' \to S$.

Это дополнительное правило вводится для того, чтобы определить, когда анализатор должен остановить разбор и зафиксировать допуск входа. Таким образом, допуск имеет

место тогда и только тогда, когда анализатор готов осуществить свертку по правилу $S' \to S$.

LR(1)-*ситуацией* называется пара $[A \to \alpha.\beta, a]$, где $A \to \alpha\beta$ — правило грамматики, a - терминал или правый концевой маркер `$`. Вторая компонента ситуации называется *аванцепочкой*.

Будем говорить, что LR(1)-ситуация $[A \to \alpha.\beta, a]$ *допустима* для активного префикса δ, если существует вывод $S \Rightarrow_r^* \gamma Aw \Rightarrow_r \gamma\alpha\beta w$, где $\delta = \gamma\alpha$ и либо a — первый символ w, либо $w = e$ и $a =$ `$`.

Будем говорить, что ситуация *допустима*, если она допустима для какого-либо активного префикса.

`Пример 1.2.` Дана грамматика $G = (\{S, B\}, \{a, b\}, P, S)$ с правилами

$S \to BB$
$B \to aB \mid b$

Существует правосторонний вывод $S \Rightarrow_r^* aaBab \Rightarrow_r aaaBab$. Легко видеть, что ситуация $[B \to a.B, a]$ допустима для активного префикса $\delta = aaa$, если в определении выше положить $\gamma = aa$, $A = B$, $w = ab$, $\alpha = a$, $\beta = B$. Существует также правосторонний вывод $S \Rightarrow_r^* BaB \Rightarrow_r BaaB$. Поэтому для активного префикса Baa допустима ситуация $[B \to a.B,$ `$`$]$.

Центральная идея метода заключается в том, что по грамматике строится детерминированный конечный автомат, распознающий активные префиксы. Для этого ситуации группируются во множества, которые и образуют состояния автомата. Ситуации можно рассматривать как состояния недетерминированного конечного автомата, распознающего активные префиксы, а их группировка на самом деле есть процесс построения детерминированного конечного автомата из недетерминированного.

Анализатор, работающий слева-направо по типу сдвиг-свертка, должен уметь распознавать основы на верхушке магазина. Состояние автомата после прочтения содержимого магазина и текущий входной символ определяют очередное действие автомата. Функцией переходов этого конечного автомата является функция переходов LR-анализатора.

Чтобы не просматривать магазин на каждом шаге анализа, на верхушке магазина всегда хранится то состояние, в котором должен оказаться этот конечный автомат после того, как он прочитал символы грамматики в магазине от дна к верхушке.

Рассмотрим ситуацию вида $[A \to \alpha.B\beta, a]$ из множества ситуаций, допустимых для некоторого активного префикса z. Тогда существует правосторонний вывод $S \Rightarrow_r^* yAax \Rightarrow_r y\alpha B\beta ax$, где $z = y\alpha$. Предположим, что из βax выводится терминальная строка bw. Тогда для некоторого правила вывода вида $B \to q$ имеется вывод $S \Rightarrow_r^* zBbw \Rightarrow_r zqbw$. Таким образом $[B \to .q, b]$ также допустима для z и ситуация $[A \to \alpha B.\beta, a]$ допустима для активного префикса zB. Здесь либо b может быть первым терминалом, выводимым из β, либо из β выводится e в выводе

$\beta ax \Rightarrow_r^* bw$ и тогда b равно a. То есть b принадлежит $FIRST(\beta ax)$. Построение всех таких ситуаций для данного множества ситуаций, то есть его замыкание, делает приведенная ниже функция `closure`.

Система множеств допустимых LR(1)-ситуаций для всевозможных активных префиксов пополненной грамматики называется *канонической системой* множеств допустимых LR(1)-ситуаций. Алгоритм построения канонической системы множеств приведен ниже.

Алгоритм 4.10. Конструирование канонической системы множеств допустимых LR(1)-ситуаций.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. Каноническая система C множеств допустимых LR(1)-ситуаций для грамматики G.

Метод. Выполнить для пополненной грамматики G' процедуру `items`, которая использует функции `closure` и `goto`.

```
function closure(I){ /* I - множество ситуаций */
  do{
    for (каждой ситуации [A → α.Bβ, a] из I,
         каждого правила вывода B → γ из G',
         каждого терминала b из FIRST(βa),
```

```
        такого, что [B → .γ, b] нет в I)
      добавить [B → .γ, b] к I;
  }
  while (к I можно добавить новую ситуацию);
  return I;
}

function goto(I,X){ /* I - множество ситуаций;
                       X - символ грамматики */
  Пусть J = {[A → αX.β, a] | [A → α.Xβ, a] ∈ I};
  return closure(J);
}

procedure items(G'){ /* G' - пополненная
                        грамматика */
  I_0 = closure({[S' → .S, $]});
  C = {I_0};
  do{
    for (каждого множества ситуаций I из
        системы C, каждого символа грамматики X
        такого, что goto(I, X) не пусто
        и не принадлежит C)
      добавить goto(I, X) к системе C;
  }
  while (к C можно добавить новое множество
        ситуаций);
```

Если I — множество ситуаций, допустимых для некоторого активного префикса δ, то $\texttt{goto}(I, X)$ — множество ситуаций, допустимых для активного префикса δX.

Работа алгоритма построения системы C множеств допустимых LR(1)-ситуаций начинается с того, что в C помещается *начальное* множество ситуаций $I_0 = \texttt{closure}(\{[S' \to .S, \$]\})$. Затем с помощью функции `goto` вычисляются новые множества ситуаций и включаются в C. По-существу, $\texttt{goto}(I, X)$ — переход конечного автомата из состояния I по символу X.

Рассмотрим теперь, как по системе множеств LR(1)-ситуаций строится LR(1)-таблица, то есть функции действий и переходов LR(1)-анализатора.

Алгоритм 4.11. Построение LR(1)-таблицы.

Вход. Каноническая система $C = \{I_0, I_1, \ldots, I_n\}$ множеств допустимых LR(1)-ситуаций для грамматики G.

Выход. Функции *Action* и *Goto*, составляющие LR(1)-таблицу для грамматики G.

Метод. Для каждого состояния i функции $Action[i, a]$ и $Goto[i, X]$ строятся по множеству ситуаций I_i:

(1) Значения функции действия (*Action*) для состояния i определяются следующим образом:

   а) если $[A \to \alpha.a\beta, b] \in I_i$ (a — терминал) и $\texttt{goto}(I_i, a) = I_j$, то полагаем $Action[i, a] = \texttt{shift}\ j$;

   б) если $[A \to \alpha., a] \in I_i$, причем $A \neq S'$, то полагаем $Action[i, a] = \texttt{reduce}\ A \to \alpha$;

   в) если $[S' \to S., \$] \in I_i$, то полагаем $Action[i, \$] = \texttt{accept}$.

(3) Значения функции переходов для состояния i определяются следующим образом: если $\texttt{goto}(I_i, A) = I_j$, то $Goto[i, A] = j$ (здесь A — нетерминал).

(4) Все входы в *Action* и *Goto*, не определенные шагами 2 и 3, полагаем равными `error`.

(5) Начальное состояние анализатора строится из множества, содержащего ситуацию $[S' \to .S, \$]$.

Таблица на основе функций *Action* и *Goto*, полученных в результате работы алгоритма 4.11., называется *канонической* LR(1)-таблицей. Работающий с ней LR(1)-анализатор, называется *каноническим* LR(1)-анализатором.

`Пример 1.3.` Рассмотрим следующую грамматику, являющуюся пополненной для грамматики из примера 4.8.:

(0) $E' \to E$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to id$.

Множества ситуаций и переходы по `goto` для этой грамматики приведены на рис. 1.4. LR(1)-таблица для этой грамматики приведена на рис. 1.2.

Проследим последовательность создания этих множеств более подробно.

1. Вычисляем $I_0 = closure(\{[E' \to .E, \$]\})$.

1.1. Ситуация $[E' \to .E, \$]$ попадает в него по умолчанию как исходная.

1.2. Если обратится к обозначениям функции $closure$, то для неё

$\alpha = \beta = e, \quad B = E, \quad a = \$,$
$first(\beta a) = first(\$) = \{\$\}$.

Значит, для терминала \$ добавляем ситуации на основе правил со знаком E в левой части правила.

Это правила
$E \to E + T$ и $E \to T$
и соответствующие им ситуации

$[E \to .E + T, \$]$ и $[E \to .T, \$]$.

1.3. Просматриваем получившиеся ситуации.
Для ситуации $[E \to .E + T, \$]$
$\beta = +$, поэтому $first(\beta a) = first(+\$) = \{+\}$.
На основе этого добавляем к I_0

$[E \to E + .T, +]$ и $[E \to .T, +]$.

1.4. Для ситуации $[E \to .T, \$]$ $\beta = e, \quad first(\beta a) = \{\$\}$.
Поэтому добавляем к I_0

$[T \to .T * F, \$]$ и $[T \to .F, \$]$.

1.5. Подобно этому для ситуации $[E \to .T, +]$
$\beta = e, \quad first(\beta a) = \{+\}$.
Поэтому добавляем к I_0

$[T \to .T * F, +]$ и $[T \to .F, +]$.

1.6. Из ситуации $[T \to .T * F, +]$
$\beta = *, \ first(\beta a) = \{*\}$.

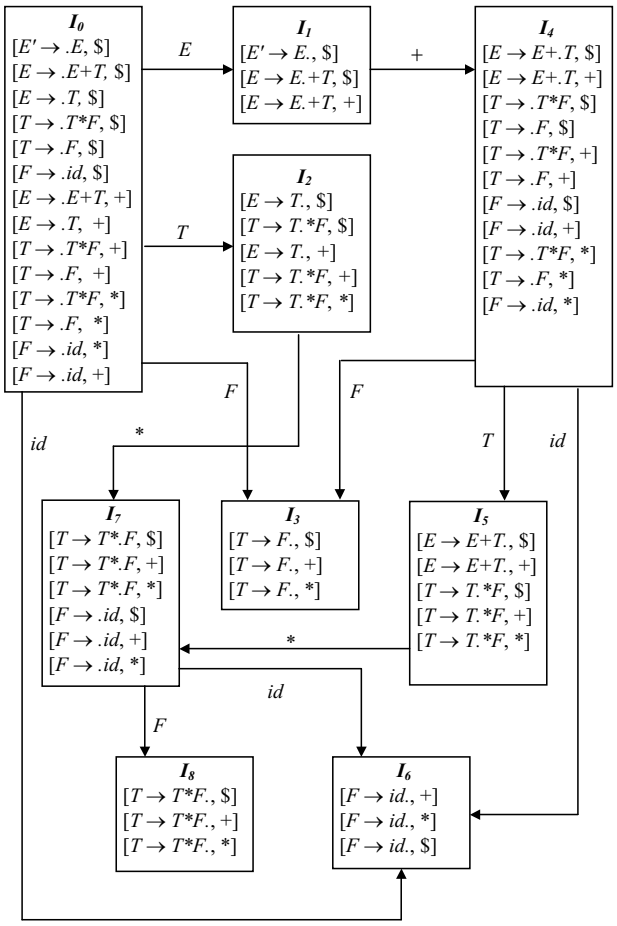

Рис. 1.4.

Поэтому добавляем к I_0

$[T \to .T * F, *]$ и $[T \to .F, *]$.

1.7. Далее из ситуации $[T \to .F, *]$ получаем ситуацию $[F \to .id, *]$.

из ситуации $[T \to .F, \$]$ — ситуацию $[F \to .id, \$]$, а
из ситуации $[T \to .F, +]$ — $[F \to .id, *]$.

Таким образом, все 14 искомых ситуаций I_0 получены.

Возвращаемся в головную функцию *items*, включаем I_0 в множество C и исследуем непустые итоги применения функции $goto(I_0, X)$, где $X \in \{E', E, T, F, +, *, \$, id\}$.

Если посмотреть на вид правил в функции $goto(I_0, X)$, то видно, что X должен встретиться в правой части хотя бы одного правила. Для E' таких правил у нас нет, поэтому значение функции $goto(I_0, E')$ пусто.

Возьмём $goto(I_0, E)$. E встречается после точки в правых частях двух ситуаций из I_0, значит берём эти два правила и переносим в них точки на один символ вправо (пока есть куда - не упёрлись в запятую), получаем:

$[E' \to E., \$]$
и
$[E \to E. + T, \$|+]$

Вычислим от каждой из этих ситуаций функцию *closure*. Но, поскольку справа от точки здесь либо пустая цепочка, либо терминал, то никаких новых ситуаций не возникает. Дальше отслеживаем, может ли куда-то сдвинуться точка дальше на право и по какому символу. Если может, строим соответствующее множество (см. рис. 4.11). И т.д.

1.0.3. LR(1)-грамматики

Если для КС-грамматики G функция *Action*, полученная в результате работы алгоритма 4.11., не содержит неоднозначно определённых входов, то грамматика называется LR(1)-грамматикой.

Язык L называется LR(1)-языком, если он может быть порожден некоторой LR(1)-грамматикой.

Иногда используется другое определение LR(1)-грамматики. Грамматика называется LR(1), если из условий

1. $S' \Rightarrow_r^* uAw \Rightarrow_r uvw$,
2. $S' \Rightarrow_r^* zBx \Rightarrow_r uvy$,
3. $FIRST(w) = FIRST(y)$

следует, что $uAy = zBx$ (то есть $u = z$, $A = B$ и $x = y$).

Согласно этому определению, если uvw и uvy — правовыводимые цепочки пополненной грамматики, у которых

$FIRST(w) = FIRST(y)$ и $A \to v$ — последнее правило, использованное в правом выводе цепочки uvw, то правило $A \to v$ должно применяться и в правом разборе при свертке uvy к uAy. Так как A дает v независимо от w, то LR(1)-условие означает, что в $FIRST(w)$ содержится информация, достаточная для определения того, что uv за один шаг выводится из uA. Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики.

Можно доказать, что эти два определения эквивалентны.

Если грамматика не является LR(1), то анализатор типа сдвиг-свёртка при анализе некоторой цепочки может достигнуть конфигурации, в которой он, зная содержимое магазина и следующий входной символ, не может решить, делать ли сдвиг или свертку (конфликт сдвиг/свертка), или не может решить, какую из нескольких свёрток применить (конфликт свертка/свёртка).

В частности, неоднозначная грамматика не может быть LR(1). Для доказательства рассмотрим два различных правых вывода

(1) $S \Rightarrow_r u_1 \Rightarrow_r \ldots \Rightarrow_r u_n \Rightarrow_r w$, и

(2) $S \Rightarrow_r v_1 \Rightarrow_r \ldots \Rightarrow_r v_m \Rightarrow_r w$.

Нетрудно заметить, что LR(1) - условие (согласно второму определению LR(1)-грамматики) нарушается для наименьшего из чисел i, для которых $u_{n-i} \neq v_{m-i}$.

`Пример 1.4.` Рассмотрим ещё раз грамматику условных операторов:

$$S \to if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$$
$$E \to b$$

Если анализатор типа сдвиг-свертка находится в конфигурации, такой, что необработанная часть входной цепочки имеет вид $else \ldots \$$, а в магазине находится $\ldots if\ E\ then\ S$, то нельзя определить, является ли $if\ E\ then\ S$ основой, вне зависимости от того, что лежит в магазине ниже. Это конфликт сдвиг/свертка. В зависимости от того, что следует на входе за $else$, правильной может быть свертка по $S \to if\ E\ then\ S$

или сдвиг *else*, а затем разбор другого S и завершение основы $if\ E\ then\ S\ else\ S$. Таким образом нельзя сказать, нужно ли в этом случае делать сдвиг или свертку, так что грамматика не является LR(1).

Эта грамматика может быть преобразована к LR(1)-виду следующим образом:

$$S \to M \mid U$$
$$M \to if\ E\ then\ M\ else\ M \mid a$$
$$U \to if\ E\ then\ S \mid if\ E\ then\ M\ else\ U$$
$$E \to b$$

Основная разница между LL(1)- и LR(1)- грамматиками заключается в следующем. Чтобы грамматика была LR(1)-грамматикой, необходимо распознавать вхождение правой части правила вывода, просмотрев все, что выведено из этой правой части и текущий символ входной цепочки. Это требование существенно менее строгое, чем требование для LL(1)-грамматики, когда необходимо определить применимое правило, видя только первый символ, выводимый из его правой части. Таким образом, класс LL(1)-грамматик есть собственный подкласс класса LR(1)-грамматик.

Справедливы также следующие утверждения [2].

`Теорема 1.1.` *Каждый LR(1)-язык является детерминированным КС-языком.*

`Теорема 1.2.` *Если L — детерминированный КС-язык, то существует LR(1)-грамматика, порождающая L.*

`Теорема 1.3.` *Для любой $LR(k)$-грамматики для $k > 1$ существует эквивалентная ей $LR(k-1)$-грамматика.*

Доказано, что проблема определения, порождает ли грамматика LR-язык, является алгоритмически неразрешимой.

1.0.4. Восстановление процесса анализа после синтаксических ошибок

Одним из простейших методов восстановления после ошибки при LR(1)-анализе является следующий. При син-

таксической ошибке просматриваем магазин от верхушки, пока не найдем состояние s с переходом на выделенный нетерминал A. Затем сканируются входные символы, пока не будет найден такой, который допустим после A. В этом случае на верхушку магазина помещается состояние $Goto[s, A]$ и разбор продолжается. Для нетерминала A может иметься несколько таких вариантов. Обычно A — это нетерминал, представляющий одну из основных конструкций языка, например оператор.

При более детальной проработке реакции на ошибки можно в каждой пустой клетке анализатора поставить обращение к своей подпрограмме. Такая подпрограмма может вставлять или удалять входные символы или символы магазина, менять порядок входных символов.

1.0.5. Варианты LR-анализаторов

Часто построенные таблицы для LR(1)-анализатора оказываются довольно большими. Поэтому при практической реализации используются различные методы их сжатия. С другой стороны, часто оказывается, что при построении для языка синтаксического анализатора типа «сдвиг-свертка» достаточно более простых методов. Некоторые из этих методов базируются на основе LR(1)-анализаторов.

Одним из способов такого упрощения является LR(0)-анализ — частный случая LR-анализа, когда ни при построении таблиц, ни при анализе не учитывается аванцепочка.

Еще одним вариантом LR-анализа являются так называемые SLR(1)-анализаторы (Simple LR(1)). Они строятся следующим образом. Пусть $C = \{I_0, I_1, \ldots, I_n\}$ — набор множеств допустимых LR(0)-ситуаций. Состояния анализатора соответствуют I_i. Функции действий и переходов анализатора определяются следующим образом.

1. Если $[A \to u.av] \in I_i$ и $\texttt{goto}(I_i, a) = I_j$, то определим $Action[i, a] = \texttt{shift}\ j$.
2. Если $[A \to u.] \in I_i$, то, для всех $a \in FOLLOW(A)$, $A \neq S'$, определим $Action[i, a] = \texttt{reduce}\ A \to u$

3. Если $[S' \to S.] \in I_i$, то определим $Action[i, \$] = \texttt{accept}$.
4. Если $\texttt{goto}(I_i, A) = I_j$, где $A \in N$, то определим $Goto[i, A] = j$.
5. Остальные входы для функций *Action* и *Goto* определим как `error`.
6. Начальное состояние соответствует множеству ситуаций, содержащему ситуацию $[S' \to .S]$.

Распространённым вариантом LR(1)-анализа является также LALR(1)-анализ. Он основан на объединении (слиянии) некоторых таблиц. Назовем *ядром* множества LR(1)-ситуаций множество их первых компонент (то есть во множестве ситуаций не учитываются аванцепочки). Объединим все множества ситуаций с одинаковыми ядрами, а в качестве аванцепочек возьмем объединение аванцепочек. Функции *Action* и *Goto* строятся очевидным образом. Если функция *Action* таким образом построенного анализатора не имеет конфликтов, то он называется LALR(1)-анализатором (LookAhead LR(1)).Если грамматика является LR(1), то в таблицах LALR(1) анализатора могут появиться конфликты типы свертка-свертка (если одно из объединяемых состояний имело ситуации $[A \to \alpha, a]$ и $[B \to \beta, b]$, а другое $[A \to \alpha, b]$ и $[B \to \beta, a]$, то в LALR(1) появятся ситуации $[A \to \alpha, \{a, b\}]$ и $[B \to \beta, \{b, a\}]$). Конфликты типа сдвиг-свертка появиться не могут, поскольку аванцепочка для сдвига во внимание не принимается.

Глава 2.

Элементы теории перевода

До сих пор мы рассматривали процесс синтаксического анализа только как процесс анализа допустимости входной цепочки. Однако, в компиляторе синтаксический анализ служит основой еще одного важного шага — построения дерева синтаксического анализа. В примерах 4.3 и 4.8 предыдущей главы в процессе синтаксического анализа в качестве выхода выдавалась последовательность примененных правил, на основе которой и может быть построено дерево. Построение дерева синтаксического анализа является простейшим частным случаем *перевода* — процесса преобразования некоторой входной цепочки в некоторую выходную.

`Определение.` Пусть T — входной алфавит, а Π — выходной алфавит. *Переводом* (или *трансляцией*) с языка $L_1 \subseteq T^*$ на язык $L_2 \subseteq \Pi^*$ называется отображение $\tau : L_1 \to L_2$. Если $y = \tau(x)$, то цепочка y называется *выходом* для цепочки x.

Мы рассмотрим несколько формализмов для определения переводов: преобразователи с магазинной памятью и атрибутные грамматики.

2.1. Преобразователи с магазинной памятью

Рассмотрим важный класс абстрактных устройств, называемых преобразователями с магазинной памятью. Эти преобразователи получаются из автоматов с магазинной памятью, если к ним добавить выход и позволить на каждом шаге выдавать выходную цепочку.

Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка $P = (Q, T, \Gamma, \Pi, D, q_0, Z_0, F)$, где все символы имеют тот же смысл, что и в определении МП-автомата, за исключением того, что Π — конечный выходной алфавит, а D — отображение множества $Q \times (T \cup \{e\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Pi^*$.

Определим *конфигурацию* преобразователя P как четверку (q, x, u, y), где $q \in Q$ — состояние, $x \in T^*$ — цепочка на входной ленте, $u \in \Gamma^*$ — содержимое магазина, $y \in \Pi^*$ — цепочка на выходной ленте, выданная вплоть до настоящего момента.

Если множество $D(q, a, Z)$ содержит элемент (r, u, z), то будем писать $(q, ax, Zw, y) \vdash (r, x, uw, yz)$ для любых $x \in T^*$, $w \in \Gamma^*$ и $y \in \Pi^*$. Рефлексивно - транзитивное замыкание отношения $\vdash$ будем обозначать $\vdash^*$.

Цепочку y назовем *выходом* для x, если $(q_0, x, Z_0, e) \vdash^* (q, e, u, y)$ для некоторых $q \in F$ и $u \in \Gamma^*$. *Переводом* (или *трансляцией*), определяемым МП-преобразователем P (обозначается $\tau(P)$), назовем множество

$$\{(x, y) \mid (q_0, x, Z_0, e) \vdash^* (q, e, u, y) \text{ для некоторых } q \in F \text{ и } u \in \Gamma^*\}$$

Будем говорить, что МП-преобразователь P является *детерминированным* (ДМП-преобразователем), если выполняются следующие условия:

1) для всех $q \in Q$, $a \in T \cup \{e\}$ и $Z \in \Gamma$ множество $D(q, a, Z)$ содержит не более одного элемента,
2) если $D(q, e, Z) \neq \varnothing$, то $D(q, a, Z) = \varnothing$ для всех $a \in T$.

Пример 2.1. Рассмотрим перевод τ, отображающий каждую цепочку $x \in \{a, b\}^*\$$, в которой число вхождений символа a равно числу вхождений символа b, в цепочку $y = (ab)^n$, где n — число вхождений a или b в цепочку x. Например, $\tau(abbaab\$) = ababab$.

Этот перевод может быть реализован ДМП-преобразователем $P = (\{q_0, q_f\}, \{a, b, \$\}, \{Z, a, b\}, \{a, b\}, D, q_0, Z, \{q_f\})$ с функцией переходов:

$D(q_0, X, Z) = \{(q_0, XZ, e)\}$, $X \in \{a, b\}$,
$D(q_0, \$, Z) = \{(q_f, Z, e)\}$,
$D(q_0, X, X) = \{(q_0, XX, e)\}$, $X \in \{a, b\}$,
$D(q_0, X, Y) = \{(q_0, e, ab)\}$, $X \in \{a, b\}$, $Y \in \{a, b\}$, $X \neq Y$.

2.2. Атрибутные грамматики

Среди всех формальных методов описания языков программирования атрибутные грамматики (введенные Кнутом [6]) получили, по-видимому, наибольшую известность и распространение. Причиной этого является то, что формализм атрибутных грамматик основывается на дереве разбора программы в КС-грамматике, что сближает его с хорошо разработанной теорией и практикой построения трансляторов.

2.2.1. Определение атрибутных грамматик

Атрибутной грамматикой называется четверка $AG = (G, A_S, A_I, R)$, где

(1) $G = (N, T, P, S)$ — приведенная КС-грамматика;

(2) A_S — конечное множество *синтезируемых атрибутов*;

(3) A_I — конечное множество *наследуемых атрибутов*, $A_S \cap A_I = \varnothing$;

(4) R — конечное множество *семантических правил*.

Атрибутная грамматика AG сопоставляет каждому символу X из $N \cup T$ множество $A_S(X)$ синтезируемых атрибутов и множество $A_I(X)$ наследуемых атрибутов. Множество всех синтезируемых атрибутов всех символов из $N \cup T$ обозначается A_S, наследуемых — A_I. Атрибуты разных символов являются различными атрибутами. Будем обозначать атрибут a символа X как $a(X)$. Значения атрибутов могут быть произвольных типов, например, представлять собой числа, строки, адреса памяти и т.д.

Пусть правило p из P имеет вид $X_0 \to X_1 X_2 \dots X_n$. Атрибутная грамматика AG сопоставляет каждому правилу p из P конечное множество $R(p)$ семантических правил вида

$$a(X_i) = f(b(X_j), c(X_k), \dots, d(X_m))$$

где $0 \leqslant j, k, \dots, m \leqslant n$, причем $1 \leqslant i \leqslant n$, если $a(X_i) \in A_I(X_i)$ (то есть $a(X_i)$ — наследуемый атрибут), и $i = 0$, если $a(X_i) \in A_S(X_i)$ (то есть $a(X_i)$ — синтезируемый атрибут).

Таким образом, семантическое правило определяет значение атрибута a символа X_i на основе значений атрибутов b, c, …, d символов X_j, X_k, …, X_m соответственно.

В частном случае длина n правой части правила может быть равна нулю, тогда будем говорить, что атрибут a символа X_i «получает в качестве значения константу».

В дальнейшем будем считать, что атрибутная грамматика не содержит семантических правил для вычисления атрибутов терминальных символов. Предполагается, что атрибуты терминальных символов — либо предопределенные константы, либо доступны как результат работы лексического анализатора.

`Пример 2.2.` Рассмотрим атрибутную грамматику, позволяющую вычислить значение вещественного числа, представленного в десятичной записи. Здесь $N = \{Num, Int, Frac\}$, $T = \{digit, .\}$, $S = Num$, а правила вывода и семантические правила определяются следующим образом (верхние индексы используются для ссылки на разные вхождения одного и того же нетерминала):

$Num \to Int\,.\,Frac$ $\quad v(Num) = v(Int) + v(Frac)$
$\quad p(Frac) = 1$

$Int \to e$ $\quad v(Int) = 0$
$\quad p(Int) = 0$

$Int^1 \to digit\, Int^2$ $\quad v(Int^1) = v(digit) * 10^{p(Int^2)} + v(Int^2)$
$\quad p(Int^1) = p(Int^2) + 1$

$Frac \to e$ $\quad v(Frac) = 0$

$Frac^1 \to digit Frac^2$ $\quad v(Frac^1) = v(digit) * 10^{-p(Frac^1)} + v(Frac^2)$
$\quad p(Frac^2) = p(Frac^1) + 1$

Для этой грамматики

$$A_S(Num) = \{v\}, \quad A_I(Num) = \varnothing,$$
$$A_S(Int) = \{v, p\}, \quad A_I(Int) = \varnothing,$$
$$A_S(Frac) = \{v\}, \quad A_I(Frac) = \{p\}.$$

Пусть дана атрибутная грамматика AG и цепочка, принадлежащая языку, определяемому соответствующей $G = (N, T, P, S)$. Сопоставим этой цепочке «значение» следующим образом. Построим дерево разбора T этой цепочки в грамматике G. Каждый внутренний узел этого дерева помечается нетерминалом X_0, соответствующим применению p-го правила грамматики; таким образом, у этого узла будет n непосредственных потомков (рис. 2.1).

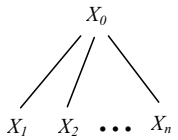

Рис. 2.1.

Пусть теперь X — метка некоторого узла дерева и пусть a — атрибут символа X. Если a — синтезируемый атрибут, то $X = X_0$ для некоторого $p \in P$; если же a — наследуемый

атрибут, то $X = X_j$ для некоторых $p \in P$ и $1 \leqslant j \leqslant n$. В обоих случаях дерево «в районе» этого узла имеет вид, приведенный на рис. 2.1. По определению, атрибут a имеет в этом узле значение v, если в соответствующем семантическом правиле

$$a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$$

все атрибуты b, c, $\ldots$, d уже определены и имеют в узлах с метками X_j, X_k, $\ldots$, X_m значения v_j, v_k, $\ldots$, v_m соответственно, а $v = f(v_1, v_2, \ldots, v_m)$. Процесс вычисления атрибутов на дереве продолжается до тех пор, пока нельзя будет вычислить больше ни одного атрибута. Вычисленные в результате атрибуты корня дерева представляют собой «значение», соответствующее данному дереву вывода.

Заметим, что значение синтезируемого атрибута символа в узле синтаксического дерева вычисляется по атрибутам символов в потомках этого узла; значение наследуемого атрибута вычисляется по атрибутам «родителя» и «соседей».

Атрибуты, сопоставленные вхождениям символов в дерево разбора, будем называть *вхождениями* атрибутов в дерево разбора, а дерево с сопоставленными каждой вершине атрибутами — *атрибутированным деревом разбора*.

`Пример 2.3.` Атрибутированное дерево для грамматики из предыдущего примера и цепочки $w = 12.34$ показано на рис. 2.2.

Будем говорить, что семантические правила *заданы корректно*, если они позволяют вычислить все атрибуты произвольного узла в любом дереве вывода.

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам. Эти зависимости могут быть представлены в виде ориентированного графа следующим образом.

Пусть T — дерево разбора. Сопоставим этому дереву ориентированный граф $D(T)$, узлами которого являются пары (n, a), где n — узел дерева T, a — атрибут символа,

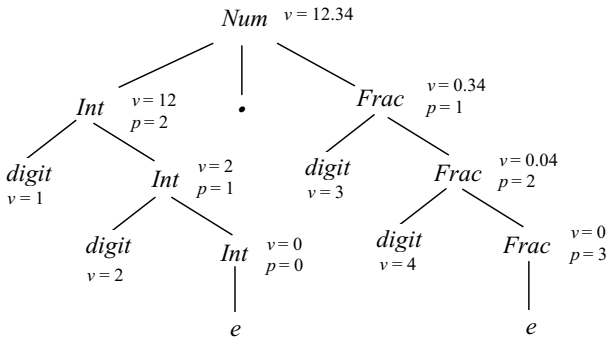

Рис. 2.2.

служащего меткой узла n. Граф содержит дугу из (n_1, a_1) в (n_2, a_2) тогда и только тогда, когда семантическое правило, вычисляющее атрибут a_2, непосредственно использует значение атрибута a_1. Таким образом, узлами графа $D(T)$ являются атрибуты, которые нужно вычислить, а дуги определяют зависимости, подразумевающие, какие атрибуты вычисляются раньше, а какие позже.

`Пример 2.4.` Граф зависимостей атрибутов для дерева разбора из предыдущего примера показан на рис. 2.3.

Можно показать, что семантические правила являются корректными тогда и только тогда, когда для любого дерева вывода T соответствующий граф $D(T)$ не содержит циклов (то есть является ориентированным ациклическим графом).

2.2.2. Классы атрибутных грамматик и их реализация

В общем виде реализация вычислителей для атрибутных грамматик вызывает значительные трудности. Это связано с тем, что множество значений атрибутов, связанных с данным деревом, приходится вычислять в соответ-

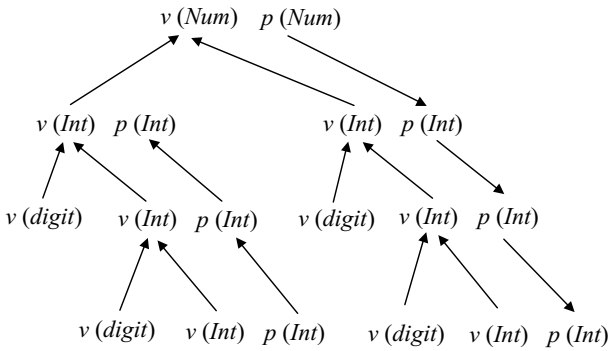

Рис. 2.3.

ствии с зависимостями атрибутов, которые образуют ориентированный ациклический граф. На практике стараются осуществлять процесс вычисления атрибутов, привязывая его к тому или иному способу обхода дерева. Рассматривают многовизитные, многопроходные и другие атрибутные вычислители. Это, как правило, ведет к ограничению допустимых зависимостей между атрибутами, поддерживаемых вычислителем.

Простейшими подклассами атрибутных грамматик, вычисления всех атрибутов для которых может быть осуществлено одновременно с синтаксическим анализом, являются S-атрибутные и L-атрибутные грамматики.

`Определение.` Атрибутная грамматика называется S-атрибутной, если она содержит только синтезируемые атрибуты.

Нетрудно видеть, что для S-атрибутной грамматики на любом дереве разбора все атрибуты могут быть вычислены за один обход дерева снизу вверх. Таким образом, вычисление атрибутов можно делать параллельно с восходящим синтаксическим анализом, например, LR(1)-анализом.

`Пример 2.5.` Рассмотрим S-атрибутную грамматику для перевода арифметических выражений в ПОЛИЗ. Здесь атрибут v имеет строковый тип, $\|$ — обозначает операцию конкатенации. Правила вывода и семантические правила определяются следующим образом

$$E^1 \to E^2 + T \quad v(E^1) = v(E^2) \,\|\, v(T) \,\|\, '+'$$

$$E \to T \quad v(E) = v(T)$$

$$T \to T * F \quad v(T^1) = v(T^2) \,\|\, v(F) \,\|\, '*'$$

$$T \to F \quad v(T) = v(F)$$

$$F \to id \quad v(F) = v(id)$$

$$F \to (E) \quad v(F) = v(E)$$

`Определение.` Атрибутная грамматика называется L-атрибутной, если любой наследуемый атрибут любого символа X_j из правой части каждого правила $X_0 \to X_1 X_2 \dots X_n$ грамматики зависит только от

(1) атрибутов символов X_1, X_2, …, X_{j-1}, находящихся в правиле слева от X_j, и
(2) наследуемых атрибутов символа X_0.

Заметим, что каждая S-атрибутная грамматика является L-атрибутной. Все атрибуты на любом дереве для L-атрибутной грамматики могут быть вычислены за один обход дерева сверху-вниз слева-направо. Таким образом, вычисление атрибутов можно осуществлять параллельно с нисходящим синтаксическим анализом, например, LL(1)-анализом или рекурсивным спуском.

В случае рекурсивного спуска в каждой функции, соответствующей нетерминалу, надо определить формальные параметры, передаваемые по значению, для наследуемых атрибутов, и формальные параметры, передаваемые по ссылке, для синтезируемых атрибутов. В качестве примера рассмотрим реализацию атрибутной грамматики из

примера 5.5 (нетрудно видеть, что грамматика является L-атрибутной).

```
void int_part(float * V0, int * P0)
{if (Map[InSym]==Digit)
  {  int I=InSym;
     float V2;
     int P2;
     InSym=getInSym();
     int_part(&V2,&P2);
     *V0=I*exp(P2*ln(10))+V2;
     *P0=P2+1;
  }
  else {*V0=0;
        *P0=0;
     }
}
void fract_part(float * V0, int P0)
{if (Map[InSym]==Digit)
  {  int I=InSym;
     float V2;
     int P2=P0+1;
     InSym=getInSym();
     fract_part(&V2,P2);
     *V0=I*exp(-P0*ln(10))+V2;
  }
  else {*V0=0;
     }
}
void number()
{ float V1,V3,V0;
  int P;
   int_part(&V1,&P);
   if (InSym!='.') error();
   fract_part(&V3,1);
   V0=V1+V3;
}
```

2.2.3. Язык описания атрибутных грамматик

Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования. Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сталкивается с большими трудностями. В связи с этим было сделано множество попыток рассматривать те или иные классы атрибутных грамматик, обладающих «хорошими» свойствами. К числу таких свойств относятся прежде всего простота алгоритма проверки атрибутной грамматики на зацикленность и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса.

Атрибутные грамматики использовались для описания семантики языков программирования и было создано несколько систем автоматизации разработки трансляторов, основанных на формализме атрибутных грамматик. Опыт их использования показал, что «чистый» атрибутный формализм может быть успешно применен для описания семантики языка, но его использование вызывает трудности при создании транслятора. Эти трудности связаны как с самим формализмом, так и с некоторыми технологическими проблемами. К трудностям первого рода можно отнести несоответствие чисто функциональной природы атрибутного вычислителя и связанной с ней неупорядоченностью процесса вычисления атрибутов (что в значительной степени является преимуществом этого формализма) и упорядоченностью элементов программы. Это несоответствие ведет к тому, что приходится идти на искусственные приемы для их сочетания. Технологические трудности связаны с эффективностью трансляторов, полученных с помощью атрибутных систем. Как правило, качество таких трансляторов довольно низко из-за больших расходов памяти, неэффективности искусственных приемов, о которых было сказано выше.

Учитывая это, мы будем вести дальнейшее изложение на языке, сочетающем особенности атрибутного формализ-

ма и обычного языка программирования, в котором предполагается наличие операторов, а значит, и возможность управления порядком исполнения операторов. Этот порядок может быть привязан к обходу атрибутированного дерева разбора сверху вниз слева направо. Что касается грамматики входного языка, то мы не будем предполагать принадлежность ее определенному классу (например, LL(1) или LR(1)). Будем считать, что дерево разбора входной программы уже построено как результат синтаксического анализа и атрибутные вычисления осуществляются в результате обхода этого дерева. Таким образом, входная грамматика атрибутного вычислителя может быть даже неоднозначной, что не влияет на процесс атрибутных вычислений.

При записи синтаксиса мы будем использовать расширенную БНФ. Элемент правой части синтаксического правила, заключенный в скобки `[ ]`, может отсутствовать. Элемент правой части синтаксического правила, заключенный в скобки `( )`, означает возможность повторения один или более раз. Элемент правой части синтаксического правила, заключенный в скобки `[()]`, означает возможность повторения ноль или более раз. В скобках `[ ]` или `[()]` может указываться разделитель конструкций.

Ниже дан синтаксис языка описания атрибутных грамматик. Приведен только синтаксис конструкций, собственно описывающих атрибутные вычисления. Синтаксис обычных выражений и операторов не приводится — он основывается на Си.

```
Атрибутная грамматика::='ALPHABET'
        ( ОписаниеНетерминала ) ( Правило )
ОписаниеНетерминала::=ИмяНетерминала
               '::' [( ОписаниеАтрибутов / ';')]'.'
ОписаниеАтрибутов::=Тип ( ИмяАтрибута / ',')
Правило::='RULE' Синтаксис 'SEMANTICS' Семантика'.'
Синтаксис::=ИмяНетерминала '::=' ПраваяЧасть
ПраваяЧасть::=[( ЭлементПравойЧасти )]
ЭлементПравойЧасти::=ИмяНетерминала
```

```
        | Терминал
        | '(' Нетерминал [ '/' Терминал ] ')'
        | '[' Нетерминал ']'
        | '[(' Нетерминал [ '/' Терминал ] ')]'
Семантика::=[(ЛокальноеОбъявление / ';')]
           [( СемантическоеДействие / ';')]
СемантическоеДействие::=Присваивание
          | [ Метка ] Оператор
Присваивание::=Переменная ':=' Выражение
Переменная::=ЛокальнаяПеременная
        |   Атрибут
Атрибут::=ЛокальныйАтрибут
     |   ГлобальныйАтрибут
ЛокальныйАтрибут::=ИмяАтрибута '<' Номер '>'
ГлобальныйАтрибут::=ИмяАтрибута '<' Нетерминал '>'
Метка::=Целое ':'
     |  Целое 'E' ':'
     |  Целое 'A' ':'
Оператор::=Условный
       |   ОператорПроцедуры
       |   ЦиклПоМножеству
       |   ПростойЦикл
       |   ЦиклСУсловиемОкончания
```

Описание атрибутной грамматики состоит из раздела описания атрибутов и раздела правил. Раздел описания атрибутов определяет состав атрибутов для каждого символа грамматики и тип каждого атрибута. Правила состоят из синтаксической и семантической части. В синтаксической части используется расширенная БНФ. Семантическая часть правила состоит из локальных объявлений и семантических действий. В качестве семантических действий допускаются как атрибутные присваивания, так и составные операторы.

Метка в семантической части правила привязывает выполнение оператора к обходу дерева разбора сверху-вниз слева направо. Конструкция `i : оператор` означает, что оператор должен быть выполнен сразу после обхода i-й

компоненты правой части. Конструкция `i E : оператор` означает, что оператор должен быть выполнен, только если порождение i-й компоненты правой части пусто. Конструкция `i A : оператор` означает, что оператор должен быть выполнен после разбора каждого повторения i-й компоненты правой части (имеется в виду конструкция повторения).

Каждое правило может иметь локальные определения (типов и переменных). В формулах используются как атрибуты символов данного правила (*локальные* атрибуты) и в этом случае соответствующие символы указываются номерами в правиле (0 — для символа левой части, 1 — для первого символа правой части, 2 — для второго символа правой части и т.д.), так и атрибуты символов предков левой части правила (*глобальные* атрибуты). В этом случае соответствующий символ указывается именем нетерминала. Таким образом, на дереве образуются области видимости атрибутов: атрибут символа имеет область видимости, состоящую из правила, в которое символ входит в правую часть, плюс все поддерево, корнем которого является символ, за исключением поддеревьев — потомков того же символа в этом поддереве.

Значение терминального символа доступно через атрибут `VAL` соответствующего типа.

`Пример 2.6.` Атрибутная грамматика из примера 5.5 записывается следующим образом:

```
ALPHABET
Num    :: float V.
Int    :: float V;
          int P.
Frac   :: float V;
          int P.
digit  :: int VAL.

RULE
Num ::= Int '.' Frac
SEMANTICS
V<0>=V<1>+V<3>; P<3>=1.
```

```
RULE
Int ::= e
SEMANTICS
V<0>=0; P<0>=0.

RULE
Int ::= digit Int
SEMANTICS
V<0>=VAL<1>*10**P<2>+V<2>; P<0>=P<2>+1.

RULE
Frac ::= e
SEMANTICS
V<0>=0.

RULE
Frac ::= digit Frac
SEMANTICS
V<0>=VAL<1>*10**(-P<0>)+V<2>; P<2>=P<0>+1.
```

Литература

1. Адельсон-Вельский Г.М., Ландис Е.М. *Один алгоритм организации информации* // ДАН СССР. 1962. Т. 146. N 2. С. 263–266.

2. Ахо А., Ульман Д. *Теория синтаксического анализа, перевода и компиляции, в 2-х т.* М.: Мир, 1978.

3. Бездушный А.Н., Лютый В.Г., Серебряков В.А. *Разработка компиляторов в системе СУПЕР.* М.: ВЦ АН СССР, 1991.

4. Грис Д. *Конструирование компиляторов для цифровых вычислительных машин.* М.: Мир, 1975.

5. Кнут Д. *Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы.* М.: Мир, 1976.

6. Кнут Д. *Семантика контекстно-свободных языков* // Семантика языков программирования. М.: Мир, 1980.

7. Курочкин В.М. *Алгоритм распределения регистров для выражений за один обход дерева вывода.* 2 Всес. конф. «Автоматизация производства ППП и трансляторов». 1983. С. 104–105.

8. Лавров С.С., Гончарова Л.И. *Автоматическая обработка данных. Хранение информации в памяти ЭВМ.* М.: Наука, 1971.

9. Надежин Д.Ю., Серебряков В.А., Ходукин В.М. *Промежуточный язык Лидер (предварительное сообщение)* // Обработка символьной информации. М.: ВЦ АН СССР, 1987. С. 50–63.

10. Aho A., Sethi R., Ullman J. *Compilers: principles, techniques and tools*. N.Y.: Addison-Wesley, 1986.

11. Aho A.U., Ganapathi M., Tjiang S.W. *Code generation using tree matching and dynamic programing* // ACM Trans. Program. Languages and Systems. 1989. V. 11. N 4.

12. Bezdushny A., Serebriakov V. *The use of the parsing method for optimal code generation and common subexpression elimination* // Techn. et Sci. Inform. 1993. V. 12. N. 1. P. 69–92.

13. Emmelman H., Schroer F.W., Landweher R. *BEG — a generator for efficient back-ends* // ACM SIGPLAN. 1989. V. 11. N 4. P. 227–237.

14. Fraser C.W., Hanson D.R. *A Retargetable compiler for ANSI C* // SIGPLAN Notices. 1991. V. 26.

15. Graham S.L., Harrison M.A., Ruzzo W.L. *An improved context-free recognizer* // ACM Trans. Program. Languages and Systems. 1980. N. 2.

16. Harrison M.A. *Introduction to formal language theory.* Reading, Mass.: Addison-Wesley, 1978.