|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления ​​​

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии ​​​

**Отчет по лабораторной работе**

**«Синтаксический разбор с использованием метода рекурсивного спуска»**

**по курсу «Конструирование компиляторов»**

**Вариант 5**

Выполнил студент группы ИУ7-21М Доманов К. И.

Проверил Ступников А. А.

*2022 г.*

**Описание задания**

Цель работы: приобретение практических навыков реализации синтаксического анализа с использованием метода рекурсивного спуска.

В процессе выполнения лабораторной работы в соответствии с вариантом 5, необходимо дополнить представленную грамматику, а также для модифицированной грамматики написать программу нисходящего синтаксического анализа с использованием метода рекурсивного спуска.

**Теоретическая часть**

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является метод рекурсивного спуска (recursive descent method). Метод основан на «зашивании» правил грамматики непосредственно в управляющие конструкции распознавателя. Синтаксические анализаторы, работающие по методу рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первая буква L в названии связана с тем, что входная цепочка читается слева направо, вторая буква L означает, что строится левый вывод входной цепочки, 1 означает, что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки.

В методе рекурсивного спуска полностью сохраняются идеи нисходящего разбора, принятые в LL(1)- грамматиках:

• происходит последовательный просмотр входной строки слева-направо;

• очередной символ входной строки является основанием для выбора одной из правых частей правил группы при замене текущего нетерминала;

• терминальные символы входной строки и правой части правила «взаимно уничтожаются»;

• обнаружение нетерминала в правой части рекурсивно повторяет этот

**Исходные данные**

Рассматривается грамматика выражений с правилами. Грамматика для варианта 5 представлена на рисунках 1 и 2.

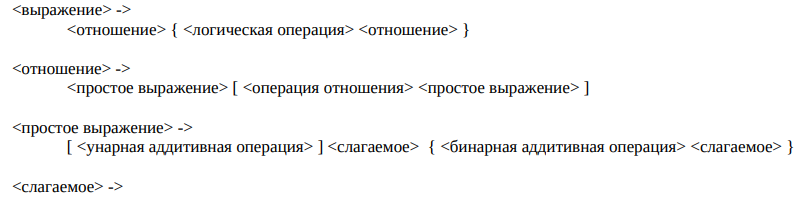


Рисунок 1 – Исходная грамматика

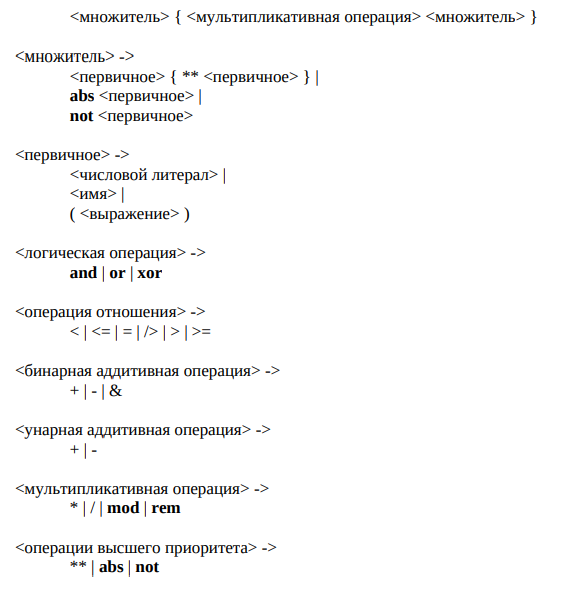


Рисунок 2 – Исходная грамматика

Необходимо дополнить грамматику блоком, состоящим из последовательности операторов присваивания. Для реализации предлагаются два варианта расширенной грамматик: грамматика в стиле Алгол-Паскаль и грамматика в стиле Си.

В качестве дополненной грамматики, была выбрана грамматика в стиле Си, представленная на рисунке 3.

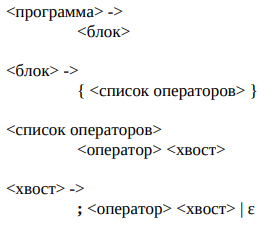




Рисунок 3 – Исходная грамматика

**Построение синтаксического анализатора**

В ходе лабораторной работы, был реализован LL(1)-парсер. Результат работы программы для входной цепочки {x = -1 > (p and (1\*\*(-p) = abs(-1))) ;x=p\*\*p ;;x=p} представлен на рисунках 4 – 5.

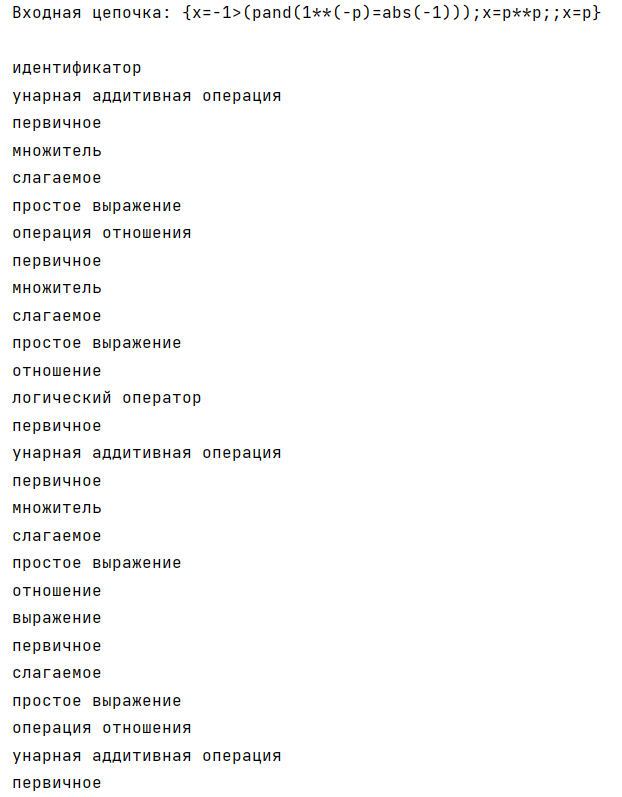


Рисунок 4 – Результат работы программы

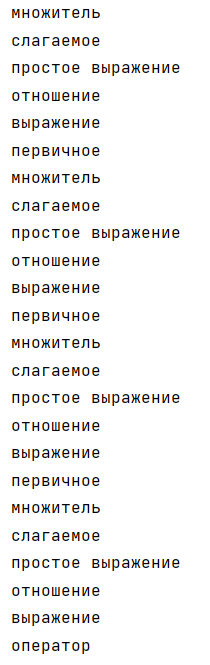
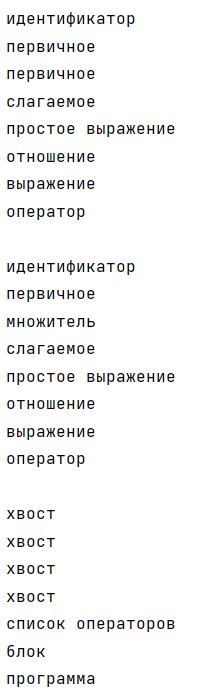
 

Рисунок 5– Результат работы программы

**Текст программы**

run.py

from Parser import Parser  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 input\_string = '{x = -1 > (p and (1\*\*(-p) = abs(-1))) ;x=p\*\*p ;;x=p}'  
 *# input\_string = '{x = -1 mul p / 1 rem p; ; ; x = p + 1 \*\* 1 + not p }'* input\_string = input\_string.replace(' ', '')  
 print('Входная цепочка: ' + input\_string + '\n')  
 parser = Parser(input\_string)

Parser.py

class Parser:  
  
 def \_\_init\_\_(self, str):  
 self.str = str  
 self.i = 0  
 self.is\_error = False  
 self.error = ''  
 self.program()  
  
 def program(self):  
 if self.block():  
 print('программа')  
 return True  
 else:  
 self.exception('ожидается программный блок')  
 print(self.error)  
 return False  
  
 def block(self):  
 try:  
 if self.str[self.i] != '{':  
 *# return False* return self.exception('ожидается {')  
 self.i = self.i + 1  
 if not self.operators\_list():  
 return self.exception('ожидается список операторов')  
 if self.str[self.i] != '}':  
 *# return False* return self.exception('ожидается }')  
 self.i = self.i + 1  
 print('блок')  
 return True  
  
 except:  
 return False  
  
 def operators\_list(self):  
 if self.str[self.i] == '}':  
 return True  
 if not self.operator():  
 return self.exception('ожидается оператор')  
 if not self.tail():  
 return self.exception('ожидается хвост')  
 print('список операторов')  
 return True  
  
 def operator(self):  
 if self.str[self.i] == 'x':  
 self.i = self.i + 1  
 print('идентификатор')  
 if self.str[self.i] == '=':  
 self.i = self.i + 1  
 if not self.expression():  
 return self.exception('ожидается выражение')  
 else:  
 return self.exception('ожидается символ =')  
 print('оператор\n')  
 return True  
 elif self.str[self.i] == '{':  
 self.i = self.i + 1  
 self.block()  
 print('оператор')  
 return True  
 else:  
 if self.str[self.i] == ';':  
 *# self.i = self.i + 1* return True  
 else:  
 return self.exception('ошибка в определении идентификатора')  
  
 def tail(self):  
 if self.str[self.i] == ';':  
 self.i = self.i + 1  
 if not self.operator():  
 return self.exception('ожидается оператор')  
 if not self.tail():  
 return self.exception('ожидается хвост')  
 print('хвост')  
 return True  
  
 def expression(self):  
 if not self.relation():  
 return self.exception('ожидается отношение')  
 while True:  
 if self.log\_op():  
 print('логический оператор')  
 if not self.relation():  
 return self.exception('ожидается отношение')  
 else:  
 break  
 print('выражение')  
 return True  
  
 def relation(self):  
 if not self.simple\_expression():  
 return self.exception('ожидается простое выражение')  
 if self.rel\_op():  
 print('операция отношения')  
 if not self.simple\_expression():  
 return self.exception('ожидается простое выражение')  
 print('отношение')  
 return True  
  
 def simple\_expression(self):  
 if self.un\_add\_op():  
 print('унарная аддитивная операция')  
 if not self.term():  
 return self.exception('ошибка в определении слагаемого')  
 while True:  
 if self.bin\_add\_op():  
 print('бинарная аддитивная операция')  
 if not self.term():  
 return self.exception('ожидается слагаемое')  
 else:  
 break  
 print('простое выражение')  
 return True  
  
 def term(self):  
 if not self.factor():  
 return self.exception('ошибка в определении множителя')  
 while True:  
 if self.mul\_op():  
 print('мультипликативный оператор')  
 if not self.factor():  
 return self.exception('ожидается множитель')  
 else:  
 break  
 print('слагаемое')  
 return True  
  
 def factor(self):  
 if self.str[self.i] == 'a':  
 if self.str[self.i + 1] == 'b':  
 if self.str[self.i + 2] == 's':  
 self.i = self.i + 3  
 if not self.primary():  
 return self.exception('ошибка в определении первичного')  
 elif self.str[self.i] == 'n':  
 if self.str[self.i + 1] == 'o':  
 if self.str[self.i + 2] == 't':  
 self.i = self.i + 3  
 if not self.primary():  
 return self.exception('ошибка в определении первичного')  
 else:  
 if self.primary():  
 if self.str[self.i] == '\*':  
 if self.str[self.i + 1] == '\*':  
 self.i = self.i + 2  
 if not self.primary():  
 return self.exception('ошибка в определении первичного')  
 return True  
 else:  
 return False  
 else:  
 return self.exception('ошибка в определении первичного')  
 print('множитель')  
 return True  
  
 def primary(self):  
 if self.str[self.i] == '1':  
 self.i = self.i + 1  
 elif self.str[self.i] == 'p':  
 self.i = self.i + 1  
 elif self.str[self.i] == '(':  
 self.i = self.i + 1  
 if not self.expression():  
 return self.exception('ожидается выражение')  
 if self.str[self.i] == ')':  
 self.i = self.i + 1  
 else:  
 return self.exception('ожидается )')  
 else:  
 return self.exception('ошибка в определении первичного')  
 print('первичное')  
 return True  
  
 def mul\_op(self):  
 if self.mul\_op\_mul() | self.mul\_op\_div() | self.mul\_op\_mod() | self.mul\_op\_rem():  
 return True  
 else:  
 return False  
  
 def mul\_op\_mul(self):  
 if self.str[self.i] == 'm':  
 if self.str[self.i + 1] == 'u':  
 if self.str[self.i + 2] == 'l':  
 self.i = self.i + 3  
 return True  
 return False  
 return False  
 return False  
  
 def mul\_op\_div(self):  
 if self.str[self.i] == '/':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def mul\_op\_mod(self):  
 if self.str[self.i] == 'm':  
 if self.str[self.i + 1] == 'o':  
 if self.str[self.i + 2] == 'd':  
 self.i = self.i + 3  
 return True  
 return False  
 return False  
 return False  
  
 def mul\_op\_rem(self):  
 if self.str[self.i] == 'r':  
 if self.str[self.i + 1] == 'e':  
 if self.str[self.i + 2] == 'm':  
 self.i = self.i + 3  
 return True  
 return False  
 return False  
 return False  
  
 def bin\_add\_op\_m(self):  
 if self.str[self.i] == '-':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def bin\_add\_op\_p(self):  
 if self.str[self.i] == '+':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def bin\_add\_op\_amp(self):  
 if self.str[self.i] == '&':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def bin\_add\_op(self):  
 if self.bin\_add\_op\_m() | self.bin\_add\_op\_p() | self.bin\_add\_op\_amp():  
 return True  
 return False  
  
 def un\_add\_op(self):  
 if self.un\_add\_op\_p() | self.un\_add\_op\_m():  
 return True  
 return False  
  
 def un\_add\_op\_p(self):  
 if self.str[self.i] == '+':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def un\_add\_op\_m(self):  
 if self.str[self.i] == '-':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def rel\_op(self):  
 if self.rel\_op\_mr() | self.rel\_op\_m() | self.rel\_op\_r() | self.rel\_op\_nr() | self.rel\_op\_br() | self.rel\_op\_b():  
 return True  
 else:  
 return False  
  
 def rel\_op\_m(self):  
 if self.str[self.i] == '<':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def rel\_op\_mr(self):  
 if self.str[self.i] == '<':  
 if self.str[self.i + 1] == '=':  
 self.i = self.i + 2  
 return True  
 return False  
 return False  
  
 def rel\_op\_r(self):  
 if self.str[self.i] == '=':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def rel\_op\_nr(self):  
 if self.str[self.i] == '/':  
 if self.str[self.i + 1] == '>':  
 self.i = self.i + 2  
 return True  
 return False  
 return False  
  
 def rel\_op\_b(self):  
 if self.str[self.i] == '>':  
 self.i = self.i + 1  
 return True  
 return False  
  
 def rel\_op\_br(self):  
 if self.str[self.i] == '>':  
 if self.str[self.i + 1] == '=':  
 self.i = self.i + 2  
 return True  
 return False  
 return False  
  
 def log\_op(self):  
 if self.log\_op\_and() | self.log\_op\_or() | self.log\_op\_xor():  
 return True  
 return False  
  
 def log\_op\_and(self):  
 if self.str[self.i] == 'a':  
 if self.str[self.i + 1] == 'n':  
 if self.str[self.i + 2] == 'd':  
 self.i = self.i + 3  
 return True  
 return False  
 return False  
 return False  
  
 def log\_op\_or(self):  
 if self.str[self.i] == 'o':  
 if self.str[self.i + 1] == 'r':  
 self.i = self.i + 2  
 return True  
 return False  
 return False  
  
 def log\_op\_xor(self):  
 if self.str[self.i] == 'x':  
 if self.str[self.i + 1] == 'o':  
 if self.str[self.i + 2] == 'r':  
 self.i = self.i + 3  
 return True  
 return False  
 return False  
 return False  
  
 def exception(self, error\_string):  
 if not self.is\_error:  
 self.error = 'Позиция ' + str(self.i) + ': ' + error\_string  
 self.is\_error = True  
 return False