ОСНОВЫ ТЕОРИИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Пышкин Евгений Валерьевич
к.т.н., доцент

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Блок 8. Оъектно-ориентированный проект

Постановка задачи

- 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

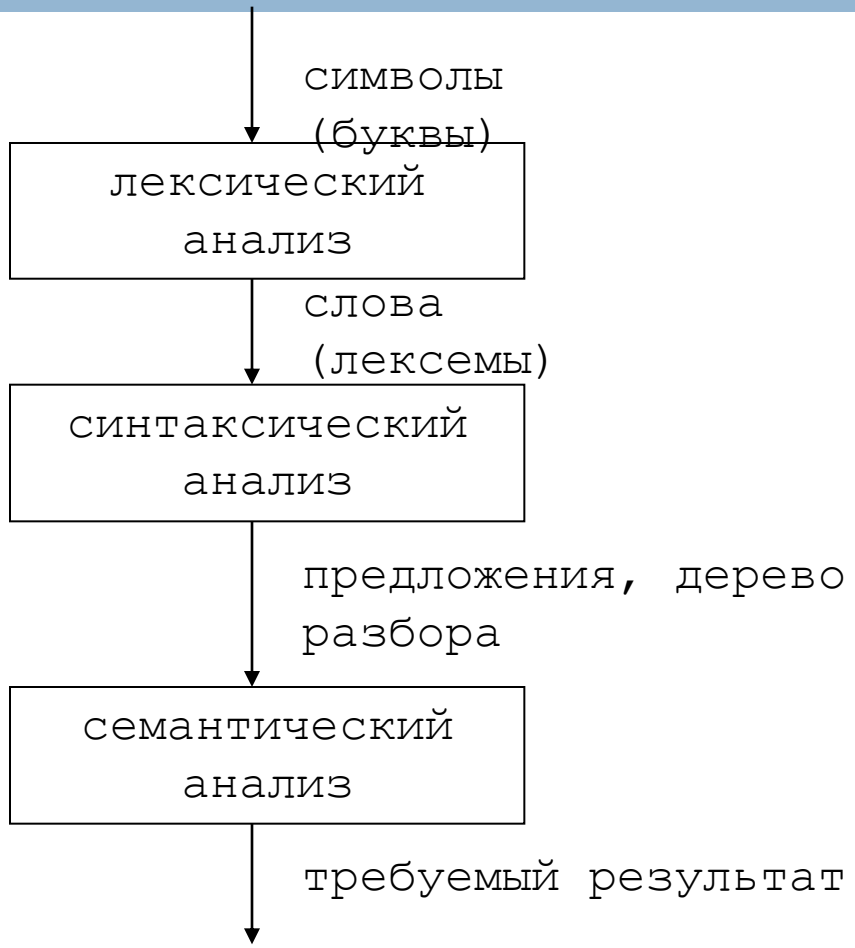
В ходе решения задачи будут рассмотрены

- Теоретические элементы:
  - принципы решения задач лексического и синтаксического анализа
  - принципы вычисления сложных выражений
  - простые методы численного интегрирования
- Элементы языка C++:
  - перечисления (**enum**)
  - 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 в нашем случае?

- 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
- Для определения возможных типов лексем будем использовать **перечисления**

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;
```

Перечисления

- В памяти компьютера хранятся как целые числа
- По умолчанию, первому элементу перечисления соответствует 0, второму 1 и так далее
- Разрешается указать целые значения явно:
  - enum LexemType
  - {
  - LT_NONE=1,
  - LT_NUMBER=2,
  - LT_VARIABLE=4,
  - LT_OPERATION=5
  - };
- Возможно преобразование из перечисления в целое и обратно
  - LexemType type=(LexemType)2;
  - **int** code=(**int**)type;

Преимущества перечислений

- В целую переменную можно записать любое целое значение, в переменную типа «перечисление» - только одно из заданных значений (то есть, устраняется возможная некорректность)
- Кроме этого, записи с перечислениями лучше читаются, сравните:
  - LexemType lexemType=LT_OPERATION;
  - **int** lexemType=3;

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 описание 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;
```

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!=o && 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;
}
```

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

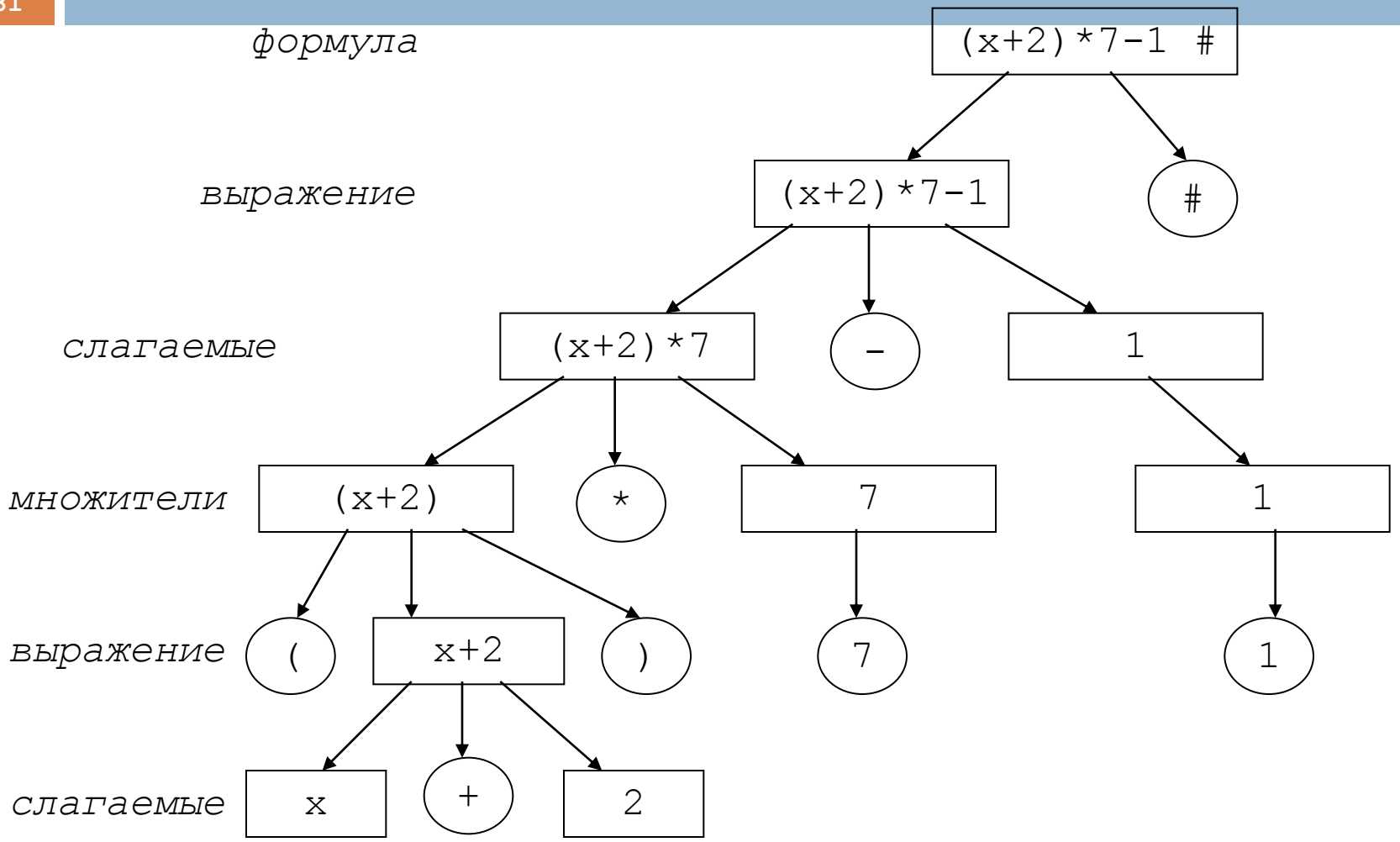
Контекстно-свободные грамматики

- 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



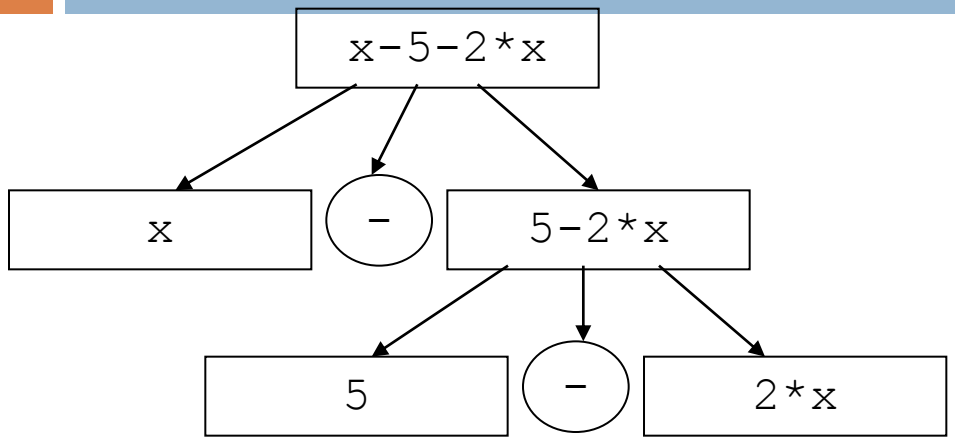
Грамматика выражений

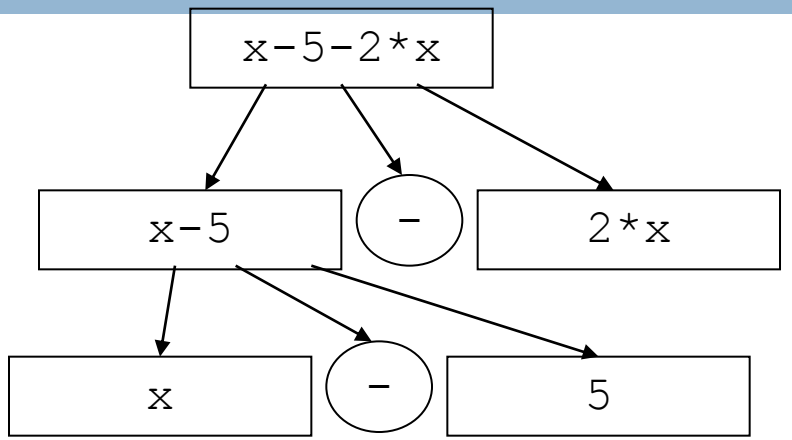
- <формула> ::= <выражение> #
- <выражение> ::= <слагаемое>
- <выражение> ::= <слагаемое> + <выражение>
- <выражение> ::= <слагаемое> - <выражение>
- <слагаемое> ::= <множитель>
- <слагаемое> ::= <множитель> * <слагаемое>
- <слагаемое> ::= <множитель> / <слагаемое>
- <множитель> ::= <переменная>
- <множитель> ::= <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
  - рекурсивный парсер
  - LL-парсер
- Восходящий парсер - идем начиная от слов, пытаясь корректно составить из них данное предложение
  - LR-парсер, SLR-парсер, LALR-парсер

Рекурсивный парсер

- Видимо, самый простой из существующих алгоритмов
- 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

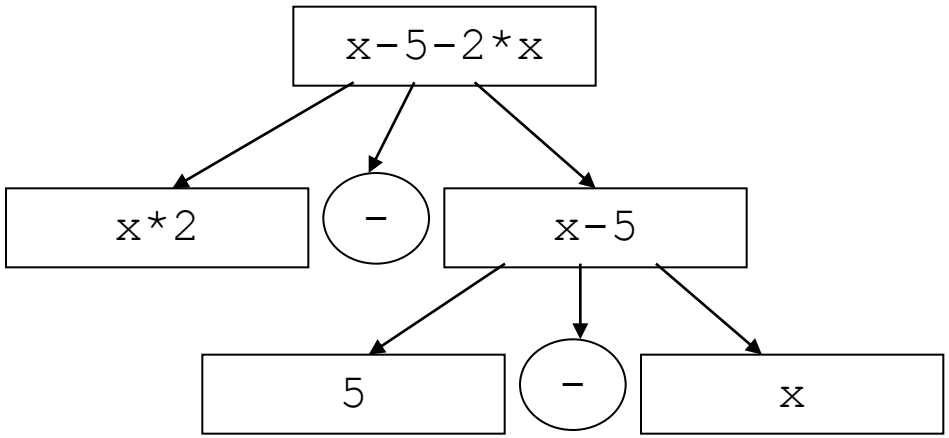
- Выбор правил, левая рекурсия
  - <выражение>::=<слагаемое>
  - <выражение>::=<выражение>+<слагаемое>
- Выбор правил, правая рекурсия
  - <выражение>::=<слагаемое>
  - <выражение>::=<слагаемое>+<выражение>

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



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
- Сразу же переставляем элементы в обратном порядке (оставляя в конце LT_NONE)
- 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[size-1-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> +<выражение>
- <выражение> ::= <слагаемое> - <выражение>
  - Таким образом, вначале мы должны разобрать слагаемое
  - Затем должны посмотреть на следующую лексему
  - И, если это плюс или минус, то еще раз разобрать выражение
  - А если нет, то закончить этот разбор

Разбор выражения


```
void Parser::parseExpr()
{
  parseItem();
  Lexem lexem;
  lexem = *inputPtr;
  switch (lexem.type)
  {
  case LT_OPERATION:
    switch (lexem.operation)
    {
    case OT_PLUS:
    case OT_MINUS:
      inputPtr++;
      parseExpr();
      break;
    }
  }
}
```

Разбор слагаемого

- <слагаемое> ::= <множитель>
- <слагаемое> ::= <множитель> * <слагаемое>
- <слагаемое> ::= <множитель> / <слагаемое>
  - Вначале разбираем множитель
  - Потом проверяем следующую лексему
  - Если это знак умножения или деление, еще раз разбираем слагаемое
  - А если нет, то заканчиваем это разбор

Разбор слагаемого


```
void Parser::parseItem()
{
  parseFactor();
  Lexem lexem;
  lexem = *inputPtr;
  switch (lexem.type)
  {
  case LT_OPERATION:
    switch (lexem.operation)
    {
    case OT_MULT:
    case OT_DIV:
      inputPtr++;
      parseItem();
      break;
    }
  }
}
```

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_RBRACK:
      parseExpr();
      lexem = *inputPtr++;
      if (lexem.type==LT_OPERATION && lexem.operation==OT_LBRACK)
        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

Как посчитать значение разобранного 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

Как посчитать значение разобранного 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

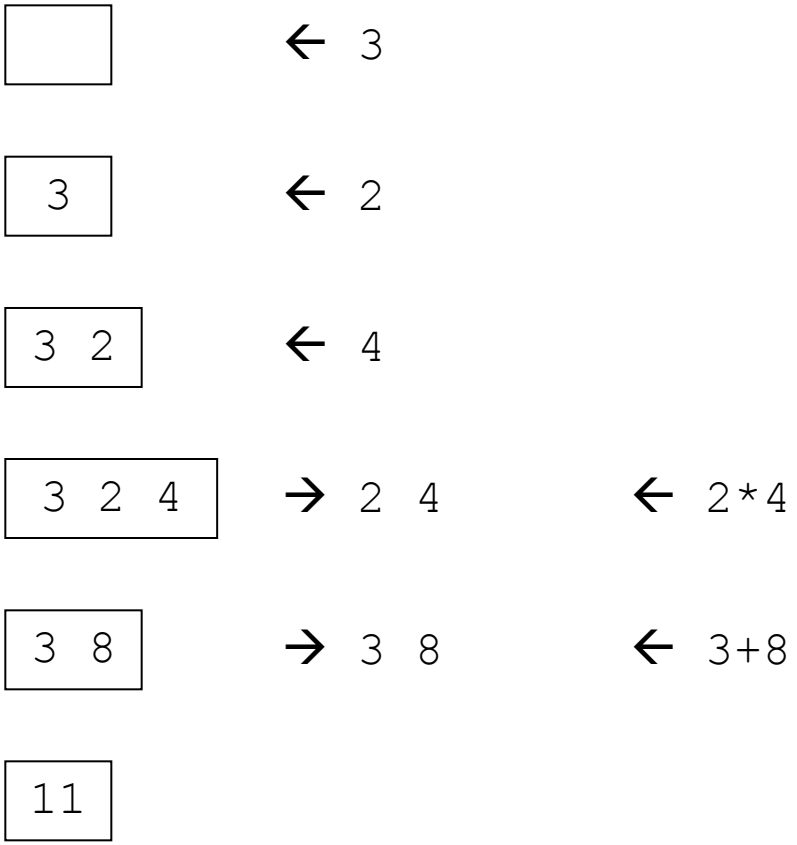
- Обычная (инфиксная) запись
  - знак операции расположен между операндами
  - примеры: (3+x)*4, 3+x*4
  - более привычна нам, однако требует использования скобок для выбора правильного порядка операций
- Обратная польская (постфиксная) запись
  - знак операции расположен после операндов
  - примеры: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



Как сформировать обратную польскую запись при разборе?

- 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> + <выражение>
- <выражение> ::= <слагаемое> - <выражение>
- <слагаемое> ::= <множитель> * <слагаемое>
- <слагаемое> ::= <множитель> / <слагаемое>
  - 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 обратной польской записи
  Lexem* outputPtr;       // позиция в обратной польской записи
  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;
  switch (lexem.type)
  {
  case LT_OPERATION:
    switch (lexem.operation)
    {
    case OT_PLUS:
    case OT_MINUS:
      inputPtr++;
      parseExpr();
      *outputPtr++ = lexem; // добавление лексемы к ОПЗ
      break;
    }
  }
}
```

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];
  }
```

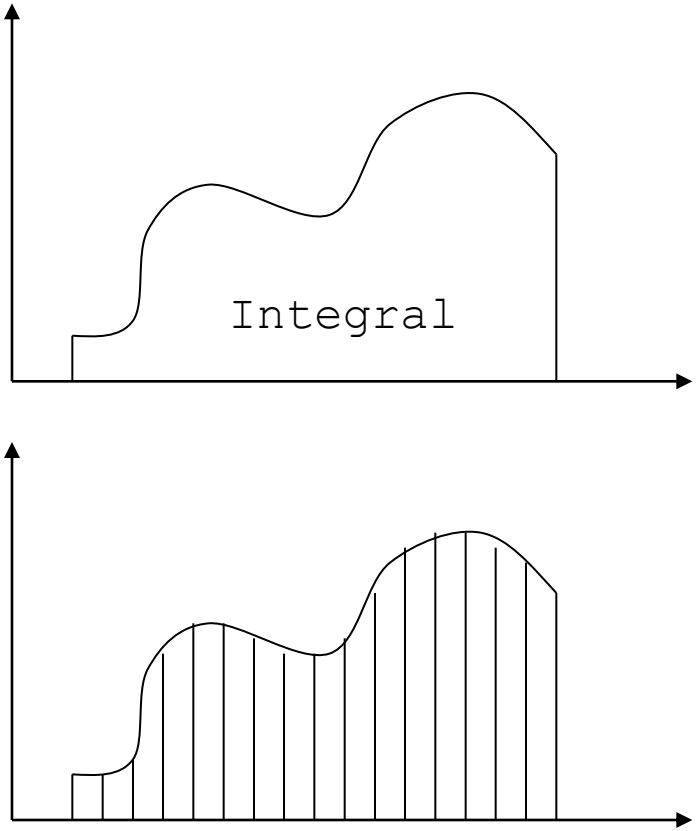
Расчет результата операции


```
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();
}
```

Интегрирование

- 2 основных варианта:
  - Используя известные правила интегрирования элементарных функций, аналитически построить первообразную F(x) и затем вычислить F(b)-F(a)
  - Вычислить интеграл приблизительно (численно), заменив его суммой. Реализуется проще, и обычно не уступает в точности аналитическому расчету

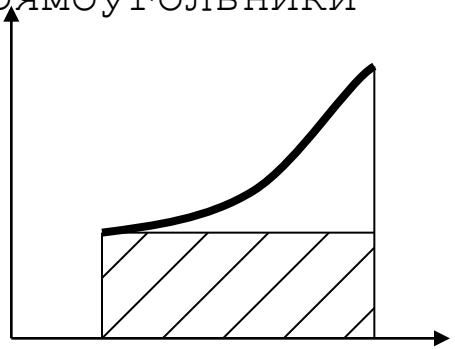
Численное интегрирование



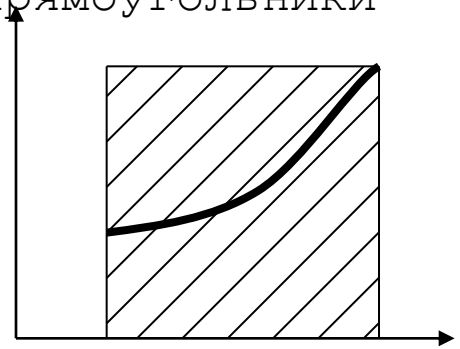
$$Integral = I_1 + I_2 + \ldots + I_{n-1} + I_n$$

Численное интегрирование, варианты

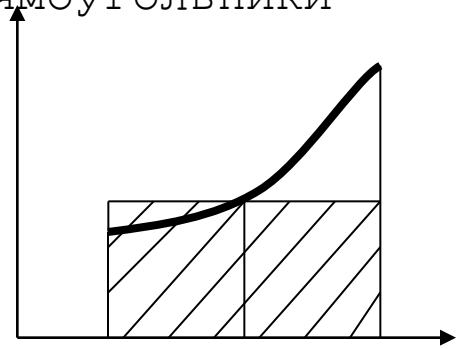
левые прямоугольники



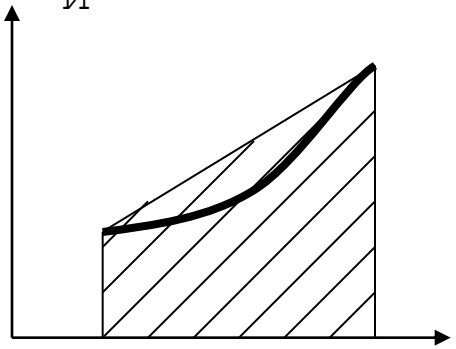
правые прямоугольники



средние прямоугольники



трапеции



Численное интегрирование

- Метод левых прямоугольников:
  I[x,x+d]=d*f(x)
- Метод правых прямоугольников:
  I[x,x+d]=d*f(x+d)
- Метод средних прямоугольников:
  I[x,x+d]=d*f(x+d/2)
- Метод трапеций:
  I[x,x+d]=d*(f(x)+f(x+d))/2
- Метод парабол (метод Симпсона):
  I[x,x+d]=d*(f(x)+4*f(x+d/2)+f(x+d))/6

Качество численного интегрирования

- Основные показатели:
  - как повышается точность с ростом числа отрезков N (СМПС>СП>ТРАП>ЛП=ПП)
  - сколько раз необходимо вычислить функцию f(x) при числе отрезков N (СМПС>ТРАП>ЛП=СП=ПП)
- В совокупности, метод Симпсона при том же числе вычислений функции f(x) в среднем дает значительно более точный результат

Как добиться нужной точности?

- Задаемся необходимой точностью, например, 10^{-8}
- На первом шаге считаем I[a,b]
- На втором шаге считаем I[a,(a+b)/2]+I[(a+b)/2,b]
- На третьем шаге сумму четырех, на четвертом - сумму восьми и так далее
- На каждом шаге считаем $|I_N - I_{N-1}|$. Если вычисленная разница меньше заданной погрешности расчета, останавливаем алгоритм и считаем I_N результатом

Реализация на C++

- Реализуем в рамках объекта Integrator
- Данные: функция f(x), которую интегрируем
- Действия: расчет интеграла calc

Определение класса


```
class Integrator
{
  // Указатель на функцию f(x)
  // Похоже на объявление функции, но f заменяется на (*f)
  double (*f)(double x);
  // Расчет кусочка интеграла
  double calcInterval(double a, double b);
public:
  Integrator(double (*ff)(double x));
  // Расчет всего интеграла с заданной погрешностью
  double calc(double a, double b, double eps);
};

class IntegratorException {};
```

Использование класса


```
#include <math.h>
#include "integrator.h"
int main(void)
{
  // ...
  Integrator integrator(&sin);
  double result=integrator.calc(0, 3.14, 1e-6);
  // ...
  return 0;
}
```

Функции класса


```
Integrator::Integrator(double (*ff)(double x))
{
  f=ff;
}

double Integrator::calcInterval(double a, double b)
{
  // Используем метод Симпсона
  return (b-a)*((*f)(a)+4.0*(*f)((a+b)/2)+(*f)(b))/6.0;
}
```

Функции класса


```
double Integrator::calc(double a, double b, double eps)
{
  if (eps<=0.0 || a>b)
    throw IntegratorException();
  double oldValue, newValue = calcInterval(a, b);
  int intervalNum = 1;
  do
  {
    oldValue = newValue;
    intervalNum *= 2;
    newValue = 0.0;
    double step = (b-a)/intervalNum;
    double start=a;
    for (int i=0; i<intervalNum; i++)
    {
      newValue += calcInterval(start, start+step);
      start += step;
    }
  } while (fabs(newValue-oldValue)>eps);
  return newValue;
}
```

Расчет интеграла по обратной польской записи


```
// Функция calcValue
// не должна быть членом класса
Executor* pExecutor = 0;

double calcValue(double x)
{
  if (pExecutor==0)
    return 0.0;
  return pExecutor->calcValue(x);
}
```

Расчет интеграла по обратной польской записи


```
int main(void)
{
  try
  {
    //…
    Executor executor(output);
    //…
    pExecutor = &executor;
    Integrator integrator(&calcValue);
    cout<<"Integral(0,10)="<<integrator.calc(0,10.0,1e-6)<<endl;
  } catch (ExecutorException&)
  {
    cout<<endl<<"Execution error"<<endl;
  } catch (IntegratorException&)
  {
    cout<<endl<<"Integration error"<<endl;
  }
  return 0;
}
```

Модификация постановки задачи разработки парсера

- Последовательность функций одной переменной вводится из файла.
- В качестве имен переменных могут быть использованы произвольные идентификаторы.
- Некоторые значения переменных могут быть определены явно.
  - На входе программы может быть последовательность формул, так что вычисленное значение одной может использоваться в другой
- Помимо десятичных целочисленных констант могут быть и другие константы

Модификация требований к реализации

- Определить класс «таблица имен» в качестве отдельной сущности
- Выделить в отдельный класс, отвечающий за реализацию лексического анализатора
- Выделить код, ответственный за формирование и обработку польской записи, в отдельный класс.