

7. Восходящие анализаторы

- Восходящие анализаторы. LR (k)-анализаторы
- Построение LR (0)-анализатора
- LR (1)-анализатор. LALR-анализаторы
- Неоднозначные грамматики. Различные типы конфликтов
- Разрешение конфликтов

Лекция 7. Восходящие анализаторы

В этой лекции рассматриваются следующие вопросы:

- Восходящие анализаторы
- LR(k)-анализаторы
- Построение LR(0)-анализатора
- LR(1)-анализатор. LALR-анализаторы
- Неоднозначные грамматики. Различные типы конфликтов
- Разрешение конфликтов

Восходящие анализаторы

$S \rightarrow aABe$

$A \rightarrow Abc$

$A \rightarrow b$

$B \rightarrow d$

Свертка цепочки $abbcde$ в аксиому S :

$abbcde, aAbcde, aAde, aABe, S.$

Правый вывод цепочки:

$S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abbcde$

Восходящие анализаторы

Восходящий анализатор (bottom-up parsing) предназначен для построения дерева разбора, начиная с листьев и двигаясь вверх к корню дерева разбора. Мы можем представить себе этот процесс как "свертку" исходной строки w к аксиоме грамматики. Каждый шаг свертки заключается в сопоставлении некоторой подстроки w и правой части какого-то правила грамматики и замене этой подстроки на нетерминал, являющийся левой частью правила. Если на каждом шаге подстрока выбирается правильно, то в результате мы получим правый вывод строки w .

Пример. Рассмотрим грамматику

$S \rightarrow aABe$

$A \rightarrow Abc$

$A \rightarrow b$

$B \rightarrow d$

Цепочка $abbcde$ может быть свернута в аксиому следующим образом:

$abbcde, aAbcde, aAde, aABe, S.$

Фактически, эта последовательность представляет собой правый вывод этой цепочки, рассматриваемый справа налево:

$S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abbcde.$

LR(k)-анализатор

LR(k) означает, что

- входная цепочка обрабатывается слева направо
- выполняется правый вывод
- не более k символов цепочки используются для принятия решения

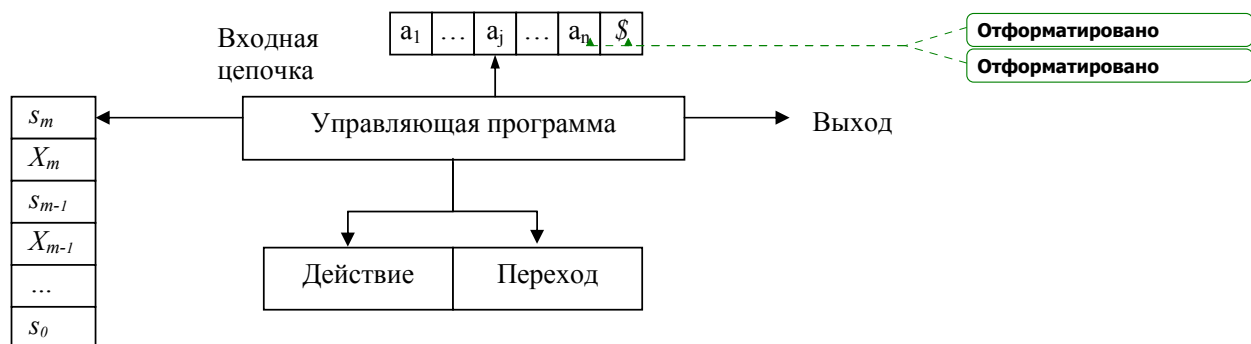
LR(k)-анализатор

LR(k) означает, что

- входная цепочка обрабатывается слева направо (left-to-right parse);
- выполняется правый вывод (rightmost derivation);
- не более k символов цепочки (k-token lookahead) используются для принятия решения.

При LR(k)-анализе применяется метод "перенос-свертка" (*shift-reduce*). Этот метод использует магазинный автомат. Суть метода сводится к следующему. Символы входной цепочки переносятся в магазин до тех пор, пока на вершине магазина не накопится цепочка, совпадающая с правой частью какого-нибудь из правил (операция "перенос", "*shift*"). Далее все символы этой цепочки извлекаются из магазина и на их место помещается нетерминал, находящийся в левой части этого правила (операция "свертка", "*reduce*"). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнения операции свертка, в магазине окажется только аксиома грамматики.

Анализатор состоит из входной цепочки, выхода, магазина, управляющей программы и таблицы, которая имеет две части (действие и переход). Схема такого анализатора выглядит следующим образом:



Управляющая программа анализатора

- Управляющая программа одинакова для всех LR-анализаторов
- Рассматривается пара: s_m – текущее состояние на вершине магазина, a_i – текущий входной символ; после этого вычисляется action $[s_m, a_i]$:
 1. shift s , где s – состояние,
 2. свертка по правилу $A \rightarrow \beta$,
 3. допуск (accept)
 4. ошибка.

Управляющая программа анализатора

Управляющая программа одинакова для всех LR-анализаторов, а таблица изменяется от одного анализатора к другому. Программа анализатора читает последовательно символы входной цепочки. Программа использует магазин для запоминания строки следующего вида $s_0 X_1 s_1 X_2 \dots X_m s_m$, где s_m – вершина магазина. Каждый X_i – символ грамматики, а s_i – символ, называемый состоянием. Каждое состояние суммирует информацию, содержащуюся в стеке перед ним. Комбинация символа состояния на вершине магазина и текущего входного символа используется для индексирования управляющей таблицы и определения операции переноса-свертки. При реализации грамматические символы не обязательно располагаются в магазине; однако, мы будем использовать их при обсуждении для лучшего понимания поведения LR-анализатора.

Программа, управляющая LR-анализатором, ведет себя следующим образом. Рассматривается пара: s_m – текущее состояние на вершине магазина, a_i – текущий входной символ; после этого вычисляется action $[s_m, a_i]$, которое может иметь одно из четырех значений:

1. shift s , где s – состояние,
2. свертка по правилу $A \rightarrow \beta$,
3. допуск (accept)
4. ошибка.

Функция **goto** получает состояние и символ грамматики и выдает состояние. Функция **goto**, строящаяся по грамматике G , есть функция переходов детерминированного магазинного автомата, который распознает язык, порождаемый грамматикой G .

Управляющая программа выглядит следующим образом:

```
Установить ip на первый символ входной цепочки w$;
while (цепочка не закончилась)
{
    Пусть s - состояние на вершине магазина,
    a - символ входной цепочки, на который указывает ip.
    if (action [s, a] == shift s')
    {
        push (a);
        push (s');
        ip++;
    }
    else if (action [s, a] == reduce A→β)
    {
        for (i=1; i≤|β|; i++)
        {
            pop ();
            pop ();
        }
        Пусть s' - состояние на вершине магазина;
        push (A);
        push (goto [s', A]);
        Вывод правила (A→β);
    }
    else if (action [s, a] == accept)
    {
        return success;
    }
    else
    {
        error ();
    }
}
```

Управляющая таблица LR(0)-анализатора

- LR(k)-анализатор использует содержимое магазина и очередные k символов входной цепочки для принятия решения о том, какие действия он должен выполнить.
- LR(0)-анализатор использует только содержимое магазина.

Управляющая таблица LR(0)-анализатора

Обсудим подробно алгоритм построения управляющей таблицы на примере LR(0)-анализаторов.

Заметим, что LR(0)-анализатор принимает решение о своих действиях только на основании содержимого магазина, не учитывая символы входной цепочки. Для иллюстрации построения таблиц LR(0)-анализатора мы будем использовать грамматику G_0 :

- (1) $S \rightarrow (L)$
- (2) $S \rightarrow x$
- (3) $L \rightarrow S$
- (4) $L \rightarrow L, S$

Определение. Пусть $G = (V_T, V_N, P, S)$ – КС-грамматика. *Пополненной грамматикой* (augmented grammar) будем называть грамматику $G' = (V_T, V_N + \{S'\}, P + \{S' \rightarrow S\}, S')$, где S' – нетерминал, не принадлежащий множеству N .

Определение. Пусть $G = (V_T, V_N, P, S)$ – КС-грамматика. Будем называть $[A \rightarrow w_1 w_2, u]$ *LR(k)-ситуацией* (*LR(k)-item*), если $A \rightarrow w_1 w_2$ является правилом из P и u – цепочка терминалов, длина которой не превосходит k .

Понятно, что LR(0)-ситуации не должны содержать терминальной цепочки, то есть мы можем записывать их следующим образом: $[A \rightarrow w_1 w_2]$.

Далее мы рассмотрим поведение анализатора грамматики при разборе входной цепочки.

Состояния 0 и 1

- Состояние 0 определяется множеством ситуаций:
 $\{[S' \Rightarrow .S], [S \Rightarrow .x], [S \Rightarrow .(L)]\}$
- Состояние 1 определяется множеством ситуаций:
 $\{[S \Rightarrow x.]\}$

Состояния 0 и 1

В начале работы магазин пуст (на самом деле, на вершине магазина находится маркер конца \$), и указатель входной цепочки находится перед ее первым символом. Этому состоянию соответствует ситуация $[S' \rightarrow .S]$.

Значит, входная цепочка может начинаться с любого терминального символа, с которого начинается правая часть любого правила с левой частью S . Мы укажем это следующим образом:

$[S' \rightarrow .S]$ $[S \rightarrow .x]$ $[S \rightarrow .(L)]$

Состояние автомата определяется множеством ситуаций. Назовем это состояние 0.

Теперь мы должны выяснить, что произойдет, если анализатор выполнит перенос или свертку. Предположим, что мы выполним перенос x (то есть на вершине магазина окажется x). Этому случаю соответствует ситуация $[S \rightarrow x.]$. Понятно, что правила $S' \rightarrow S$ и $S \rightarrow (L)$ не могут быть применены, поэтому мы их игнорируем. Таким образом, новое состояние, в которое автомат перейдет после переноса в магазин символа x , определяется ситуацией

$[S \rightarrow x.]$

Это состояние назовем 1.

Состояния 2 и 3

- Состояние 2 определяется множеством ситуаций:
 $\{[S \Rightarrow (.L)], [L \Rightarrow .L, S], [L \Rightarrow .S], [S \Rightarrow .(L)], [S \Rightarrow .x]\}$
- В состояние 3 переход происходит из состояния 0 по нетерминалу S .

Состояния 2 и 3

Теперь предположим, что выполнен перенос открывающей круглой скобки. Этому случаю соответствует ситуация $[S \rightarrow (.L)]$. То есть на вершине магазина окажется открывающая круглая скобка, а входная цепочка должна начинаться с некоторой цепочки, которая выводится из L и перед которой находится открывающая круглая скобка. Таким образом, к нашей ситуации мы должны добавить все ситуации, получающиеся из правил, левая часть которых суть нетерминал L , т.е. $[L \rightarrow .L, S]$ и $[L \rightarrow .S]$. Помимо этого, поскольку правая часть правила $L \rightarrow S$ начинается нетерминалом S , мы должны добавить все ситуации, получающиеся из правил, левая часть которых суть нетерминал S , т.е. $[S \rightarrow .L]$ и $[S \rightarrow .x]$. Таким образом, новое состояние, в которое автомат перейдет после переноса в магазин открывающей круглой скобки, определяется ситуациями:

$[S \rightarrow (.L)]$
$[L \rightarrow .L, S]$
$[L \rightarrow .S]$
$[S \rightarrow .(L)]$
$[S \rightarrow .x]$

Это состояние 2. Мы можем изобразить часть первой строки таблицы переходов автомата:

	()	x	,	\$
⋮					
0	s_3		s_2		

Понятно, что в состоянии 0 свертка выполняться не может.

Обсудим, что произойдет, если в состоянии 0 мы оказались после анализа некоторой цепочки, которая выводится из аксиомы грамматики. Это может случиться, если после переноса x или открывающей круглой скобки произошла свертка по правилу, левая часть которого – S . Все символы правой части такого правила будут извлечены из магазина, и анализатор будет выполнять переход для символа S в состоянии 0. Этому случаю соответствует ситуация $[S' \rightarrow S.S]$, определяющая состояние 3.

Базовые операции

- Для построения множества состояний автомата необходимы две базовые операции *closure* (I) и *goto* (I, X).

Базовые операции

В ситуации $[S \rightarrow x.]$, определяющей состояние I, точка стоит в конце правой части правила. Это означает, что вершина магазина, на которой сформирована правая часть правила $S \rightarrow x$, готова к свертке. В таком состоянии анализатор выполняет свертку.

Для построения множества состояний определим базовые операции *closure* (I) и *goto* (I, X), где I – множество ситуаций, X – символ грамматики (терминал или нетерминал). Операция *closure* добавляет ситуации к множеству ситуаций, у которых точка стоит слева от нетерминала. Добавляются те ситуации, которые получаются из правил, в левой части которого находится этот нетерминал.

```
closure (I)
{
  do {
    for (каждой ситуации [A->w.Xv] из I) {
      for (каждого правила грамматики X->u) {
        I+= [X->.u];      /* Операция += добавляет элемент к множеству */
      }
    }
  } while (I изменилось);
  return I;
}
```

Операция *goto* "переносит" точку после символа X. Это означает переход из одного состояния в другое под воздействием символа X.

```
goto (I, X)
{
  J={};      /* {} обозначает пустое множество */
  for (каждой ситуации [A->w.Xv] из I) {
    J+= [A->wX.v];
  }
  return closure (J);
}
```

Алгоритм построения конечного автомата

- Пополняем грамматику
- Строим множество состояний T и множество переходов E

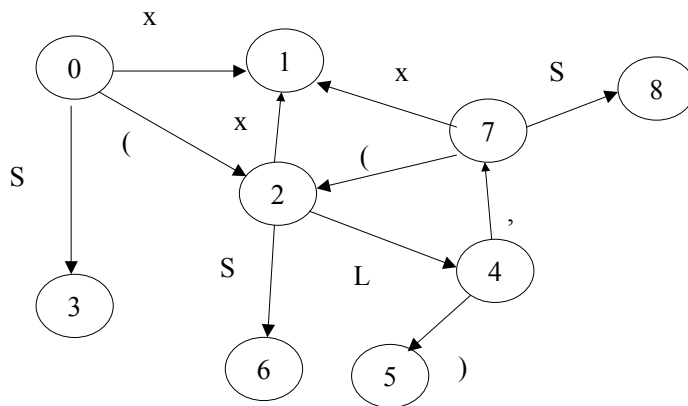
Алгоритм построения конечного автомата

Теперь обсудим алгоритм построения анализатора. Во-первых, пополним грамматику. Обозначим T множество состояний, E – множество переходов.

```
T = {closure ([S' -> .S])};
E = {};
do
{
    for (каждого состояния I из T)
    {
        for (каждой ситуации [A->w.Xv] из I)
        {
            J = goto (I, X);
            T+={J};      /* ко множеству состояний добавляется новое состояние */
            E+=(I->J);   /* ко множеству ребер добавляется ребро, идущее из
                           состояния I в состояние J. Этот переход
                           осуществляется по символу X */
        }
    }
} while (E или T изменились);
```

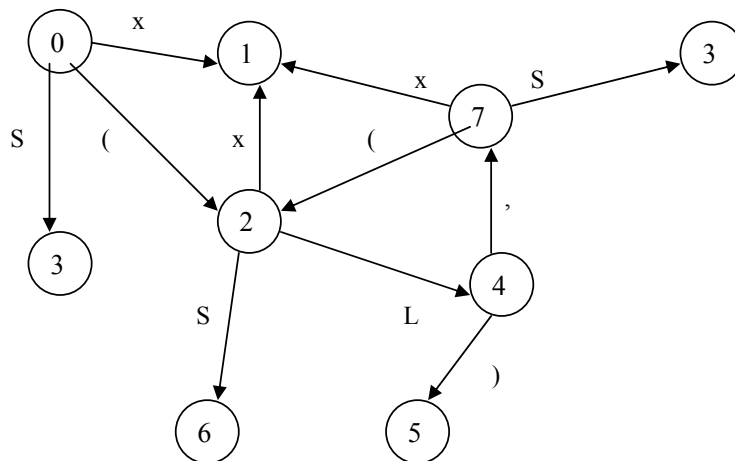
Поскольку для символа $\$$ операция $\text{goto}(I, \$)$ не определена, мы выполняем действие ассерт.

Автомат для грамматики



Автомат для грамматики

Для определенной нами грамматики автомат получится следующим:



где состояния определяются следующим образом:

- 0: $\{[S'.S], [S \rightarrow .x], [S \rightarrow .(L)]\}$
- 1: $\{[S \rightarrow x.]\}$
- 2: $\{[S \rightarrow (L), [L \rightarrow .L,S], [L \rightarrow .S], [S \rightarrow .(L)], [S \rightarrow .x]\}$
- 3: $\{[S' \rightarrow S.]\}$
- 4: $\{[S \rightarrow (L.), [L \rightarrow L., S]\}$
- 5: $\{[S \rightarrow (L).]\}$
- 6: $\{[L \rightarrow S.]\}$
- 7: $\{[L \rightarrow L., S], [S \rightarrow .(L)], [S \rightarrow .x]\}$
- 8: $\{[L \rightarrow L,S.]\}$

Управляющая таблица

	()	x	,	\$	S	L
0	s2		s1			3	
1	r2	r2	r2	r2	r2		
2	s2		s1			6	4
3					acc		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			8	
8	r4	r4	r4	r4	r4		

Управляющая таблица

Теперь мы можем вычислить множество сверток R:

```
R = empty set;
for (each state I in T)
{
    for (each item [A->w.] in I)
    {
        R+= { (I, A->w) };
    }
}
```

Таким образом, алгоритм построения управляющей таблицы автомата состоит из следующих шагов:

- Пополнение грамматики
- Построение множества состояний
- Построение множества переходов
- Построение множества сверток

Для того, чтобы построить таблицу анализатора для грамматики, поступим следующим образом:

1. для каждого ребра $I \xrightarrow{X} J$ мы поместим в позицию $[I, X]$ таблицы
 - shift J , если X – терминал,
 - goto J , если X – нетерминал.
2. для каждого состояния I , содержащего ситуацию $[S' \rightarrow S.]$ мы поместим ассепт в позицию $[I, \$]$
3. для состояния, содержащего ситуацию $[A \rightarrow w.]$ (правило номер n с точкой в конце правила), поместим reduce n в позицию $[I, Y]$ для каждого терминала Y .
4. пустая ячейка означает ошибочную ситуацию

Приведем управляющую таблицу для грамматики G_0 :

	()	x	,	\$	S	L
0	s2		s2			3	
1	r2	r2	r2	r2	r2		
2	s2		s1			6	4
3					acc		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s3		s2			8	
8	r4	r4	r4	r4	r4		

LR(1)-анализатор

- LR(1)-анализатор мощнее, чем LR(0)-анализатор
- Для построения управляющей таблицы такого анализатора мы должны модифицировать базовые процедуры closure и goto

LR(1)-анализатор

LR(1)-анализатор использует для принятия решения один символ входной цепочки. Алгоритм построения управляющей таблицы LR(1)-анализатора подобен уже рассмотренному алгоритму для LR (0)-анализатора, но понятие ситуации в LR(1)-анализаторе более сложное: LR(1)-ситуация состоит из правила грамматики, позиции правой части (представляемой точкой) и одного символа входной строки (*lookahead symbol*). LR(1)-ситуация выглядит следующим образом: $[A \rightarrow w_1 \cdot w_2, a]$, где a – терминальный символ. Ситуация $[A \rightarrow w_1 \cdot w_2, a]$ означает, что цепочка w_1 находится на вершине магазина, и префикс входной цепочки выводим из цепочки w_2 x . Как и прежде, состояние автомата определяется множеством ситуаций. Для построения управляющей таблицы необходимо переопределить базовые операции closure и goto.

```
closure (I) {
  do {
    Iold=I;
    for (каждой ситуации  $[A \rightarrow w_1 \cdot Xw_2, z]$  из I)
      for (любого правила  $X \rightarrow u$ )
        for (любой цепочки  $w$ , принадлежащей FIRST ( $w_2 z$ ))
          I+= $[X \rightarrow \cdot u, w]$ ;
    } while (I != Iold);
  }
  return I;
}

goto (I, X) {
  J = {};
  for (каждой ситуации  $[A \rightarrow w_1 \cdot Xw_2, z]$  из I)
    J+= $[A \rightarrow w_1 X \cdot w_2, z]$ ;
  return closure (J);
}
```

Естественно, операция reduce также зависит от символа входной цепочки:

```
R=[]
foreach (I из T)
  foreach ( $[A \rightarrow w \cdot, z]$  из I)
    R+= $\{ (I, z, A \rightarrow w) \}$ 
```

Тройка $(I, z, A \rightarrow w)$ означает, что в состоянии I для символа z входной цепочки анализатор будет осуществлять свертку по правилу $A \rightarrow w$.

Управляющая таблица LR(1)-анализатора

Построим управляющую таблицу
анализатора для следующей грамматики:

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Управляющая таблица LR(1)-анализатора

Пример. Рассмотрим грамматику:

(1) $E \rightarrow T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

Управляющая таблица для такой грамматики выглядит следующим образом:

Состо- яние	action						goto			
	<i>id</i>	+	*	()	\$	E	T	F	
0	<i>s5</i>			<i>s4</i>				1	2	3
1		<i>s6</i>				accept				
2		<i>r2</i>	<i>s7</i>		<i>r2</i>	<i>r2</i>				
3		<i>r4</i>	<i>r4</i>		<i>r4</i>	<i>r4</i>				
4	<i>S5</i>			<i>s4</i>				8	2	3
5		<i>r6</i>	<i>r6</i>		<i>r6</i>	<i>r6</i>				
6	<i>s5</i>			<i>s4</i>					9	3
7	<i>s5</i>			<i>s4</i>						10
8		<i>s6</i>			<i>s11</i>					
9		<i>r1</i>	<i>s7</i>		<i>r1</i>	<i>r1</i>				
10		<i>r3</i>	<i>r3</i>		<i>r3</i>	<i>r3</i>				
11		<i>r5</i>	<i>r5</i>		<i>r5</i>	<i>r5</i>				

Как обычно,

si – перенос и переход в состояние i

ri – свертка по правилу i

i – переход в состояние i

LALR(1)-анализатор

- Таблицы LR(1)-анализатора могут быть очень большими
- Таблицы LALR(1)-анализатора получаются из таблиц LR(1)-анализатора слиянием "эквивалентных" состояний в одно

LALR(1)-анализатор

Таблицы LR(1)-анализатора могут оказаться очень большими, ведь даже маленькая грамматика нашего примера привела к автомату с двенадцатью состояниями. Таблицы меньшего размера можно получить путем слияния любых двух состояний, которые совпадают с точностью до символов входной строки (lookahead symbols).

Пример. Рассмотрим грамматику G_1 с правилами:

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

Пополним эту грамматику правилом $S' \rightarrow S$.

Для этой грамматики мы получим следующие состояния:

- 0: $\{[S' \rightarrow S, \$], [S \rightarrow AA, \$], [A \rightarrow aA, a], [A \rightarrow aA, b], [A \rightarrow b, a], [A \rightarrow b, b]\}$
1: $\{[S' \rightarrow S, \$]\}$
2: $\{[S' \rightarrow A.A, \$], [A \rightarrow aA, \$], [A \rightarrow b, \$]\}$
3: $\{[A \rightarrow a.A, a], [A \rightarrow a.A, b], [A \rightarrow a.A, a], [A \rightarrow a.A, b], [A \rightarrow b, a], [A \rightarrow b, b]\}$
4: $\{[A \rightarrow b., a], [A \rightarrow b., b]\}$
5: $\{[S \rightarrow AA, \$]\}$
6: $\{[A \rightarrow a.A, \$], [A \rightarrow aA, \$], [A \rightarrow b, \$]\}$
7: $\{[A \rightarrow b., \$]\}$
8: $\{[A \rightarrow aA., a], [A \rightarrow aA., b]\}$
9: $\{[A \rightarrow aA., \$]\}$

Граф переходов выглядит так:

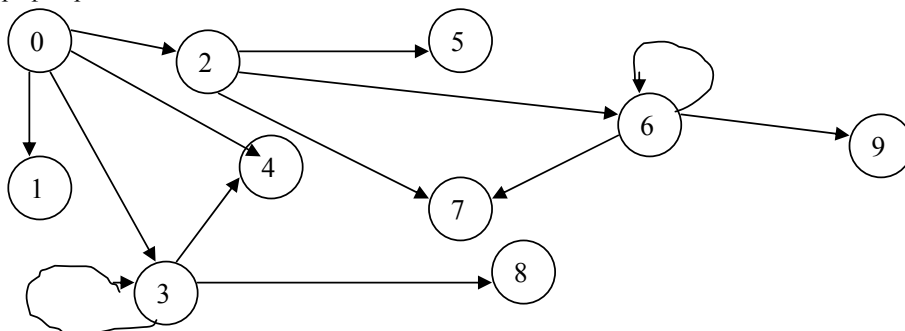


Таблица LALR-анализатора для грамматики G_1

	a	b	\$	S	A
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Таблица LALR-анализатора для грамматики G_1

Теперь можно построить таблицу LR-анализатора:

State	action			goto	
	a	b	\$	S	A
0	s3	s4		1	2
1	accept				
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5	r1				
6	s6	s7			9
8	r2	r2			
9	r2				

Нетрудно заметить, что пары состояний 3 и 6, 4 и 7, 8 и 9 различаются только вторыми компонентами, определяющих их ситуаций. Поэтому мы можем «склеить» эти пары. В результате получится таблица LALR-анализатора:

State	action			goto	
	a	b	\$	S	A
0	s36	s47		1	2
1	accept				
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5	r1				
89	r2	r2	r2		

Пример LR(1)-грамматики

- $S \rightarrow aAd$
- $S \rightarrow bBd$
- $S \rightarrow aBe$
- $S \rightarrow bAe$
- $A \rightarrow c$
- $B \rightarrow c$

Пример LR(1)-грамматики

Таким образом, LALR анализатор имеет значительное преимущество в размере таблиц по сравнению с LR. Однако, существуют грамматики, которые можно обработать LR анализатором, но нельзя LALR анализатором. LALR анализатор будет считать их неоднозначными, а LR анализатор не будет.

Пример. Грамматика

$S \rightarrow aAd$

$S \rightarrow bBd$

$S \rightarrow aBe$

$S \rightarrow bAe$

$A \rightarrow c$

$B \rightarrow c$

не является LALR грамматикой, поскольку для входной цепочки ac мы можем выполнить свертку либо по правилу $A \rightarrow c$ (если текущий входной символ d) либо по правилу $B \rightarrow c$ (если текущий входной символ e).

Однако на практике большинство таких неоднозначностей можно устранить. Перейдем к рассмотрению различных возможных неоднозначностей и методов их устранения.

Неоднозначные грамматики. Конфликты «перенос-свертка»

- $S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{other}$

Подобные конфликты могут быть решены путем трансформации грамматики с следующего виду:

- $S \rightarrow M \mid U$
- $M \rightarrow \text{if } E \text{ then } M \text{ else } M \mid \text{other}$
- $U \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } M \text{ else } U$

Неоднозначные грамматики. Конфликты «перенос-свертка»

Вопрос неоднозначности становится особенно важным в процессе построения управляющей таблицы анализатора LR(k)-языка, так как неоднозначность грамматики приводит к конфликтам при построении таблицы.

Рассмотрим сначала конфликты типа перенос-свертка (*shift/reduce*). Конфликты данного типа возникают, когда в процессе работы анализатора возникает выбор между переносом текущего символа на вершину стека и сверткой последовательности, расположенной на вершине стека.

Пример. Пусть дана грамматика G_1 , имеющая следующий набор правил:

- (1) $stmt \rightarrow \text{if } expr \text{ then } stmt$
- (2) $stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt$
- (3) $stmt \rightarrow other$

, где *other* мы используем для обозначения других операторов.

На следующем слайде рассмотрим, что происходит при анализе следующей входной цепочки:

$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2.$

Пример конфликта «перенос-свертка»

- Рассмотрим следующую входную последовательность:
if E₁ then if E₂ then S₁ else S₂
- Во время анализа возникнет неоднозначность; обычно ее разрешают путем введения дополнительных соглашений о принадлежности ветки *else*
- Грамматику можно преобразовать к эквивалентной форме без конфликта "перенос/свертка"

Пример конфликта перенос-свертка

Итак, имеется следующая входная цепочка: *if E₁ then if E₂ then S₁ else S₂*. Рассмотрим работу анализатора пошагово:

Содержимое стека	Необработанная часть входной цепочки	Действие
\$	<i>if E₁ then if E₂ then S₁ else S₂</i>	<i>shift</i>
\$ <i>if</i>	<i>E₁ then if E₂ then S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁</i>	<i>then if E₂ then S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁ then</i>	<i>if E₂ then S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁ then if</i>	<i>E₂ then S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁ then if E₂</i>	<i>then S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁ then if E₂ then</i>	<i>S₁ else S₂</i>	<i>shift</i>
\$ <i>if E₁ then if E₂ then S₁</i>	<i>else S₂</i>	<i>shift</i>

После последнего шага возникают две альтернативы: либо (а) применить свертку по правилу 1 к последовательности *if E₂ then S₁* на вершине стека, либо (б) перенести символ *else* на вершину стека. Обе альтернативы легко угадываются из вида правил 1 и 2. Грамматики с правилами такого типа называются грамматиками с «висящим» (*dangling*) *else*.

Для подавляющего большинства языков программирования, имеющих условные операторы описанного вида, действие (б) предпочтительно. Общепринятым правилом для данной ситуации является соотнесение каждого *else* с «ближайшим» *then*. Это правило может быть формализовано с использованием однозначной грамматики. Идея в том, чтобы установить соответствие между *then* и *else*, что эквивалентно требованию, чтобы между *then* и *else* могли появиться только оператор, не являющийся условным оператором, или условный оператор с обязательным *else* (*if expr then stmt else stmt*).

Разрешение конфликта перенос-свертка

- Конфликт перенос-свертка может быть решен следующими методами:
 - Вводя новые нетерминалы *matched_statement* и *unmatched_statement*
 - Явным образом разрешая конфликт при его возникновении (предпочитая перенос в случае возникновения конфликта перенос-свертка)

Разрешение конфликта перенос-свертка

Следуя формализации правила явного предпочтения, может быть построена следующая грамматика:

- (1) $stmt \rightarrow matched_stmt$
- (2) $stmt \rightarrow unmatched_stmt$
- (3) $matched_stmt \rightarrow \text{if } expr \text{ then } matched_stmt \text{ else } matched_stmt$
- (4) $matched_stmt \rightarrow Other$
- (5) $unmatched_stmt \rightarrow \text{if } expr \text{ then } stmt$
- (6) $unmatched_stmt \rightarrow \text{if } expr \text{ then } matched_stmt \text{ else } unmatched_stmt$

Новая грамматика порождает тот же язык, что и старая, но вывод цепочки $\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$ теперь не содержит конфликтов.

Альтернативой построению новой грамматики может служить «соглашение», что в случае конфликта перенос-свертка, перенос является предпочтительным действием.

После принятия одной из этих альтернатив вывод может быть продолжен следующим образом:

Stack contents	Unprocessed input string	Action
$\$ \text{ if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else}$	S_2	<i>shift</i>
$\$ \text{ if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$		<i>reduce [2]</i>
$\$ \text{ if } E_1 \text{ then } S$		<i>reduce [1]</i>
$\$$		

Неоднозначные грамматики. Конфликт перенос-перенос

- $S \rightarrow id(L) \mid E = E$
- $L \rightarrow L, P \mid P$
- $P \rightarrow id$
- $E \rightarrow id(I) \mid id$
- $I \rightarrow I, E \mid E$

Неоднозначные грамматики. Конфликт перенос-перенос

Второй тип конфликта, который может возникнуть, это так называемый конфликт перенос-перенос (reduce/reduce), который возникает, когда на вершине стека анализатора возникает строка терминалов, к которой может быть применена свертка по двум различным правилам.

Пример. Рассмотрим грамматику G_2 (*id*, '(', ')', '=' и ',' – терминалы).

- (1) $stmt \rightarrow id(parameters_list)$
- (2) $stmt \rightarrow expr = expr$
- (3) $parameter_list \rightarrow parameter_list, parameter$
- (4) $parameter_list \rightarrow Parameter$
- (5) $parameter \rightarrow Id$
- (6) $expr \rightarrow id(expr_list)$
- (7) $expr \rightarrow Id$
- (8) $expr_list \rightarrow expr_list, expr$
- (9) $expr_list \rightarrow Expr$

В процессе разбора входной цепочки *id(id, id)* происходит следующее:

Содержимое стека	Необработанная часть	Действие
\$	<i>id(id, id)</i>	<i>shift</i>
\$ <i>id</i>	<i>(id, id)</i>	<i>shift</i>
\$ <i>id</i> (<i>id, id)</i>	<i>shift</i>
\$ <i>id</i> (<i>id</i>	<i>, id)</i>	<i>shift</i>

Очевидно, что после выполнения последнего шага необходимо произвести свертку находящегося на вершине стека терминала *id*. Но какое правило использовать? Если использовать правило (5), то будет получен вызов процедуры, если использовать правило (7), то получится вырезка из массива. Чтобы избежать неоднозначности, в первом правиле можно заменить терминал *id* на другой терминал, например, *procid*. Но в этом случае, чтобы вернуть правильный лексический класс, лексический анализатор должен выполнить сложную работу по определению, является ли данный идентификатор обозначением процедуры или массива.

Неоднозначные грамматики. Приоритет операций

- Иногда конфликт перенос-свертка не может быть решён предпочтением переноса, например, рассмотрим грамматику:

$$E \rightarrow id \mid num \mid E * E \mid E + E$$
- В этой грамматике нет приоритетов операций, поэтому сложение и умножение выполняются в порядке появления во входной последовательности. Следующее преобразование решает эту проблему:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id \mid num \mid (E)$$

Неоднозначные грамматики. Приоритет операций

Пример. Рассмотрим грамматику G_3

- (1) $E \rightarrow id$
- (2) $E \rightarrow num$
- (3) $E \rightarrow E * E$
- (4) $E \rightarrow E + E$

Рассмотрим процесс анализа входной цепочки $2 * 3 + 4$.

Содержимое стека	Необработанная часть входной цепочки	Действие
\$	2*3+4	<i>shift</i>
\$2	*3+4	reduce [2]
\$E	*3+4	<i>shift</i>
\$E*	3+4	<i>shift</i>
\$E*3	+4	reduce [2]
\$E*E	+4	<i>shift</i>

После выполнения последнего описанного шага возникают две альтернативы: либо (а) произвести свертку последовательности, находящейся на вершине стека, используя правило 3, либо (б) произвести перенос символа + на вершину стека. Так как приоритет умножения больше приоритета сложения, следует сделать свертку. Однако, это противоречит общему правилу, согласно которому, в случае конфликта перенос-свертка, перенос является предпочтительной операцией (которое было таким удобным способом разрешить конфликт в первом случае).

В данной ситуации существует эквивалентная грамматика G_4 , в которой цепочка $2 * 3 + 4$ имеет единственный вывод:

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow id$
- (6) $F \rightarrow num$

Неоднозначные грамматики. Ассоциативность

- Рассмотрим предыдущую грамматику и входную цепочку $1+2+3$
- Неясно, как посредством грамматики определить, является ли сложение левоассоциативным или нет; для задания ассоциативности аппарата грамматик не достаточно
- Большинство генераторов синтаксических анализаторов позволяют явно задать ассоциативность операторов (`%left`, `%nonassoc...`)

Неоднозначные грамматики. Ассоциативность

Рассмотрим конфигурации, возникающие при анализе строки $1+2+3$.

Содержимое стека	Необработанная часть входной цепочки	Действие
\$	$1+2+3$	<i>Shift</i>
$\$1$	$+2+3$	Reduce [2]
$\$E$	$+2+3$	<i>Shift</i>
$\$E+$	$2+3$	<i>Shift</i>
$\$E+2$	$+3$	

После последнего шага возникает конфликт перенос-свертка. Выбор переноса делает сложение правоассоциативным, выбор свертки - левоассоциативным. Так как левоассоциативное сложение более естественно, свертка предпочтительна. В данном случае не существует эквивалентной однозначной грамматики. Формализма грамматик не хватает для описания данной ситуации и необходимы дополнительные средства.

Таким образом, существуют ряд стандартных ситуаций, в которых возникают неоднозначные грамматики. Большая часть подобных ситуаций может быть решена преобразованием грамматик («висящие» *else*, приоритет операций и т.д.), но не всегда это необходимо («висящие» *else*) и не всегда является лучшим решением. Существуют ситуации, когда подходящее преобразование грамматики не существует, в таких случаях необходимо привлекать дополнительные механизмы, как это сделано, например, для определения ассоциативности операций.

Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман
"Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001, 768 стр.
- D. Grune, C. H. J. Jacobs "Parsing Techniques – A Practical Guide", Ellis Horwood, 1990. 320 pp.

Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001, 768 стр.
- D. Grune, C. H. J. Jacobs "Parsing Techniques – A Practical Guide", Ellis Horwood, 1990, 320 pp.