Теория и технология программирования

Основы программирования на языках C и C++

Лекция 15. Синтаксический анализ, обратная польская запись

Глухих Михаил Игоревич, к.т.н., доц.
mailto: glukhikh@mail.ru

Рассматриваем на примере

- Пользователь вводит с клавиатуры функцию от переменной x, используя четыре арифметических действия, целые константы, символ x и знаки скобок, например:
  - (x+2)*4-7
  - (5/(x-3)+2*x)*(x-5)
- Затем пользователь вводит границы интервала интегрирования, например:
  - 2.8 4.2
  - -1.0 1.5
- Задача – рассчитать интеграл от заданной функции в заданном интервале и вывести на экран

На прошлой лекции были рассмотрены

- Разбиение по этапам задач анализа и распознавания текста
- Этап лексического анализа - выделение в тексте отдельных лексем (слов)
- Были выделены следующие лексемы:
  - переменная (x)
  - число (21)
  - знаки операций +-*/()
  - признак конца строки #

Этап синтаксического анализа

- Синтаксический анализ, или **парсинг (parsing)** – процесс анализа входной последовательности символов с целью разбора грамматической структуры
  - обычно осуществляется в соответствии с заданной **формальной грамматикой**
- Синтаксический анализатор, или **парсер (parser)** – программа или часть программы, выполняющая синтаксический анализ

Формальная грамматика

- Формальная грамматика – способ описания формального языка
- Иначе говоря, способ описания определенных подмножеств из всего множества предложений языка и их частей
- Например:
  - дано предложение (x+2)*7-1
  - x – это переменная
  - 2, 7, 1 – это числа
  - (x+2), 7 – это множители
  - x, 2, (x+2)*7, 1 – это слагаемые
  - x+2, (x+2)*7-1 – это выражения

Терминалы и нетерминалы

- Терминалы – понятия нижнего уровня, которые уже не требуется определять
  - если у нас уже выполнен лексический анализ, в качестве терминалов используются лексемы
  - если лексический и синтаксический анализ объединяются, в качестве терминалов используются символы
- Нетерминалы – понятия, определяемые через терминалы

Контекстно-свободные грамматики

- Определяются рядом правил вида:
  - <нетерминал>::=<понятие1>…<понятиеN>
  - понятия могут быть терминалами или нетерминалами
  - нетерминалы обычно записываются в угловых скобках
  - терминалы записываются в виде символа (в случае лексемы можно также использовать угловые скобки)
  - данная форма записи грамматических правил называется **формой Бэкуса-Наура**
- Например:

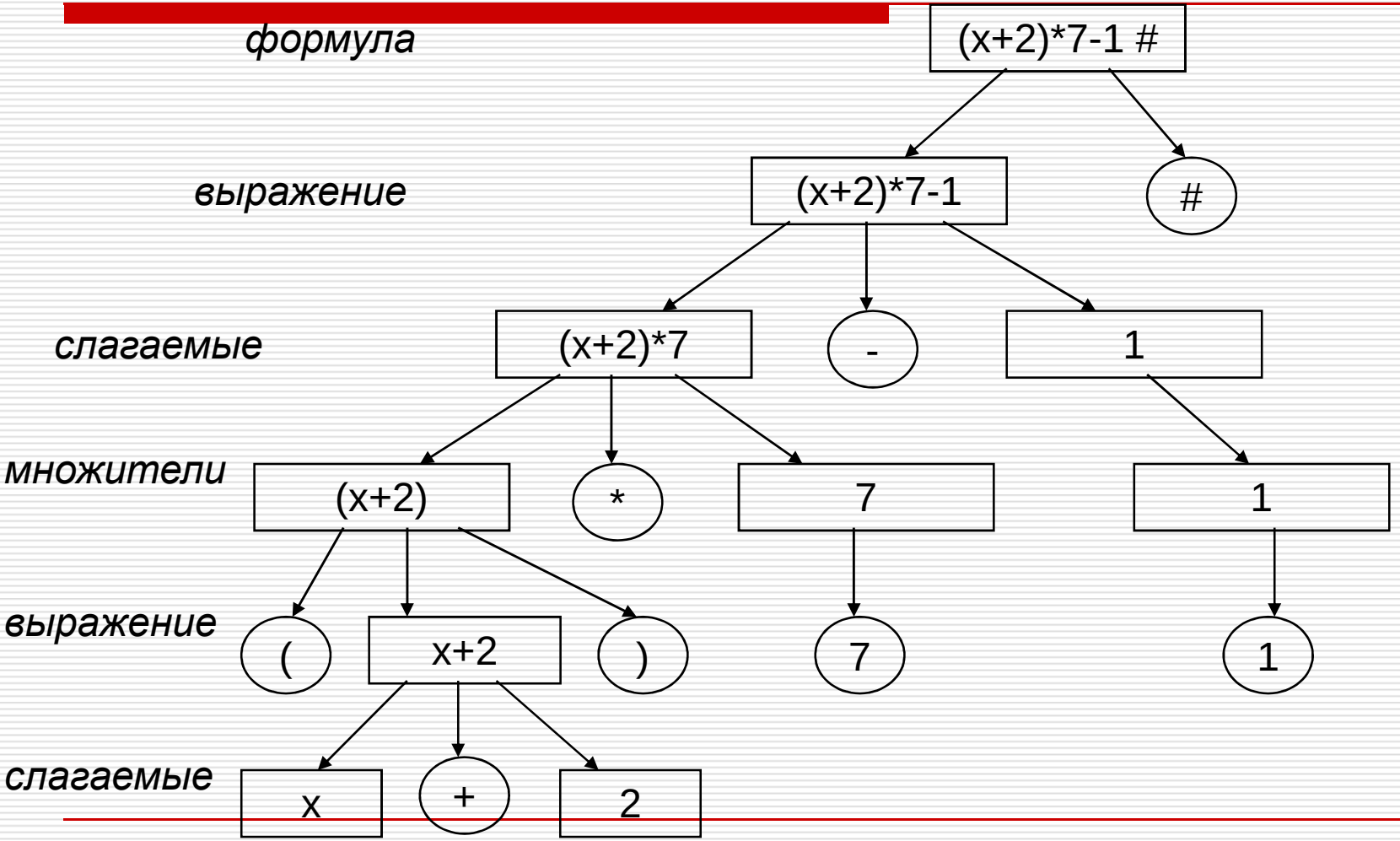
```
<формула>::=<выражение> #
<выражение>::=<слагаемое>
<выражение>::=<слагаемое> + <выражение>
<выражение>::=<слагаемое> - <выражение>
```

- Одно из понятий объявляется корневым, соответствующим всему предложению в целом (например, формула)

Принципы построения грамматики выражений

- Операции сложения и вычитания при отсутствии скобок выполняются на верхнем уровне дерева, поэтому первым определяется понятие **слагаемое**
- На нижних уровнях дерева выполняются операции умножения и вычитания, поэтому следующим определяется понятие **множитель**
- Еще ниже выполняются операции в скобках - в них в свою очередь могут быть **слагаемые** и **множители**, то есть – целые **выражения**
- Вместо бесконечных правил используются **рекурсивные** определения

Грамматическое дерево



Грамматика выражений

```
<формула> ::= <выражение> #
<выражение> ::= <слагаемое>
<выражение> ::= <слагаемое> + <выражение>
<выражение> ::= <слагаемое> - <выражение>
<слагаемое> ::= <множитель>
<слагаемое> ::= <множитель> * <слагаемое>
<слагаемое> ::= <множитель> / <слагаемое>
<множитель> ::= <переменная>
<множитель> ::= <число>
<множитель> ::= ( <выражение> )
```

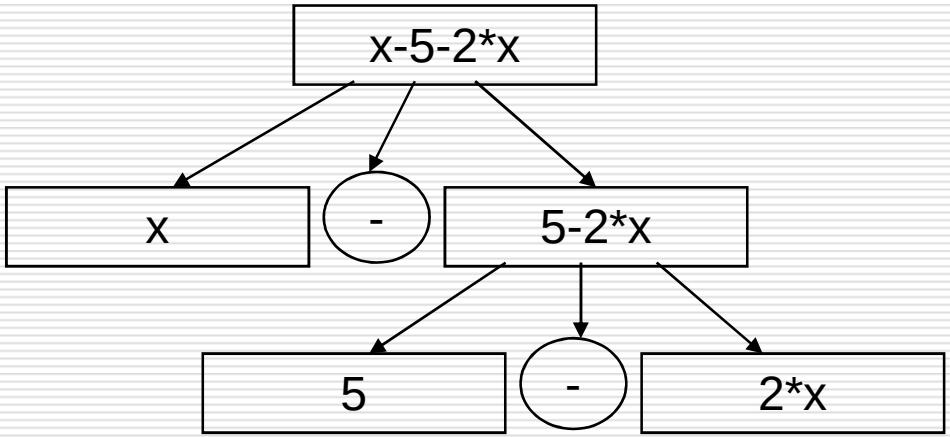
Правая и левая рекурсия

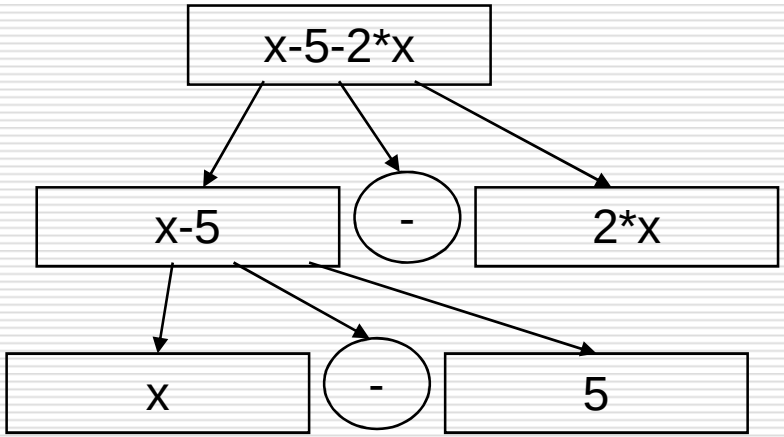
- Правила вида

  `<нетерминал> ::= <понятие> <нетерминал>`
- называются праворекурсивными, так как нетерминал в определении повторяется справа
- Правила вида

  `<нетерминал> ::= <нетерминал> <понятие>`
- называются леворекурсивными, так как нетерминал в определении повторяется слева
- Обычно грамматика формируется из однотипных правил, и соответственно называется (в целом) право- или леворекурсивной. Наша грамматика выражений – праворекурсивная

Отличия правой и левой рекурсии





Алгоритмы синтаксического разбора

- Нисходящий парсер – идем начиная от предложения, пытаясь корректно разбить его на слова, нужна праворекурсивная грамматика
  - LL-парсер
    - рекурсивный парсер
- Восходящий парсер – идем начиная от слов, пытаясь корректно составить из них данное предложение, нужна леворекурсивная грамматика
  - LR-парсер, SLR-парсер
  - LALR-парсер, GLR-парсер

Генераторы парсеров и лексеров

- Грамматика в форме Бэкуса-Наура → разбирающий модуль
  - ANTLR (LL)
  - Bison (LALR, GLR)
  - JavaCC (LL)
  - Yacc (LALR)
- Простой язык → модуль-лексер
  - Lex, Flex

Рекурсивный парсер

- Видимо, самый простой из существующих алгоритмов, достаточно прост для ручного написания
- Каждому из нетерминалов ставим в соответствие функцию, которая его разбирает
- Функция разбора нетерминала должна определить, какое из правил следует применить (если их несколько)
- После чего прочитать из предложения терминалы и/или вызвать функции для разбора нетерминалов
- Такие функции могут вызывать сами себя – рекурсия, или вызывать другие функции, которые в свою очередь могут вызвать их – взаимная рекурсия

Рекурсивный парсер, выбор правил

- Выбор правил, левая рекурсия

  `<выражение>::=<слагаемое>`

  `<выражение>::=<выражение>+<слагаемое>`

- Выбор правил, правая рекурсия

  `<выражение>::=<слагаемое>`

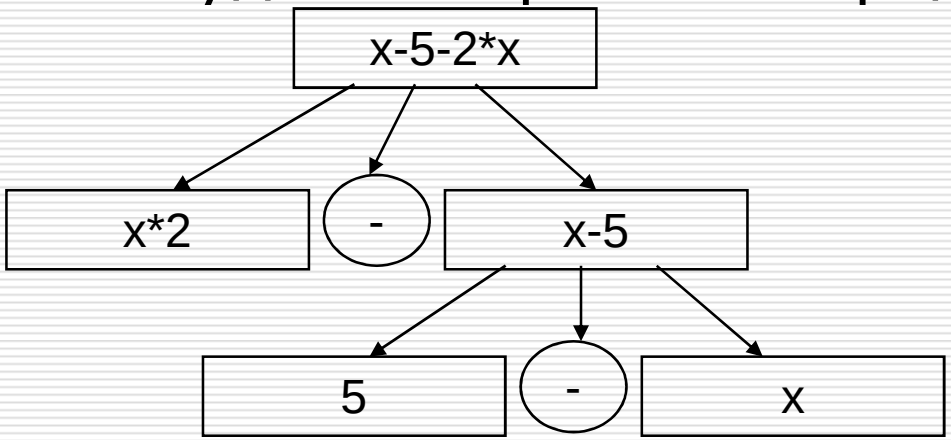
  `<выражение>::=<слагаемое>+<выражение>`

Рекурсивный парсер, выбор правил

- Для левой рекурсии функция **сначала** должна **вызвать сама себя**, а уже потом мы увидим, есть ли там знак сложения - это приводит к бесконечной рекурсии
- Для правой рекурсии мы **сначала читаем слагаемое**, а уже потом, если необходимо, **вызываем сами себя** – это не приводит к бесконечной рекурсии
- Таким образом, рекурсивный парсер неприменим для разбора леворекурсивных грамматик

Что делать в данном случае?

- Вообще-то, леворекурсивная грамматика подходит нам больше – формируется правильный порядок операций
- Мы можем выкрутиться, если применим праворекурсивную грамматику, но читать выражение будем в обратном порядке



Что делать в данном случае?

- Более правильно, однако, применить запись правил в бесконечной форме:

```
<выражение> ::= <слагаемое>
<выражение> ::= <слагаемое> <знак>
   <слагаемое> <знак> ... <знак> <слагаемое>
<знак> ::= +
<знак> ::= -
```

- Не очень честно, зато эффективно и более понятно

Реализация рекурсивного парсера

- Реализуем в рамках класса «парсер»
- Действия парсера
  - разбор формулы parse()
  - разбор выражения parseExpr()
  - разбор слагаемого parseItem()
  - разбор множителя parseFactor()
- Общие данные
  - предложение, которое разбираем (массив лексем)
  - местоположение в данный момент (указатель или индекс)

Определение класса «парсер»

```
class Parser
{
   // указатель на входной массив
   Lexem* inputLexems;
   // указатель на текущую позицию
   Lexem* inputPtr;
   void parseExpr();
   void parseItem();
   void parseFactor();
public:
   Parser(Lexem* input);
   void parse();
};
```

Конструктор

- Принимаем указатель на массив лексем
- Ставим указатель позиции на начало

```
Parser::Parser(Lexem* input)
{
   int size;
   for (size=0; input[size].type!=LT_NONE; size++);
   inputLexems = new Lexem[size+1];
   for (int i=0; i<size; i++)
      inputLexems[i] = input[i];
   inputLexems[size].type = LT_NONE;
   inputPtr = inputLexems;
}
```

Разбор формулы

<формула> ::= <выражение> #

- Так как вариантов нет, разбираем выражение, а затем проверяем, нашли ли признак конца LT_NONE

```
void Parser::parse()
{
   parseExpr(); // разбор выражения
   Lexem lexem;
   lexem = *inputPtr; // чтение очередной лексемы
   if (lexem.type == LT_NONE) return;
   else throw ParserException();
}
```

Разбор выражения

```
<выражение> ::= <слагаемое>
<выражение> ::= <слагаемое> <знак>
   <слагаемое> ... <слагаемое>
```

- Таким образом, вначале мы должны разобрать слагаемое
- Затем должны посмотреть на следующую лексему
- И, если это плюс или минус, то еще раз разобрать слагаемое
- А если нет, то закончить этот разбор

Разбор выражения

```
void Parser::parseExpr()
{
   parseItem();
   Lexem lexem;
   lexem = *inputPtr;
   while (lexem.type==LT_OPERATION &&
          (lexem.operation==OT_PLUS ||
           lexem.operation==OT_MINUS))
   {
      inputPtr++;
      parseItem();
      lexem = *inputPtr;
   }
}
```

Разбор слагаемого

```
<слагаемое> ::= <множитель>
<слагаемое> ::= <множитель> <знак_мн>
   <множитель> ... <множитель>
```

- Таким образом, вначале мы должны разобрать множитель
- Затем должны посмотреть на следующую лексему
- И, если это умножение или деление, то еще раз разобрать множитель
- А если нет, то закончить этот разбор

Разбор слагаемого

```
void Parser::parseItem()
{
   parseFactor();
   Lexem lexem;
   lexem = *inputPtr;
   while (lexem.type==LT_OPERATION &&
          (lexem.operation==OT_MULT ||
           lexem.operation==OT_DIV))
   {
      inputPtr++;
      parseFactor();
      lexem = *inputPtr;
   }
}
```

Разбор множителя

```
<множитель> ::= <переменная>
<множитель> ::= <число>
<множитель> ::= ( <выражение> )
```

- Действуем в зависимости от первой лексемы
- Если это переменная или число, то разбор множителя на этом заканчивается
- Если же это скобка, то следует разобрать выражение
- И затем искать другую скобку, если ее нет – то это ошибка

Разбор множителя

```
void Parser::parseFactor() {
   Lexem lexem = *inputPtr++;
   switch (lexem.type) {
   case LT_NUMBER:
   case LT_VARIABLE:
      return;
   case LT_OPERATION:
      switch (lexem.operation) {
      case OT_LBRACK:
         parseExpr();
         lexem = *inputPtr++;
         if (lexem.type==LT_OPERATION &&
             lexem.operation==OT_RBRACK) return;
      }
   default: throw ParserException();
   }
}
```

Результат работы данного парсера

- Либо мы успешно разбираем формулу до конца и, значит, формула соответствует данной формальной грамматике
- Либо возникает ParserException и, значит, формула не соответствует данной формальной грамматике

Расчет значения выражения при заданном значении x

- Есть два основных способа
  - Первый способ заключается в том, чтобы запомнить построенное дерево разбора, подставить в него значение переменной и, двигаясь от листьев к вершине, получить результат
  - Как правило, парсеры так и делают. Однако дерево разбора – сложная структура, и, если можно, лучше обойтись без него

Расчет значения выражения при заданном значении x

- Есть два основных способа
  - Второй способ заключается в том, чтобы вместо дерева разбора построить **обратную польскую (постфиксную)** запись выражения

Формы записи выражения

- Обычная (инфиксная) запись
  - знак операции расположен между операндами
  - примеры: (3+x)*4, 3+x*4
  - более привычна нам, однако требует использования скобок для выбора правильного порядка операций
- Обратная польская (постфиксная) запись
  - знак операции расположен после операндов
  - примеры: 3 x + 4 *, 3 x 4 * +
  - в отличие от инфиксной записи, не требует использования скобок

Вычисление выражения в обратной польской записи

- Для вычисления используется стек (LIFO)
- Переменные и константы последовательно заталкиваются в стек
- Если встречаем операцию, достаем из стека два аргумента, выполняем операцию и заталкиваем в стек результат (размер стека при этом уменьшается на 1)
- По окончании вычислений в стеке должен остаться ровно 1 элемент

Вычисление выражения в обратной польской записи

- Пример: 3 x 4 * + при x=2
- эквивалент 3+x*4=3+2*4=11

	← 3
3	← 2
3 2	← 4
3 2 4	→ 2 4 ← 2*4
3 8	→ 3 8 ← 3+8
11	

Как сформировать обратную польскую запись?

- обратная польская запись по сути представляет собой последовательность лексем и является выходом парсера

```
<множитель> ::= <переменная>
<множитель> ::= <число>
```

  - Когда используются эти правила, переменную (или число) необходимо добавить к обратной польской записи

```
<выражение> ::= <слагаемое> + <слагаемое>
<слагаемое> ::= <множитель> * <множитель>
```

  - Когда используются эти правила, в конце их разбора операцию следует добавить к обратной польской записи

Модифицированный класс «парсер»

```
class Parser
{
   Lexem* inputLexems;
   Lexem* inputPtr;
   Lexem* outputLexems; // массив для о.п.з.
   Lexem* outputPtr;    // позиция в о.п.з.
   void parseExpr();
   void parseItem();
   void parseFactor();
public:
   Parser(Lexem* input);
   void parse();
   const Lexem* getResult() const;
};
```

Модифицированный разбор выражения

```
void Parser::parseExpr()
{
   parseItem();
   Lexem lexem;
   lexem = *inputPtr;
   while (lexem.type==LT_OPERATION &&
          (lexem.operation==OT_PLUS ||
           lexem.operation==OT_MINUS))
   {
      inputPtr++;
      parseItem();
      *outputPtr++ = lexem; // добавление элемента к ОПЗ
      lexem = *inputPtr;
   }
}
```

Реализация вычисления значения ОПЗ

- Используем для этого класс «исполнитель» (Executor)
- Действия: расчет значения calcValue
- Данные:
  - ОПЗ (массив лексем)
  - стек (массив вещественных чисел)

Определение класса «исполнитель»

```
class Executor
{
    const Lexem* inputLexems;
    double* stack;
public:
    Executor(const Lexem* input);
    double calcValue(double x);
};

class ExecutorException {};
```

Конструктор

- Необходимо посчитать длину ОПЗ и создать массив для стека аналогичного размера

```
Executor::Executor(const Lexem* input)
{
    inputLexems = input;
    int size;
    for (size=0; input[size].type != LT_NONE;
          size++);
    stack = new double[size];
}
```

Расчет значения

```
double Executor::calcValue(double x)
{
   int stackSize = 0;
   const Lexem* inputPtr = inputLexems;
   double arg1, arg2;
   while (inputPtr->type != LT_NONE)
   {
      switch (inputPtr->type)
      {
      case LT_NUMBER:
         stack[stackSize++] = inputPtr->number;
         break;
      case LT_VARIABLE:
         stack[stackSize++] = x;
         break;
```

Расчет значения

```
        case LT_OPERATION:
            if (stackSize < 2)
                throw ExecutorException();
            arg1 = stack[stackSize-1];
            arg2 = stack[stackSize-2];
            stack[stackSize-2] = getResult(arg1, arg2,
                inputPtr->operation);
            stackSize--;
            break;
        }
        inputPtr++;
    }
    if (stackSize!=1)
        throw ExecutorException();
    return stack[0];
}
```

Расчет результата операции

```
static double getResult(double arg1, double arg2,
                        OperationType operation)
{
   switch (operation)
   {
   case OT_PLUS:
      return arg1+arg2;
   case OT_MINUS:
      return arg1-arg2;
   case OT_MULT:
      return arg1*arg2;
   case OT_DIV:
      return arg1/arg2;
   }
   throw ExecutorException();
}
```

Продолжение следует…

- Далее интегрирование