Теория и технология программирования

Основы программирования на языках C и C++

Лекция 13. Лексический анализ

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 опредeнённое число, например:
  - 2.8
  - -1.0
- Задача – рассчитать значение функции в заданной точке и вывести на экран

В ходе решения задачи будут рассмотрены

- Теоретические элементы:
  - принципы решения задач лексического и синтаксического анализа
  - принципы разбора и вычисления сложных выражений
- Элементы языка C++:
  - объединения (**union**)
  - указатели на функцию
  - рекурсия

Задачи распознавания и анализа текста

- Группа задач, известная уже десятки лет. Сложные примеры:
  - компиляция программ
  - анализ грамматической корректности текста
- В их решении выделяются следующие этапы:
  - **лексический** анализ – выделение в тексте так называемых **слов**, или **лексем**; дальнейший анализ происходит уже над готовыми лексемами
  - **синтаксический** анализ – построение из лексем конструкций языка – предложений; предложения строятся по определенным грамматическим правилам – правилам **синтаксиса**, записанным формально и образующим **формальную грамматику**
  - **семантический** анализ – определение смысла построенных конструкций

Схема анализа текста

Лексический анализ

- Лексемы (слова) – последовательности символов языка (букв), имеющие самостоятельный смысл в этом языке
- При лексическом анализе отбрасываются символы, не входящие в слова языка, например, пробелы, служащие для разделения отдельных лексем
- В нашем примере есть 4 типа лексем:
  - целые константы
  - символ переменной x
  - знаки арифметических операций + - * /
  - скобки ( )

Лексический анализ

- В нашем случае удобно то, что символы, использующиеся в различных лексемах языка, не пересекаются. Это облегчает задачу лексического анализа
- Способы решения задачи лексического анализа в более сложных случаях см., например:
  - Е.В. Пышкин. Структурное проектирование: основание и развитие методов. (глава 8).

Как реализовать лексический анализ?

- Рассмотрим понятия, фигурирующие в данной задаче
  - на входе лексического анализа фигурирует строка, описывающая функцию - записывается в виде символьного массива
  - в процессе лексического анализа мы должны выявлять **типы лексем** и сами **лексемы**
  - на выходе лексического анализа мы должны получить **последовательность лексем**

Тип лексемы

- В нашем проекте встречаются лексемы 3-х типов: целые числа, знаки переменной, знаки операций
- Тип лексемы может быть задан целым числом – например, 0 для целых чисел, 1 для знаков переменной, или 2 для знаков операций

Реализация типа лексемы в виде перечисления

```
// Определение перечисления
enum LexemType
{
    LT_NONE,        // Тип не указан
    LT_NUMBER,      // Число
    LT_VARIABLE,    // Переменная
    LT_OPERATION    // Операция
};
// Переменная типа «перечисление» может
// равняться одному из четырех
// перечисленных значений
LexemType type=LT_NUMBER;
```

Лексема

- Описание лексемы всегда включает в себя ее тип
- Кроме этого, если лексема соответствует целому числу, то описание включает в себя это число
- А если лексема – это операция, то описание включает в себя тип операции

Вариант реализации лексемы

```
enum OperationType
{
    OT_PLUS,
    OT_MINUS,
    OT_MULT,
    OT_DIV,
    OT_LBRACK,
    OT_RBRACK
};
struct Lexem
{
    LexemType type;
    int number;
    OperationType operation;
};
```

Объединения

- В структуре LexemType поля number и operation никогда не используются совместно
- Чтобы подчеркнуть этот факт при описании структуры, а также в целях экономии памяти, можно объединить их в union – объединение

```
union
{
    int number;
    OperationType operation;
};
```

Использование объединений

- К полям объединений можно обращаться так же, как и к полям основной структуры:

```
struct Lexem
{
   LexemType type;
   union
   {
      int number;
      OperationType operation;
   };
};
Lexem lexem;
lexem.type=LT_NUMBER;
lexem.number=24;
// Или
lexem.type=LT_OPERATION;
lexem.operation=OP_PLUS;
```

Использование объединений

- Следует, однако, понимать, что если мы перезаписали поле number, то мы уже не сможем прочитать записанное ранее поле operation, так как в памяти эти два поля занимают одну и ту же ячейку
- В нашем случае поле type указывает, какой из элементов объединения мы используем, а какой – нет

Возможные действия над лексемами

- В нашей задаче необходимо уметь:
  - прочитать очередную лексему из строки
  - вывести лексему в поток (для удобства отладки)
- Для этой цели объявим функции-члены

```
struct Lexem
{
   LexemType type;
   union
   {
      int number;
      OperationType operation;
   };
   char* read(char* str);
   friend ostream& operator <<(ostream& in,
      const Lexem& lexem);
};
```

Реализация чтения из строки

- Аргумент хранит указатель на тот символ, с которого следует начинать чтение
- Результат функции – указатель на тот символ, с которого нужно будет продолжить чтение следующей лексемы

Реализация чтения лексемы

- Для чтения очередной лексемы необходимо:
  - пропустить возможные начальные пробелы
  - посмотреть первый символ
    - если он из +-*/() - это операция
    - если он равен x - это переменная
    - если он из 0123456789 - это число
  - для числа следует прочитать последующие цифровые символы - пока не встретиться нецифровой символ

Пропуск пробелов

```
static bool isSpace(char ch)
{
    return (ch==' ') || (ch=='\t');
}

static char* missSpaces(char* str)
{
    while (*str!=0 && isSpace(*str))
        str++;
    return str;
}
```

Функция TLexem::read

```
char* Lexem::read(char* str)
{
    char ch;
    str=missSpaces(str);
    if (*str==0)
    {
        type=LT_NONE;
        return str;
    }
    ch=*str++;
```

Функция TLexem::read

```
switch(ch)
{
   case '+':
      type = LT_OPERATION;
      operation = OT_PLUS;
      return str;
   case '-':
      type = LT_OPERATION;
      operation = OT_MINUS;
      return str;
   // аналогичные фрагменты для */()
   // …
    case 'x':
         type = LT_VARIABLE;
         return str;
}
```

Функция TLexem::read

```
    if (ch >= '0' && ch <= '9')
    {
        number = (ch - '0');
        ch=*str;
        while (ch >= '0' && ch <= '9')
        {
            number = 10 * number + (ch - '0');
            ch = *++str;
        }
        type=LT_NUMBER;
        return str;
    }
    throw LexemException();
}
```

Тестирование

```
int main(void) {
   try {
      char str[] = "(x +   2)* 4+  7 ";
      Lexem lexems[20];
      char* ptr=str;
      for (int i=0; i<20; i++) {
         ptr=lexems[i].read(ptr);
         cout<<lexems[i];
         if (lexems[i].type==LT_NONE) break;
      }
      cout<<endl;
   } catch (LexemException&) {
      cout<<endl<<"Lexical analysis error"<<endl;
   }
   return 0;
}
```

Регулярные выражения (RegExp)

- Один из мини-языков, описывающий правила построения последовательностей символов
- Некоторые примеры

```
[1-9][0-9]* натуральное число
[\+\-]?[1-9][0-9]* число со знаком
[\+\-\*\\\(\)] один из знаков операций
[\+\-\*\\\(\)\x([1-9][0-9]*)]
Один из знаков операций, x или число
```

Основные правила языка RegExp

- Регулярное выражение – это строка, которую ищем
- Почти все символы в ней воспринимаются как есть, кроме специальных ()[]{}.^$+*|/\?
- \ экранирует другой специальный символ, например \(
- . значит любой символ
- ^ начало строки, $ конец строки
- [] группировка, один из перечисленных символов
  - [0-9] или [0123456789] – один из цифровых символов
- Квантификаторы (относятся к символу или группе)
  - {n} n раз, {m,n} от m до m раз
  - ? Ноль или один, * ноль и более, + 1 и более
- () приоритет, | или

RegExp в C++

- Используется заголовочный файл <regex>
- Тип std::regex

```
regex re("[1-9][0-9]*");
```

- Короткий пример

```
const char* cstr = "22+33-42*8";
cmatch cm; // match results
regex_match (cstr, cm, re);
cout << "string literal with " << cm.size() <<
    " matches" << endl;
for (int i=0; i<cm.size(); i++)
    cout << "[" << cm[i] << "] ";
```

Продолжение следует…

- Далее:
  - синтаксический анализ
  - вычисление значения выражения
  - интегрирование