Современные технологии программирования
Лабораторная работа 11

Алгоритмы компиляторов. Стековый калькулятор. Цель: научиться программировать стековые вычислители.

При создании компиляторов и интерпретаторов для языков программирования исходный текст программы вначале приходится разбирать на отдельные слова (лексемы). Этот первый этап называется лексическим анализом, а программа, которая его делает — лексическим анализатором.

Чтобы провести лексический анализ нужно знать набор допустимых символов языка и набор допустимых слов-лексем. Программа - анализатор пропускает в тексте так называемые пробельные символы, выделяя слова состоящие целиком из значащих символов.

К пробельным символам, как правило, относятся символы следующей строки " \n\r\t". Это пробел, перевод строки, возврат каретки и табуляция (есть правда еще вертикальная табуляция \v но в редакторах текста она, как правило, уже не поддерживается).

Пробельные символы являются разделителями слов языка, но могут быть и другие разделители, такие как скобки, запятая, точка с запятой и так далее. То есть отдельные лексемы, состоящие из одного символа также могут быть разделителями слов.

Например, выражение sin(x+y) содержит 6 лексем: sin,(,x,+,y,) . При этом, sin и + функции, x и y переменные (идентификаторы). Разделителями здесь являются скобки и +.

Лексический анализатор (ЛА), посимвольно читает текст программы, выделяя слова-лексемы. Когда лексема выделена, определяется ее тип (операция, функция, число, строка, скобка, …). ЛА обнаруживает в тексте программе ошибки двух типов: недопустимый символ и недопустимое слово.

Если же все в порядке и ошибок нет, то результатом работы ЛА является массив лексем.

Следующим этапом является синтаксический анализ, который проверяет правильность завершенных языковых конструкций, состоящих из лексем. Во многих языках разделителями таких конструкций в тексте служат точки с запятой и концы строк (символ \n).

Иногда речь ведут также о семантическом анализе, который следует за синтаксическим, и который проверяет правильность языковых конструкций с точки зрения их смыслового содержания. Но этот этап - увы не поддается компьютерной формализации и именно поэтому мы вынуждены изучать кипы документации по тому языку, который используем, где этот смысл разъясняется словами и

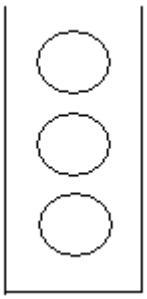
примерами.

Заключительным этапом разбора текста программы является ее исполнение (интерпретация) или же преобразование конструкций языка в исполняемый код и сохранение его в файле.

Как мы покажем на примере, следующем далее, все эти этапы могут быть объединены в одну программу, которая будет разбирать исходный текст и исполнять его, если в нем нет ошибок.

Алгоритм работы стекового калькулятора

Используются два объекта класса **Stack** (scala.collection.mutable.Stack).



Метод **push** помещает некоторый объект в стек сверху, при этом все другие объекты, если они есть, сдвигаются вниз. Метод **pop** выталкивает верхний объект из стека, при этом все другие объекты, если они есть, поднимаются вверх. Метод **top** возвращает самый верхний объект, не вынимая его из стека.

При разборе выражения все операторы, включая скобки (и), в порядке появления помещаются в первый стек (**actionStack**), а все числа - во второй (**resultStack**). Перед помещением нового оператора в стек все операторы в стеке с большим или равным ему приоритетом должны быть выполнены. Рассмотрим работу алгоритма на примере вычисления выражения:

5 * (7 + 8) + 25

1. Начало
actionStack:
resultStack:

2. Определение числа 5
actionStack:
resultStack: 5

3. Определение операции умножения *
actionStack: *
resultStack: 5

4. Определение скобки (
actionStack: *, (
resultStack: 5

5. Определение числа 7
actionStack: *, (
resultStack: 5, 7

6. Определение операции сложения +
actionStack: *, (, +
resultStack: 5, 7

7. Определение числа 8
actionStack: *, (, +
resultStack: 5, 7, 8

8. Определение скобки)
Закрывающая скобка завершает подвыражение, поэтому все операции внутри скобок должны быть выполнены. Таким образом, будет вытолкнута из стека **actionStack** операция сложения + и, поскольку она имеет два аргумента, то из стека **resultStack** будут вытолкнуты два аргумента для нее: 8 и 7. После выполнения сложения, его результат 15 будет снова помещен в стек **resultStack**, а открывающая скобка просто вытолкнута из стека **actionStack**. В результате получим:

actionStack: *
resultStack: 5, 15

9. Определение операции сложения +
Умножение имеет более высокий приоритет, чем сложение, поэтому оно должно быть выполнено до того как + будет помещен в стек. Таким образом, будет вытолкнута из стека **actionStack** операция умножения * и, поскольку она имеет два аргумента, то из стека **resultStack** будут вытолкнуты два аргумента для нее: 15 и 5. После выполнения умножения, его результат 75 будет снова помещен в стек **resultStack**. После этого операция сложения + помещается в стек **actionStack**. В результате получим:

actionStack: +
resultStack: 75

10. Определение числа 25

actionStack: +
resultStack: 75, 25

11. Конец выражения
Выполняется последняя операция сложения +. Она выталкивается из стека и два аргумента для нее 25 и 75, выполняется сложение и его результат помещается в

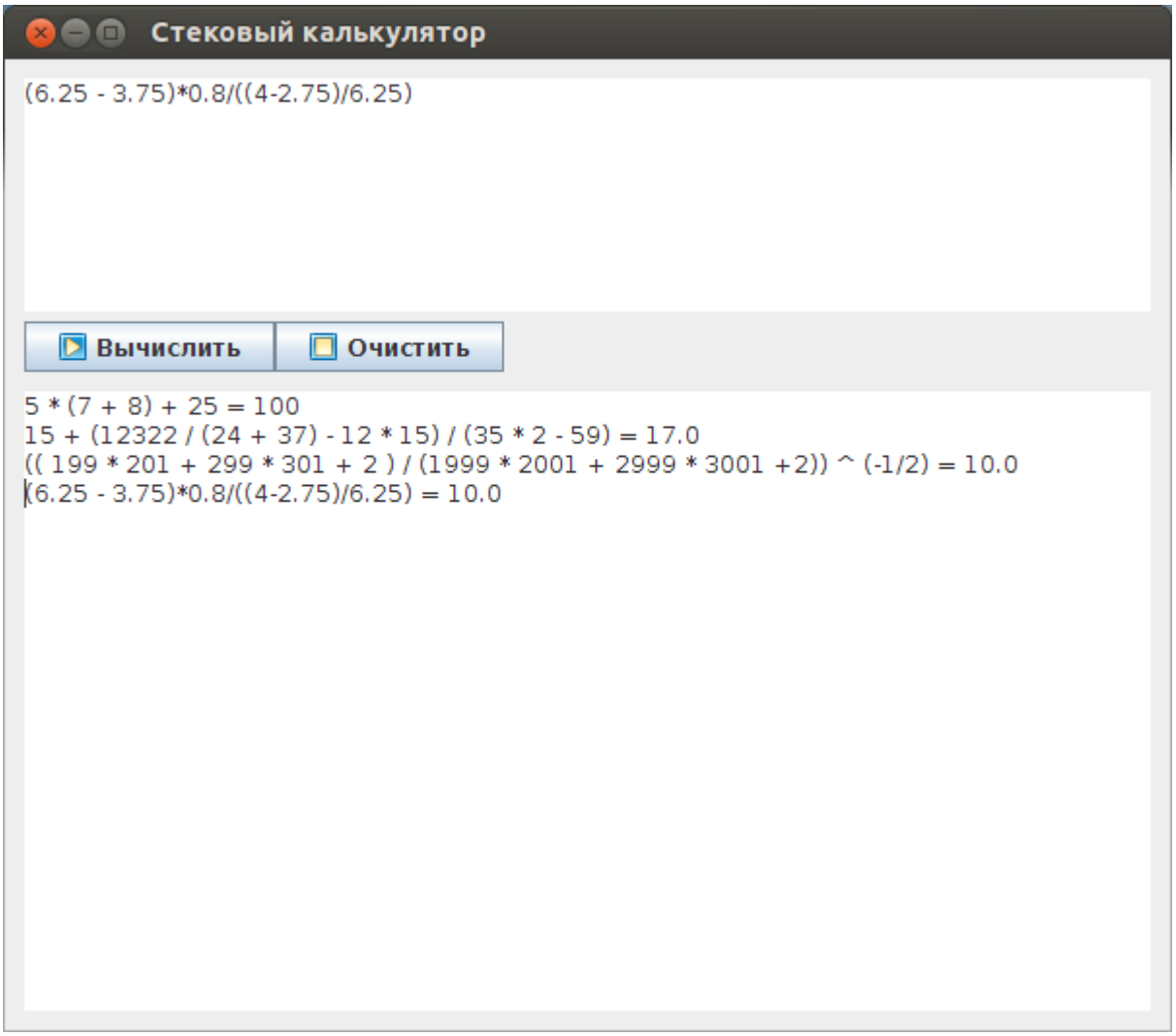
стек **resultStack**.

actionStack:
resultStack: 100

В конце работы алгоритма в стеке **resultStack** остается единственный элемент - результат вычисления. Для упрощения обработки конца выражения, исходное выражение перед вычислением помещается в скобки, тогда обработка последней скобки завершает работу алгоритма.

Варианты заданий лабораторной работы

Общая часть задания для всех вариантов.
Создайте в Intellij IDEA или в NetBeans графическое (GUI) приложение на языке java с использованием компонентов библиотеки Swing. Внешний вид приложения:



Выполнение работы связано с изменением предоставляемого вам кода калькулятора (Calc.scala) в соответствии с Вашим вариантом.

Взяв за основу уже готовый код калькулятора (см. файл Calc.scala в архиве), Вы должны, в соответствии с заданием варианта, добавить в него необходимую функциональность, не испортив при этом то, что он уже умеет.

Calc.scala при помощи плагина assembly для sbt собирайте в библиотеку calc.jar, добавляйте ее к проекту и вызывайте код на Scala из java: calc.Calc.calculate(text); (для удобства пакет в java должен иметь то же имя calc, что и Calc.scala)

Нажатие на кнопку "Вычислить" должно приводить к вычислению выражения, введенного в окне вверху и добавлению результата в нижнее окно как показано на скриншоте (предыдущие вычисления не стираются). В нижнем окне данные не редактируются.

Нажатие на кнопку "Очистить" приводит к удалению информации из нижнего окна.

Возможности калькулятора (Calc.scala):

- идентификация вещественных и целых чисел,
- 5 бинарных операции над ними: +, -, *, /, ^ (сложение, вычитание, умножение, деление, возведение в степень),
- две унарные операции минус и плюс (например, -35 или +21.48),
- круглые скобки в выражениях, проверка парности,
- более высокий приоритет бинарных умножения и деления по отношению к бинарным сложению и вычитанию, и операции возведения в степень перед умножением и делением
- обнаружение неправильных выражений и вывод сообщений об ошибках с указанием строки и позиции в тексте программы,
- допускается размещение выражения в нескольких строках текста, символы пробела, табуляции, перевода строки и возврата каретки игнорируются при разборе.

Варианты заданий.

1. Добавить разбор строк и операцию конкатенации (+) строк. Строки начинаются и заканчиваются двойной кавычкой. Если один из аргументов сложения число, а другой строка, то результатом должна быть соединенная строка.

2. Добавить операцию взятия остатка от деления a%b - остаток a при делении на b, a и b целые. Вещественные аргументы, в случае отсутствия или равенства нулю дробной части, приводить к целым. Приоритет операции как у умножения. Отслеживать и выводить возможные ошибки.

3. Добавить унарную операцию факториал n! - n целое >=0. Вещественный аргумент в случае отсутствия либо равенства нулю дробной части, приводить к целому. Отслеживать и выводить возможные ошибки переполнения.

4. Добавить знак $, который будет указывать на то, что сразу после него идет 16-тиричное целое или вещественное число. В нем допускаются 16-тиричные цифры: A=10, B=11, C=12, D=13, E=14 и F=15. Результат вычислений возвращается как и

ранее в десятичном виде.

5. Добавить вычисление стандарных функций sin(x), cos(x), exp(x), ln(x) и извлечение квадратного корня sqrt(x). Отслеживать и выводить возможные ошибки.

6. Добавить операцию модуль числа |a| - a вещественное или целое число, а также поддержку однострочных // и многострочных /* */ комментариев. Обеспечить проверку парности и вывод ошибок.

7. Добавить операции сравнения выражений: = (равно), # (не равно), < (меньше), > (больше). Возвращаемые значения 1 (истина) и 0 (ложь).

8. Добавить переменные: буквы латинского алфавита (a..z), операцию присваивания = и разделитель ; (точку с запятой). Каждой переменной можно присвоить выражение, содержащее (или нет) другие переменные. Возвращаемое значение - результат вычисления последнего выражения (после последней точки с запятой).

9. Добавить функцию нахождения наибольшего общего делителя gcd(n,m). n и m целые положительные. Вещественные аргументы в случае отсутствия либо равенства нулю дробной части, приводить к целым. Отслеживать и выводить возможные ошибки.

10. Добавить функцию округления чисел round(x), где x - вещественное или целое, функцию отбрасывания остатка floor(x), где x — вещественное или целое и получения случайного вещественного числа random. Эти функции должны работать так, как работают соответствующие функции из math.

11. Добавить функцию нахождения наименьшего общего кратного lcm(n, m). n и m целые положительные. Вещественные аргументы в случае отсутствия либо равенства нулю дробной части, приводить к целым. Отслеживать и выводить возможные ошибки.

Усложненное задание

Переписать код Calc.scala, переориентируя его на абстрактное синтаксическое дерево (AST). То есть в основу положить объект-выражение (expression) и уже у него определять операции +, * и так далее. Пример такого подхода:

http://www.ibm.com/developerworks/ru/library/j-scala08268/index.html

Примеры использования и код калькулятора (Calc.scala).

Для вычисления выражения text: String достаточно передать его аргументом в метод calculate объекта Calc.

```
Calc.calculate(text)
```

Функция isError возвращает true, если произошла ошибка и false, если вычисление выражения прошло успешно. Ошибку в виде строки возвращает функция errorStr.

Примеры использования архива calc.jar (сборка Calc.scala при помощи assembly):

```
$ java -jar calc.jar "5 * (7 + 8) + 25"
100

$ java -jar calc.jar -f  expr.txt
48.0
```

В первом примере выражение для вычисления передается прямо в командной строке, во втором - выражение читается из файла expr.txt (ключ -f означает, что следующий аргумент имя файла)

Код файла **Calc.scala**

```
==========================================================================
package calc

import scala.collection.mutable.Stack
import scala.io.Source

abstract class Token {
  def pos: Int
  def lex: String
}

case class Action(pos: Int, lex: String, pri: Int) extends Token
case class Result(pos: Int, lex: String, rt: Result.ResultType) extends Token

object Result extends Enumeration {
  type ResultType = Value
  val rtString, rtInteger, rtDouble, rtUnknown = Value
}

object Bracket extends Enumeration {
  type Bracket = Value
  val btOpen, btClose, rtDouble, btNone = Value
}
```

```
object Calc {
  val operators="+-*/^"
  val spaces=" \n\r\t"
  val delimiters=operators+spaces+"()"
  var expr = ""
  var ind = 0
  var char = ' '
  var lex = ""
  var error = -1
  val actionStack = new Stack[Action]
  val resultStack = new Stack[Result]
  var opened = 0
  var bracket = Bracket.btNone

  val errorMsg = List (
    "Неизвестная ошибка",           //0
    "Остановлено пользователем",    //1
    "Недопустимый символ",          //2
    "Недопустимое число",           //3
    "Недопустимая строка",          //4
    "Недопустимая функция",         //5
    "Несоответствие скобок",        //6
    "Некорректное выражение",       //7
    "Недопустимый аргумент"         //8
  )

  def isError = error > -1
  def errorStr = if ((error > -1) && (error < errorMsg.length))
    errorMsg(error)+": лексема="+lex+" строка="+line(ind)+" позиция="+position(ind)
    else ""

  def stop(error: Int) = this.error=error

  def line(pos: Int): Int = {
    if ((pos<1) || (pos>expr.length)) 1
    else expr.substring(0,pos-1).count(_=='\n')+1
  }

  def position(pos: Int): Int = {
    val posLineBreak = expr.lastIndexOf('\n',pos-1)
    if (posLineBreak > -1) pos - posLineBreak else pos+1
  }

  def next = {
    if (ind < expr.length) ind+=1
    if (ind == expr.length) char=' '
    else char = expr.charAt(ind)
  }

  def priority(op: Char): Int = op match{
    case '+' | '-' => 1
    case '*' | '/' => 2
    case '^'  => 3
    case _ => 0
  }
```

```
def open = {
  if (bracket==Bracket.btClose) stop(7)
  else {
    bracket==Bracket.btOpen
    opened+=1
    actionStack.push(Action(ind,"(",0))
    next
  }
}

def close = {
  if (bracket==Bracket.btOpen) stop(7)
  else {
    bracket==Bracket.btClose
    opened-=1
    processOperators
    actionStack.pop
    next
  }
}

def nextNumber = {
  lex=""
  bracket==Bracket.btNone
  while((""+char).matches("[\\d\\.]")) {
    lex+=char
    next
  }
  if (delimiters.indexOf(char) > -1) {
    try{
      val rt =
        if (lex.matches("\\d*")) {
          lex.toInt
          Result.rtInteger
        }
        else {
          lex.toDouble
          Result.rtDouble
        }
      resultStack.push(Result(ind-lex.length,lex,rt))
    }
    catch { case _ : Throwable => stop(3)}
  }
  else stop(3)
}

def nextOperator = {
  bracket==Bracket.btNone
  processOperators
  actionStack.push(Action(ind,""+char,priority(char)))
  next
}

def overPriority(a: Action): Boolean = {
  if (isError) false
  else if (a.lex=="(") false
  else if (a.lex==")") true
  else a.pri >= priority(char)
}
```

```
def processOperators = {
  if (actionStack.isEmpty) stop(7)
  else while (overPriority(actionStack.top)) execOperator
}

def execOperator = {
  var unary = false
  if ((actionStack.isEmpty) || (resultStack.isEmpty)) stop(7)
  else {
    val action = actionStack.pop
    val result = resultStack.pop
    if (actionStack.isEmpty) stop(7)
    else {
      if (resultStack.isEmpty) {
        if ((actionStack.top.lex=="(")&&(actionStack.top.pos>result.pos)) stop(7)
        else unary = true
      }
      else if ((actionStack.top.lex=="(")&&(actionStack.top.pos>resultStack.top.pos)) unary=true
    }
    if (!isError) {
      if (unary) {
        if (action.pos<result.pos) action.lex match {
          case "+" => resultStack.push(result)
          case "-" => resultStack.push(minus(result))
          case _ => stop(7)
        } else stop(7)
      }
      else {
        action.lex match {
          case "+" => resultStack.push(add(resultStack.pop,result))
          case "-" => resultStack.push(substract(resultStack.pop,result))
          case "*" => resultStack.push(multiply(resultStack.pop,result))
          case "/" => resultStack.push(divide(resultStack.pop,result))
          case "^" => resultStack.push(power(resultStack.pop,result))
          case _ => stop(7)
        }
      }
    }
  }
}

def isNumber(r: Result) = {
  (r.rt == Result.rtInteger) || (r.rt == Result.rtDouble)
}

def isDouble(r1: Result, r2: Result) = {
  (r1.rt == Result.rtDouble) || (r2.rt == Result.rtDouble)
}

def minus(r: Result) = Result(r.pos,if (r.lex.startsWith("-")) r.lex.substring(1) else "-"+r.lex,r.rt)
```

```
def add(r1: Result,r2: Result) = {
  if ((isNumber(r1))&&(isNumber(r2))) {
    if (isDouble(r1,r2)) Result(r2.pos,""+(r1.lex.toDouble+r2.lex.toDouble),Result.rtDouble)
    else Result(r2.pos,""+(r1.lex.toInt+r2.lex.toInt),Result.rtInteger)
  }
  else {
   stop(8)
   null
  }
}

def substract(r1: Result,r2: Result) = {
  if ((isNumber(r1))&&(isNumber(r2))) {
    if (isDouble(r1,r2)) Result(r2.pos,""+(r1.lex.toDouble-r2.lex.toDouble),Result.rtDouble)
    else Result(r2.pos,""+(r1.lex.toInt-r2.lex.toInt),Result.rtInteger)
  }
  else {
   stop(8)
   null
  }
}

def multiply(r1: Result,r2: Result) = {
  if ((isNumber(r1))&&(isNumber(r2))) {
    if (isDouble(r1,r2)) Result(r2.pos,""+(r1.lex.toDouble*r2.lex.toDouble),Result.rtDouble)
    else Result(r2.pos,""+(r1.lex.toInt*r2.lex.toInt),Result.rtInteger)
  }
  else {
   stop(8)
   null
  }
}

def divide(r1: Result,r2: Result) = {
  if ((isNumber(r1))&&(isNumber(r2))) {
    Result(r2.pos,""+(r1.lex.toDouble/r2.lex.toDouble),Result.rtDouble)
  }
  else {
   stop(8)
   null
  }
}

def power(r1: Result,r2: Result) = {
  if ((isNumber(r1))&&(isNumber(r2))) {
    Result(r2.pos,""+(math.pow(r1.lex.toDouble,r2.lex.toDouble)),Result.rtDouble)
  }
  else {
   stop(8)
   null
  }
}
```

```
  def calculate(expr: String) = {
    this.expr=expr
    error = -1
    actionStack.clear
    resultStack.clear
    ind = -1
    open
    while ((ind<expr.length)&&(!isError)) {
      if (spaces.indexOf(char) != -1) next
      else if (""+char).matches("\\d")) nextNumber
      else if (operators.indexOf(char) != -1) nextOperator
      else if (char=='(') open
      else if (char==')') close
      else stop(2)
    }
    if (!isError){
      close
      if (opened!=0) stop(6)
      else if ((!actionStack.isEmpty)||(resultStack.size!=1)) stop(7)
    }
  }

  def result = if (resultStack.size==1) resultStack.top.lex else ""

  def main(args: Array[String]): Unit = {
    if (args.length>0) {
      var expr = ""
      if ((args(0)=="-f")&&(args.length>1)) {
        try {
          expr = Source.fromFile(args(1)).mkString
        }
        catch { case _ : Throwable => println("Ошибка чтения файла "+args(1))}
      } else expr = args(0)
      if (expr!="") {
        calculate(expr)
        if (isError) println(errorStr)
        else println(result)
      }
    }
    else println("Отсутствуют аргументы: expression или -f filename")
  }
} // Конец файла Calc.scala
```