Практикум на ЭВМ. Интерпретатор. Бинарные операторы

Баев А.Ж.

Казахстанский филиал МГУ

17 марта 2022

Интерпретатор

1. Арифметические операторы
2. Оператор присваивания
3. **Бинарные операторы**
4. Оператор перехода (goto)
5. Условный оператор
6. Цикл while
7. Массивы
8. Функции
9. Рекурсия (стек для вызова функций)

Интерпретатор (битовые операторы)

Битовые операторы

1. `a & b;`
2. `a | b;`
3. `a ^ b;`
4. `a >> b;`
5. `a << b;`

Интерпретатор (операторы сравнения)

Операторы сравнения

1. `a > b;`
2. `a >= b;`
3. `a < b;`
4. `a <= b;`
5. `a == b;`
6. `a != b;`

Результаты вычислений: 1 — истина, 0 — ложь.

Интерпретатор (логические операторы)

Логические операторы

1. `a and b;`
2. `a or b;`

Приоритет

```
enum OPERATOR {
    LBRACKET, RBRACKET,
    ASSIGN,
    OR,
    AND,
    BITOR,
    XOR,
    BITAND,
    EQ, NEQ,
    LEQ, LT,
    GEQ, GT,
    SHL, SHR,
    PLUS, MINUS,
    MULT, DIV, MOD
};
```

```
int PRIORITY[] = {
    -1, -1,
    0,
    1,
    2,
    3,
    4,
    5,
    6, 6,
    7, 7,
    7, 7,
    8, 8,
    9, 9,
    10, 10, 10
};
```

Текстовое представление

```
enum OPERATOR {
    LBRACKET, RBRACKET,
    ASSIGN,
    OR,
    AND,
    BITOR,
    XOR,
    BITAND,
    EQ, NEQ,
    LEQ, LT,
    GEQ, GT,
    SHL, SHR,
    PLUS, MINUS,
    MULT, DIV, MOD
};
```

```
string OPERTEXT[] = {
    "(", ")",
    ":=",
    "or",
    "and",
    "|",
    "^",
    "&",
    "==", "!=",
    "<=", "<",
    ">=", ">",
    "<<", ">>",
    "+", "-",
    "*", "/", "%"
};
```

Реализация

```
1  int main() {
2      std::string codeline;
3      std::vector<Lexem *> infix;
4      std::vector<Lexem *> postfix;
5      int value;
6
7      while (std::getline(std::cin, codeline)) {
8          infix = parseLexem(codeline);
9          postfix = buildPostfix(infix);
10         value = evaluatePostfix(postfix);
11         std::cout << value << std::endl;
12     }
13     return 0;
14 }
```

Реализация parseLexem

```
 1 std::vector<Lexem *> parseLexem(
 2     const std::string &codeline)
 3 {
 4   std::vector<Lexem *> lexems;
 5   int n = sizeof(OPERTEXT) / sizeof(std::string);
 6   for (int i = 0; i < codeline.size(); i++) {
 7     for (int op = 0; op < n; op++) {
 8       string subcodeline =
 9         codeline.substr(i, OPERTEXT[op].size());
10       if (OPERTEXT[op] == subcodeline) {
11         lexems.push_back(new Operator(op));
12         break;
13       }
14     }
15   }
16 }
```

Рекурсивный спуск

Для построения лексического и синтаксического анализатора можно использовать рекурсивный спуск – прямое описание грамматики языка.

Пример для арифметики

```
<POLIZ> := \{ <COMMAND>; \}
<COMMAND> := <EXPR>
<EXPR> := <NUMBER> <BIN_OPERATOR> <EXPR> | <VARIABLE>
<NUMBER> := <DIGIT> \{ <DIGIT> \}
<VARIABLE> := a-z
<BIN_OPERATOR> := + | - | *
<ASSIGN> := :=
```

```
class PostfixParser {
    int position;
    std::vector<Lexem *> poliz;
    bool get_command();
    bool get_expression();
    bool get_number();
    bool get_binary_operator();
    bool get_assign_operator();
    bool get_variable();
public:
```