|  |  |
| --- | --- |
|  | **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 1**

Выполнил студент группы ИУ7-22М Андреев А.А.

Проверил Ступников А. А.

*2024 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, 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 представлена на рисунках 1 и 2.

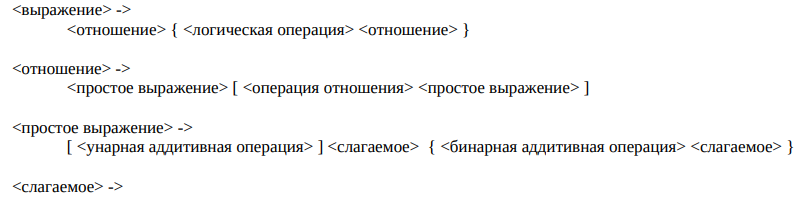


Рисунок 1 – Исходная грамматика

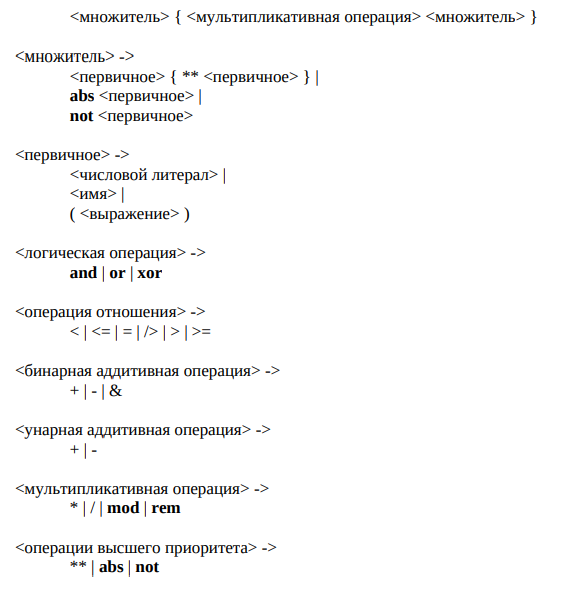


Рисунок 2 – Исходная грамматика

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.

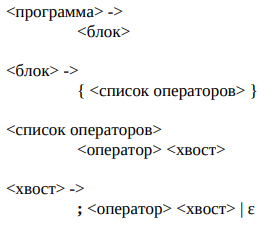




Рисунок 3 – Исходная грамматика

**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.

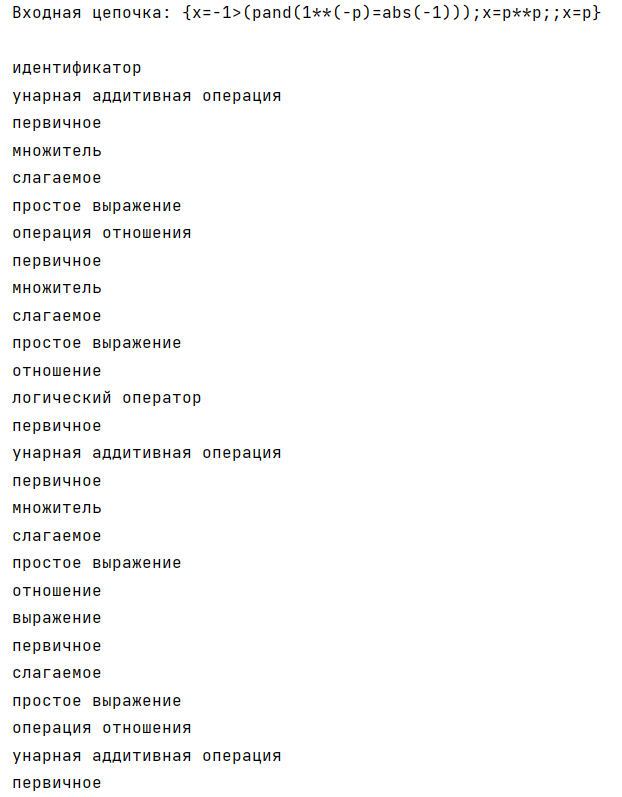


Рисунок 4 – Результат работы программы

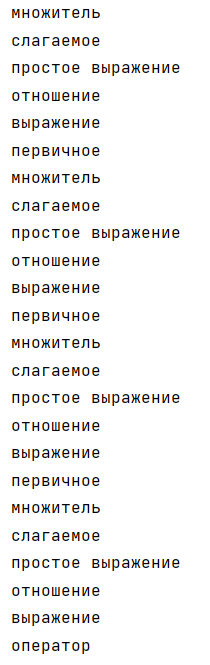
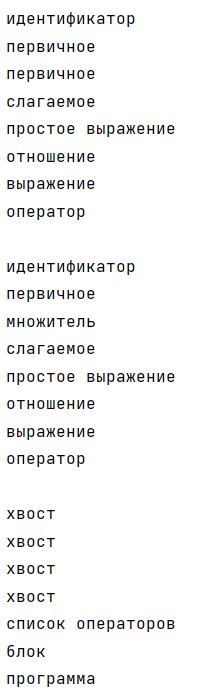
 

Рисунок 5– Результат работы программы

**Текст программы**

run.py

1. **from** parser **import** Parser
2. tests = [
3. '{ a := 3 ; aaa := p ; { a := p ; x := p and not p } }',
4. '{ a := ( abs p and ( 1 \*\* 25 ) ) }',
5. # '{ a? := 1 and - ( + p and 1 \*\* 2 \*\* 455 ) }',
6. '{ c := not ( p mod 5 / ( 2 + 2 ) ) }',
7. '{ c := abs ( - p + 2 & 4 mod 5 / ( 2 ) ) }',
8. '{ ll := ( p and - 22 ) }'
9. ]
10. **def** **main**(input\_string):
11. input\_string\_split = list(input\_string.strip().split())
12. print()
13. print('Tokens')
14. print(input\_string\_split)
15. parser = Parser(input\_string\_split)
16. **if** \_\_name\_\_ == "\_\_main\_\_":
18. **for** \_ **in** tests:
19. main(\_)

Parser.py

1. **class** **Parser**:
2. **def** **\_\_init\_\_**(self, input):
3. self.input = input
4. self.i = 0
5. self.program()
6. **def** **program**(self):
7. **if** self.block():
8. print('program')
9. **return** True
10. **else**:
11. **return** False
12. **def** **error**(self):
13. print('Syntax error on ', self.i)
14. exit()
15. **def** **get\_current\_token**(self):
16. **if** self.i < len(self.input):
17. **return** self.input[self.i]
18. **return** None
19. **def** **block**(self):
20. **if** self.get\_current\_token() == '{':
21. self.i = self.i + 1
22. **if** self.get\_current\_token() == '}':
23. print('block')
24. **return** True
25. **elif** self.operators\_list():
26. print('operator\_list')
27. **if** self.get\_current\_token() == '}':
28. self.i = self.i + 1
29. print('block')
30. **return** True
31. **else**:
32. self.error()
33. self.error()
34. **def** **operators\_list**(self):
35. **if** self.operator():
36. print('operator')
37. **if** self.tail():
38. print('tail')
39. **return** True
40. self.error()
41. self.error()
42. **def** **operator**(self):
43. **if** self.get\_current\_token().isalpha():
44. self.i = self.i + 1
45. print('identifier')
46. **if** self.get\_current\_token() == ':=':
47. self.i = self.i + 1
48. **if** self.expression():
49. print('expression')
50. **return** True
51. self.error()
52. self.error()
53. **elif** self.block():
54. **return** True
55. **else**:
56. **return** self.error()
57. **def** **primary**(self):
58. **if** self.get\_current\_token() == 'p' **or** self.get\_current\_token().isdigit():
59. self.i = self.i + 1
60. **return** True
61. **if** self.get\_current\_token() == '(':
62. self.i = self.i + 1
63. **if** self.expression():
64. print('expression')
65. **if** self.get\_current\_token() == ')':
66. self.i = self.i + 1
67. **return** True
68. self.error()
69. **def** **factor**(self):
70. **if** self.get\_current\_token() **in** ('abs', 'not'):
71. self.i = self.i + 1
72. **if** self.primary():
73. print('primary')
74. **return** True
75. self.error()
76. **if** self.primary():
77. print('primary')
78. **while** True:
79. **if** self.get\_current\_token() == '\*\*':
80. self.i = self.i + 1
81. **if** self.primary():
82. print('primary')
83. **continue**
84. self.error()
85. **else**:
86. **break**
87. **return** True
88. **def** **term**(self):
89. **if** self.factor():
90. print('factor')
91. **while** True:
92. **if** self.get\_current\_token() **in** ('\*', '/', 'mod', 'rem'):
93. print('mul operation')
94. self.i = self.i + 1
95. **if** self.factor():
96. print('factor')
97. **continue**
98. self.error()
99. **else**:
100. **break**
101. **return** True
102. **def** **simple\_expression**(self):
103. **if** self.get\_current\_token() **in** ('-', '+'):
104. self.i = self.i + 1
105. print('un add operation')
106. **if** self.term():
107. print('term')
108. **while** True:
109. **if** self.get\_current\_token() **in** ('+', '-', '&'):
110. self.i = self.i + 1
111. print('bin add operation')
112. **if** self.term():
113. print('term')
114. **continue**
115. self.error()
116. **else**:
117. **break**
118. **return** True
119. **def** **expression**(self):
120. **if** self.relation():
121. print('relation')
122. **if** self.get\_current\_token() **in** ('and', 'or', 'xor'):
123. self.i = self.i + 1
124. print('logic operation')
125. **if** self.relation():
126. print('relation')
127. **return** True
128. self.error()
129. **return** True
130. **else**:
131. self.error()
132. **def** **relation**(self):
133. **if** self.simple\_expression():
134. print('simple expression')
135. **if** self.get\_current\_token() **in** ('<', '<=', '==', '/>', '>=', '>'):
136. self.i = self.i + 1
137. print('relation operation')
138. **if** self.simple\_expression():
139. print('simple expression')
140. **return** True
141. self.error()
142. **return** True
143. **else**:
144. self.error()
145. **def** **tail**(self):
146. **if** self.get\_current\_token() == ';':
147. self.i = self.i + 1
148. **if** self.operator():
149. print('operator')
150. **if** self.tail():
151. print('tail')
152. **return** True
153. self.error()
154. self.error()
155. **return** True