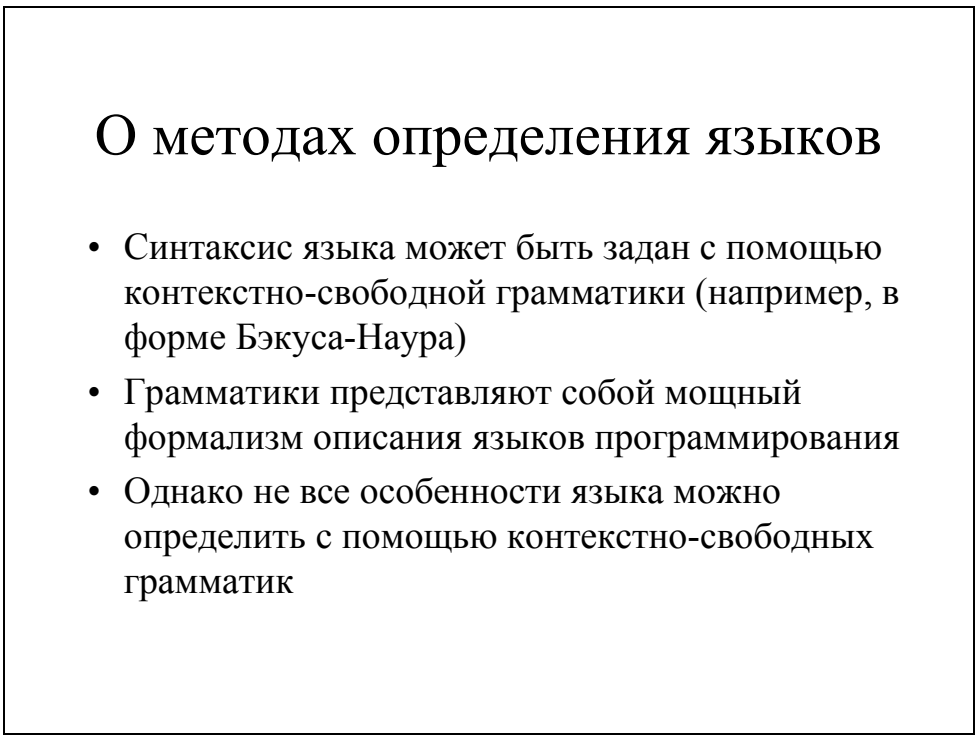
6. Синтаксический анализ

- Синтаксический анализ
- Контекстно-свободные грамматики
- Нисходящие анализаторы
- Метод рекурсивного спуска

Лекция 6. Синтаксические анализаторы. Нисходящие анализаторы

В этой лекции рассматриваются следующие вопросы:

- Синтаксический анализ
- Контекстно-свободные грамматики
- Нисходящие анализаторы
- Метод рекурсивного спуска



О методах определения языков

Каждый язык программирования описывается с помощью набора правил, определяющих структуру правильной программы. Как мы уже обсуждали в предыдущих лекциях, наиболее удобным формализмом для описания синтаксических конструкций языка программирования являются контекстно-свободные грамматики (например, широко распространена нормальная форма Бэкуса-Наура).

Грамматики одинаково помогают решать задачи как программистов, использующих язык, так и создателей компиляторов для данного языка:

- Грамматика предоставляет точную и достаточно легкую для понимания синтаксическую спецификацию языка программирования.
- Для некоторых классов грамматик мы можем автоматически сконструировать эффективный анализатор, который определяет, является ли исходная программа синтаксически правильной.
- Аккуратно созданная грамматика может придать языку программирования такую структуру, которая будет полезна и при трансляции исходной программы в правильный объектный код, и при определении ошибок.
- Компиляторы, разработанные на базе грамматик, могут быть достаточно легко расширены (это особенно полезно для добавления новых конструкций, появившихся как результат развития языка)

Еще раз подчеркнем, что с помощью контекстно-свободных грамматик определяется только так называемая контекстно-свободная составляющая языка программирования, то есть только то, каким образом записывается та или иная конструкция языка. Другая важная часть определения синтаксической правильности программы – правильность использования типов в программе – не может быть определена с помощью контекстно-свободных грамматик. Поэтому если программа выводима в грамматике, это еще не означает, что она полностью синтаксически правильна.

Синтаксический анализ

- Вход синтаксического анализатора – последовательность лексических классов, которая является выходом лексического анализатора
- Выход синтаксического анализатора – дерево разбора, которое является входом для следующего просмотра компилятора

Синтаксический анализ

Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой. В принципе, по любой грамматике можно построить синтаксический анализатор, но грамматики, используемые на практике, имеют специальную форму. Например, известно, что для любой контекстно-свободной грамматики может быть построен анализатор, сложность которого не превышает $O(n^3)$ для входной строки длины n, но в большинстве случаев по заданному языку программирования мы можем построить такую грамматику, которая позволит сконструировать и более быстрый анализатор. Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Вход синтаксического анализатора – последовательность лексических и таблицы, например, таблица внешних представлений, которые являются выходом лексического анализатора.

Выход синтаксического анализатора – дерево разбора и таблицы, например, таблица идентификаторов и таблица типов, которые являются входом для следующего просмотра компилятора (например, это может быть просмотр, осуществляющий контроль типов).

Отметим, что совсем необязательно, чтобы фазы лексичекого и синтаксического анализа выделялись в отдельные просмотры. Обычно эти фазы взаимодействуют друг с другом на одном просмотре. Основной фазой такого просмотра считается фаза синтаксического анализа, при этом синтаксический анализатор обращается к лексическому анализатору каждый раз, когда у него появляется потребность в очередном терминальном символе.

Классы синтаксических анализаторов

- Большинство методов анализа принадлежит к одному из двух классов:
  - *Нисходящие анализаторы*, которым соответствуют LL-грамматики
  - *Восходящие анализаторы*, которым соответствуют LR-грамматики

Классы синтаксических анализаторов

Большинство известных методов анализа принадлежат одному из двух классов, один из которых объединяет *нисходящие (top-down)* алгоритмы, а другой – *восходящие (bottom-up)* алгоритмы. Происхождение этих терминов связано с тем, каким образом строятся узлы синтаксического дерева: либо от корня (аксиомы грамматики) к листьям (терминальным символам), либо от листьев к корню.

Нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. С нисходящими анализаторами связаны так называемые LL-грамматики, которые обладают следующими свойствами:

- Они могут быть проанализированы без возвратов
- Первая буква L означает, что мы просматриваем входную цепочку слева направо *(left-to-right scan)*
- Вторая буква L означает, что строится левый вывод цепочки *(leftmost derivation)*.

Популярность нисходящих анализаторов связана с тем, эффективный нисходящий анализатор достаточно легко может быть построен вручную, например, методом рекурсивного спуска. Кроме того, LL-грамматики легко обобщаются: грамматики, не являющиеся LL-грамматиками, обычно могут быть проанализированы методом рекурсивного спуска с возвратами.

С другой стороны, восходящие анализаторы могут анализировать большее количество грамматик, чем нисходящие, и поэтому именно для таких методов существуют программы, которые умеют автоматически строить анализаторы. С восходящими анализаторами связаны LR-грамматики. В этом обозначении буква L по-прежнему означает, что входная цепочка просматривается слева направо (*left-to-right scan*), а буква R означает, что строится правый вывод цепочки *(rightmost derivation)*. С помощью LR-грамматик можно определить большинство использующихся в настоящее время языков программирования.

Метод рекурсивного спуска

- Для объяснения принципов, лежащих в основе метода рекурсивного спуска, рассмотрим задачу вычисления значения арифметической формулы.

Метод рекурсивного спуска

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является *метод рекурсивного спуска (recursive descent method)*.

Для объяснения принципов, лежащих в основе метода рекурсивного спуска, рассмотрим задачу вычисления значения арифметической формулы. Будем рассматривать формулы, состоящие из целочисленных значений, бинарных операций сложения (+), вычитания (–), умножения (*) и деления нацело (/), а также круглых скобок. Как обычно, приоритеты операций умножения и деления равны и их приоритет больше, чем приоритеты операций сложения и вычитания, причем приоритеты этих операций также равны. Будем называть операции + и – операциями типа сложения, а операции * и / – операциями типа умножения. Круглые скобки используются для изменения стандартного порядка выполнения операций. Наша задача заключается в написании программы, вычисляющей значение формулы.

Изучаемые нами формулы можно представить следующим образом:

$T_1+T_2+\ldots+T_n$,

где T_i – это формула вида $F_{1i}*F_{2i}*\ldots*F_{mi}$. В свою очередь, F_{ji} – это либо число, либо произвольная формула, заключенная в круглые скобки.

Представим себе процесс вычисления значения формулы. Вначале вычисляется F_{11}, далее мы выясняем, какая операция следует за F_{11}. Если это операция типа умножения, то мы, зная ее левый операнд, вычисляем правый операнд и выполняем операцию. Тем самым, мы получим левый операнд для возможных следующих операций типа умножения. Когда мы закончим вычисление формулы $F_1*F_2*\ldots*F_n$, то, возможно, увидим далее операцию типа сложения, и процесс вычисления такой формулы будет аналогичен только что описанному процессу.

Вычисление значения формулы

- Простейшие формулы – числа и произвольные формулы, заключенные в круглые скобки
- Формулы, содержащие операции типа умножения, т.е. умножение и деление.
- Формулы, содержащие операции типа сложения, т.е. сложение и вычитание

Вычисление значения формулы

Итак, мы можем разделить все формулы на следующие классы:

- Простейшие формулы: числа и произвольные формулы, заключенные в круглые скобки, например, 354, (17+3–18)
- Формулы, содержащие операции типа умножения, т.е. умножение и деление, например, 18*2*(35+2)*7
- Формулы, содержащие операции типа сложения, т.е. сложение и вычитание, например, 18*2*(35+2)*7+354+ (17+3-18)*(12–7).Теперь мы можем представить себе процесс вычисления значения формулы, как следующий вызов:

Expression (Term (Factor ())),

где *Factor* – процедура вычисления простейшей формулы, являющейся либо числом, либо произвольной формулой, заключенной в круглые скобки, *Term* – процедура вычисления значения формулы, содержащей операции типа умножения, *Expression* – процедура вычисления значения формулы, содержащей операции типа сложения.

Опишем процедуру, вычисляющую значение простейшей формулы:

```
int Factor ()
{
   char ch = getChar();
   if (isDigit (ch)) return getValue(ch);
   if (ch == '(')
   {
      int result = Formula ();
      if (getChar() == ‘)’) return result;
      error ("Неожиданный символ");
      return 0;
   }
   return error ("Неожиданный символ");
}
```

Формула, содержащая операции типа умножения

- Такие формулы имеют следующий вид:
  *F1*F2*...*Fn*
- Мы сможем понять, что имеем дело с подобной формулой только в момент встречи операции умножения или операции деления.
- Обрабатывать такие формулы будет процедура *Term*, параметром которой является целочисленное значение, представляющее левый операнд формулы F_1*F_2, т.е. F_1.

Формула, содержащая операции типа умножения

Общий вид формулы, содержащей операции типа умножения:

$$F_1*F_2*...*F_n$$

Мы сможем понять, что имеем дело с подобной формулой только в момент встречи операции умножения или операции деления. Обрабатывать такие формулы будет процедура *Term*, параметром которой является целочисленное значение, представляющее левый операнд формулы F_1*F_2, т.е. F_1. Для того, чтобы вычислить значение такой формулы, мы должны выбрать ее правый операнд, который может быть простейшей формулой, а затем выполнить операцию. Это приводит нас к следующему фрагменту кода:

```
int Term (int left)
{
   char ch = getChar(); int right;
   if (ch != '*' && ch !='/')
   {
    /* как оказалось, очередной символ не является обозначением ожидаемой нами
       операции, поэтому мы должны вернуть ненужный нам операнд*/
      returnChar(); /* возвращаем неиспользованную литеру */
      return left;  /* и неиспользованное значение */
   }
/* теперь все в порядке и нам необходимо вычислить значение правого операнда */
   right = Factor();
   if (ch == '*')
   {
      return Term(left * right);
      /* Этот вызов позволит нам вычислить остальную часть формулы */
   }
   if (right == 0) return error ("Деление на нуль");
   return Term(left / right);
}
```

Формула, содержащая операции типа сложения

- Такие формулы имеют следующий вид:
  $T1+T2+...+Tn$
- Мы сможем понять, что имеем дело с такой формулой только тогда, когда встретим операцию сложения или операцию вычитания.
- Такие формулы будет обрабатывать процедура *Expression*, параметром которой является целочисленное значение, представляющее левый операнд операции типа сложения

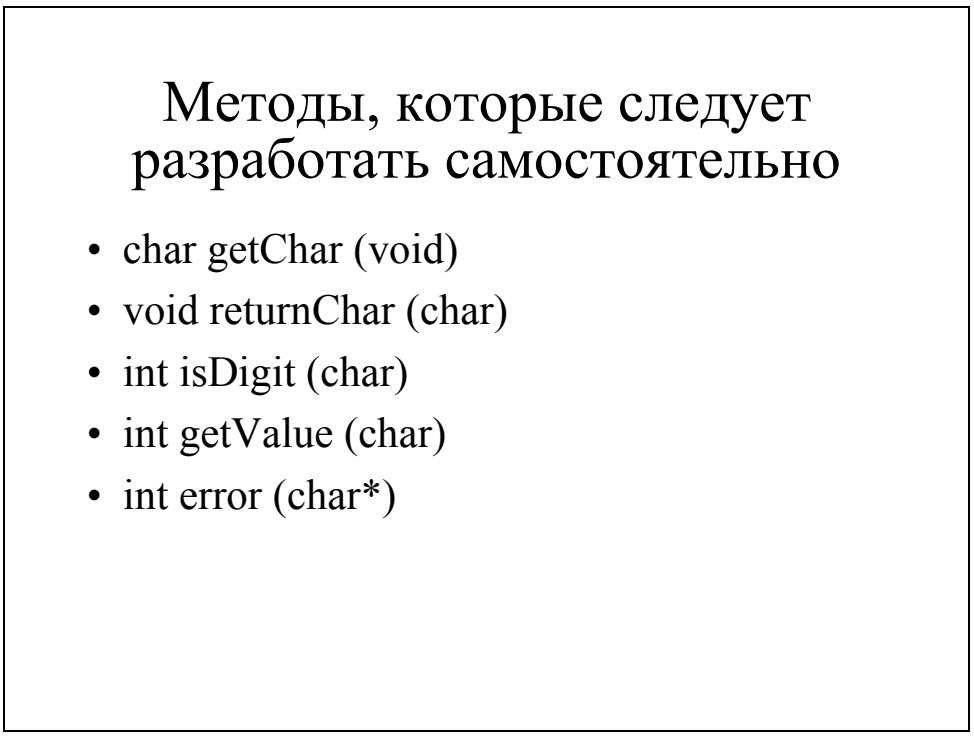
Формула, содержащая операции типа умножения

Общий вид формулы, содержащей операции типа сложения:

$$T1+T2+...+Tn$$

Мы сможем понять, что имеем дело с такой формулой только тогда, когда встретим операцию сложения или операцию вычитания. Такие формулы будет обрабатывать процедура *Expression*, параметром которой является целочисленное значение, представляющее левый операнд операции типа сложения.

```
int Expression (int left)
{
   char ch = getChar (); int right;
   if (ch != '+' && ch != '-')
   {
    /* как оказалось, очередной символ не является обозначением ожидаемой нами
       операции, поэтому мы должны вернуть ненужный нам операнд*/
      returnChar(); /* возвращаем неиспользованную литеру */
      return left;  /* и неиспользованное значение */
   }
/* теперь все в порядке и нам необходимо вычислить значение правого операнда */
   right = Term (Factor());
   if (ch == '+')
   {
      return Expression(left + right);
      /* Этот вызов позволит нам вычислить остальную часть формулы */
   }
   return Expression(left - right);
}
```



Методы, которые следует разработать самостоятельно

В данный момент нас интересует сам метод рекурсивного спуска, а не способы организации ввода данных, поэтому в нашем примере мы опирались на следующие предположения:

- Существует метод без параметров getChar, выдающий очередную литеру из входного потока
- Метод returnChar возвращает неиспользованную литеру обратно во входной поток
- Кроме того, мы не будем описывать методы isDigit и getValue. Первый из этих методов возвращает true только в том случае, когда ее параметр является цифрой. Второй метод извлекает из входного потока все цифры, которые непосредственно следуют за литерой, передаваемой ей в качестве параметра, и вычисляет соответствующее целочисленное значение.
- Метод error используется для вывода сообщений об ошибках.

Эти методы необходимо реализовать самостоятельно. При реализации некоторых из этих методов на платформе .NET можно воспользоваться методами стандартных классов.

Заметим, что при реализации этого примера мы использовали метод рекурсивного спуска с возвратами *(recursive descent with backtracking)*. Чуть позже мы увидим, почему нам не удалось ограничиться методом рекурсивного спуска без возвратов.

Условия использования метода рекурсивного спуска

- Метод рекурсивного спуска без возвратов можно использовать только для LL(1)-грамматик

Условия использования метода рекурсивного спуска

Метод рекурсивного спуска без возвратов можно использовать только для грамматик, правила которых удовлетворяют следующему условию: первого символа каждого правила должно быть достаточно для того, чтобы определить, какое правило применимо в данном случае. Более точно это условие можно формализовать путем определения множества FIRST.

Определение. Для КС-грамматики G и цепочки *w*, состоящей из терминальных и нетерминальных символов, определим множество $FIRST_k(w)$ следующим образом:

$FIRST_k(w) = \{x \mid w =>^* xv, |x| = k$ или $w =>^* x, |x| < k\}$, где k – натуральное число.

Иными словами, множество $FIRST_k(w)$ состоит из всех терминальных префиксов длины k терминальных цепочек, выводимых из *w*.

Пример. Рассмотрим грамматику, порождающую подмножество типов языка Pascal.

type → simple
type → ^**id**
type → **array** [simple] **of** type
simple → **integer**
simple → **char**
simple → **num .. num**

Для этой грамматики мы имеем:

$FIRST_1$(simple) = {**integer**, **char**, **num**}
$FIRST_1$(^**id**) = {^}
$FIRST_1$(**array** [simple] **of** type) = {**array**}

Понятно, что если цепочка *w* состоит только из терминалов, то $FIRST_k(w)$ – это первые k символов цепочки *w*, если $|w| \geq k$, или это сама цепочка w, если $|w| < k$.

Алгоритм построения множества FIRST

- Добавим в FIRST ($X_1X_2…X_k$) все непустые символы из FIRST (X_1). Затем, если пустая цепочка принадлежит FIRST (X_1), то добавим все непустые символы из FIRST (X_2), и так далее. Наконец, если для всех i FIRST (X_j) содержит пустой символ, то мы добавим е в множество FIRST ($X_1X_2…X_k$).

Алгоритм построения множества FIRST

Прежде всего, определим множество FIRST для всех символов грамматики:

1) если X – терминал, то FIRST (X) = X
2) для правила X→ε добавим ε к множеству FIRST (X)
3) если X – нетерминал и X → Y_1Y_2 … Y_k – правило грамматики, то добавим терминал *a* в FIRST (X), если для некоторого *i* этот терминал *a* принадлежит FIRST (Y_i) и ε принадлежит всем множествам FIRST (Y_1), …, FIRST (Y_{i-1}), то есть Y_1, …, Y_{i-1} =>*ε. Если ε принадлежит FIRST (Y_j) для всех j =1, 2, …, k, то добавим ε в FIRST (Y).

Теперь сформулируем сам алгоритм построения множества FIRST(w).
Вход. КС-грамматика G=(N, T, P, S) и цепочка w терминальных и нетерминальных символов.
Выход. FIRST (w).
Метод. Добавим в FIRST ($X_1X_2…X_k$) все непустые символы из FIRST (X_1). Затем, если ε принадлежит FIRST (X_1), то добавим все непустые символы из FIRST (X_2), и так далее. Наконец, если для всех j FIRST (X_j) содержит пустой символ, то мы добавим ε в множество FIRST ($X_1X_2…X_k$).

Пример. Рассмотрим грамматику с правилами:

S → B A
A → +B A
A → ε
B → D C
*C → * D C*
C → ε
D → (S)
D → a

Для этой грамматики множества *FIRST* определяются следующим образом:
FIRST (D) = {(, a}, FIRST (C) = {, ε }, FIRST (B) = FIRST (D), FIRST (A)={+, ε },*
FIRST (S) = {(, a}

LL(k)-грамматика

- Для любых двух левых выводов:
  (1) S =>* wAv => wuv =>* wx
  (2) S =>* wAv => wu1v => wy
  для которых FIRST (x) = FIRST (y)
  вытекает, что u=u1.
- Метод рекурсивного спуска без возвратов возможен только для LL(1)-грамматик.

LL(k)-грамматика

Определение. Грамматика $G = (V_T, V_N, P, S)$ называется *LL(k)-грамматикой*, если для любых двух левых выводов

$S =>^* wAv => wuv =>^* wx$

$S =>^* wAv => wu_1v =>^* wy$,

для которых $FIRST_k(x) = FIRST_k(y)$ верно, что $u=u_1$.

То есть если для данной цепочки wAv, состоящей из терминальных и нетерминальных символов и k первых символов (если они есть), выводящихся из Av, существует не более одного правила, которое можно применить к A, чтобы получить вывод какой-нибудь терминальной цепочки, начинающейся с w и продолжающейся упомянутыми k терминалами.

Пример. Рассмотрим грамматику с правилами:

$S \rightarrow aAS$

$S \rightarrow b$

$A \rightarrow a$

$A \rightarrow bSA$

и два вывода

(1) $S =>^* wSv => wuv =>^* wx$

(2) $S =>^* wSv => wu_1v =>^* wy$

Пусть цепочки x и y начинаются с a. Это означает, что в выводе участвовало правило $S \rightarrow aAS$. Следовательно, $u = u_1 = aAS$. Пусть цепочки x и y начинаются с b. Это означает, что в выводе участвовало правило $S \rightarrow b$. Следовательно, $u = u_1 = b$.

Для выводов

(3) $S =>^* wAv => wuv =>^* wx$

(4) $S =>^* wAv => wu_1v =>^* wy$

рассуждение аналогично. Таким образом, грамматика обладает свойством LL(1).

Для LL(1)-грамматик может быть построен анализатор методом рекурсивного спуска без возвратов.

Пример

(1) S -> if S then S else S
(2) S -> begin S L
(3) S -> print E
(4) L -> end
(5) L -> ; S L
(6) E -> num = num

Для этой LL (1)-грамматики построим анализатор методом рекурсивного спуска.

Пример

Рассмотрим грамматику:

(1) S → if E then S else S
(2) S → begin S L
(3) S → print E
(4) L → end
(5) L → ; S L
(6) E → num = num

Напишем анализатор языка, порождаемого этой грамматикой, методом рекурсивного спуска. Для этого нам придется описать по одной процедуре для каждого нетерминала грамматики.

```
class SimpleParser {

/* Лексические классы, т.е. терминалы */

const int IF = 1;
const int THEN = 2;
const int ELSE = 3;
const int BEGIN = 4;
const int END = 5;
const int PRINT = 6;
const int SEMICOLON = 7;
const int NUM = 8;
const int EQ = 9;

public static void nextStep(int lc)
{
   if (lexical_class == lc)
      lexical_class = getLC();
   else
      error();
}
```

```
public static void S(void)
{
   switch(getLC())
   {
      case IF:
         E(); nextStep(THEN); S(); nextStep(ELSE); S(); break;
      case BEGIN:
         S(); L(); break;
      case PRINT:
         E(); break;
      default:
         error(); break;
   }
}

public static void L(void)
{
   switch (lexical_class)
   {
      case END:
         getLC(); break;
      case SEMICOLON:
         getLC(); S(); L(); break;
      default:
         error(); break;
   }
}

public static void E(void)
{
   nextStep(NUM); nextStep(EQ); nextStep(NUM);
}

public static void main(void)
{
   lexical_class = getLC();
   S();
}

} // end of SimpleParser
```

Леворекурсивные грамматики

- Грамматика называется леворекурсивной, если среди ее нетерминалов имеется по крайней мере один леворекурсивный нетерминал.
- Нетерминал A называется леворекурсивным, если существует вывод A =>* Aw.
- Леворекурсивные грамматики не обладают свойством LL(k) ни для какого k.

Леворекурсивные грамматики

LL(k)-свойство накладывает сильные ограничения на грамматику. Иногда имеется возможность преобразовать грамматику так, чтобы получившаяся грамматика обладала свойством LL(1). Такое преобразование далеко не всегда удается, но если нам удалось получить LL(1)-грамматику, то для построения анализатора можно использовать метод рекурсивного спуска без возвратов.

Предположим, что мы хотим построить анализатор языка, порождаемого следующей грамматикой (мы уже приводили неформальное рассмотрение этого примера в лекции 4):

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow num \mid (E)$

Заметим, что терминалы множества *FIRST(T)* принадлежат также множеству *FIRST(E+T)*. В силу этого мы не сможем однозначно определить последовательность вызовов процедур, которую мы должны выполнить при анализе входной цепочки. Проблема заключается в том, что нетерминал *E* встречается на первой позиции правой части правила, левая часть которого также *E*. В такой ситуации нетерминал *E* называется непосредственно леворекурсивным.

Определение. Нетерминал *A* КС-грамматики G называется леворекурсивным, если в грамматике существует вывод $A =>* Aw$.

Грамматика, имеющая хотя бы одно леворекурсивное правило, не может быть LL(1)-грамматикой. С другой стороны, известно, что каждый КС-язык определяется хотя бы одной нелеворекурсивной грамматикой.

Алгоритм устранения леворекурсивности

- Вход. G=(N, T, P, S) – КС-грамматика
- Выход. КС-грамматика G'=(N',T,P',S') – эквивалентная данной и не содержащая леворекурсивных правил.
- Метод. Для каждого леворекурсивного нетерминала A добавляем новый нетерминал A' и все правила, содержащие A в левой части изменяем.

Алгоритм устранения леворекурсивности

Опишем алгоритм устранения непосредственной леворекурсивности. Пусть G = (N, T, P, S) – КС-грамматика и правило $A \to Aw_1 \mid Aw_2 \mid \ldots \mid Aw_n \mid v_1 \mid v_2 \mid \ldots \mid v_m$ представляет собой все правила из P, содержащие A в левой части, причем ни одна из цепочек v_i не начинается с нетерминала A. Добавим к множеству N еще один нетерминал A' и заменим правила, содержащие A в левой части, на следующие:

$A \to v_1 \mid v_2 \mid \ldots \mid v_m \mid v_1 A' \mid v_2 A' \mid \ldots \mid v_m A'$

$A' \to w_1 \mid w_2 \mid \ldots \mid w_n \mid w_1 A' \mid w_2 A' \mid \ldots \mid w_n A'$

Можно доказать, что полученная грамматика эквивалентна исходной.

В результате применения этого преобразования к приведенной выше грамматике, описывающей арифметические выражения, мы получим следующую грамматику:

$E \to T \mid TE'$

$E' \to +T \mid +TE'$

$T \to F \mid FT'$

$T' \to *F \mid *FT'$

$F \to (E) \mid num$

Нетрудно показать, что получившаяся грамматика обладает свойством LL(1).

Еще одна подобная проблема связана с тем, что два правила для одного и того же нетерминала начинаются одними и теми же символами. Например,

$S \to if\ E\ then\ S\ else\ S$

$S \to if\ E\ then\ S$

В этом случае мы добавим еще один нетерминал, который будет соответствовать различным окончаниям этих правил. Мы получим следующие правила:

$S \to if\ E\ then\ S\ S'$

$S' \to$

$S' \to else\ S$

Для полученной грамматики может быть реализован метод рекурсивного спуска.

Рекурсивный спуск с возвратами

- Для обхода трудностей, связанных с совпадением множеств FIRST, на практике зачастую используется следующий прием:
  - Перед началом разбора потенциально неоднозначного фрагмента запоминается текущее состояние лексического анализатора
  - Затем запускается разбор первой из возможных конструкций
  - В случае неудачного завершения разбора, мы восстанавливаем состояние лексического анализатора и переходим к следующему варианту разбора и т.д.
  - Если все варианты завершаются неудачно, то мы сообщаем об ошибке
- Такой прием использован в демонстрационном примере к курсу (компилятор Си-бемоль)

Рекурсивный спуск с возвратами

Итак, для того, чтобы иметь возможность применить метод рекурсивного спуска, мы должны преобразовать грамматику к виду, в котором множества FIRST не пересекаются. Это сложный и неприятный процесс. Поэтому на практике зачастую используется следующий прием, называемый *рекурсивным спуском с возвратами*.

Для этого лексический анализатор представляется в виде объекта, у которого помимо традиционных методов scan, next и т.п., есть также копирующий конструктор. Затем во всех ситуациях, где может возникнуть неоднозначность, мы перед началом разбора запоминаем текущее состояние лексического анализатора (т.е. заводим копию лексического анализатора) и пытаемся продолжить разбор текста, считая, что мы имеем дело с первой из возможных в данной ситуации конструкций. Если этот вариант разбора заканчивается неудачей, то мы восстанавливаем состояние лексического анализатора и пытаемся заново разобрать тот же самый фрагмент с помощью следующего варианта грамматики и т.д. Если все варианты разбора заканчиваются неудачно, то мы сообщаем об ошибке.

Понятно, что такой метод разбора потенциально медленнее, чем рекурсивный спуск без возвратов, но взамен нам удается сохранить грамматику в ее оригинальном виде и сэкономить усилия программиста.

Именно таким образом реализован синтаксический разбор в демонстрационном компиляторе C♭. Например, в следующем фрагменте реализован метод, определяющий по ходу разбора, является ли данная конструкция оператором-выражением или описанием. И та, и другая конструкция могут начинаться с последовательности идентификаторов, разделенных точками, и, возможно, содержащими квадратные скобки (см. пример на следующей странице):

```
AST.Stmt stmt_or_decl ()
{
    Lexer saved = new Lexer (lexer);
    try {
       AST.Type t = type_opt ();
       if (lexer.Is (Token.Tag.Ident))
           return decl_tail (saved.Curr.coor, t);
    }
    catch (ParseFailed) {
       lexer = saved;
       AST.Expr expr = this.expr ();
       return new AST.Stmt.Expr (compiler,
          saved.Curr.coor|lexer.req (Token.Tag.Semicolon).coor, expr);
    }
}
```

В блоке try мы пытаемся разобрать конструкцию, предполагая, что это описание. Если это нам не удалось, то анализатор создает исключение, которое затем отлавливается в блоке catch и приводит к повторному разбору той же конструкции как выражения.

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 (полный текст этой книги был доступен в Интернете на момент написания курса, см. http://www.cs.vu.nl/~dick/PTAPG.html)