Государственное образовательное учреждение
высшего профессионального образования
“Московский государственный технический университет
имени Н.Э. Баумана”

Отчет

по лабораторной работе №4
по курсу “Конструирование компиляторов”
по теме
“Синтаксический анализатор операторного предшествия”

Студент: Адамова И.О.
Группа: ИУ7-22м
Вариант: 2
Преподаватель: Ступников А.А.

Москва, 2020

Цель работы

Приобретение практических навыков реализации таблично управляемых синтаксических анализаторов на примере анализатора операторного предшествия.

Задачи работы

1) Ознакомиться с основными понятиями и определениями, лежащими в основе синтаксического анализа операторного предшествия.
2) Изучить алгоритм синтаксического анализа операторного предшествия.
3) Разработать, тестировать и отладить программу синтаксического анализа в соответствии с предложенным вариантом.
4) Включить в программу синтаксического анализа семантические действия для реализации синтаксически управляемого перевода инфиксного выражения в обратную польскую запись.

Описание работы программы

Программа принимает на вход строку, разбивает её на символы, дополняет её новым символом “$” в конце. Также имеется стек, изначально содержащий символ “$”. Далее осуществляется просмотр символов в строке. Верхний элемент в стеке сравнивается с первым символом строки по отношению операторного предшествия. В случае, если верхний элемент уступает приоритет или того же приоритета, первый элемент строки из нее изымается и помещается в стек. Иначе из стека поочередно изымаются элементы, пока отношение очередного из них не будет “<.” по отношению к новому верхнему элементу. Попутно эти элементы добавляются к строке, которая в результате образует польскую инверсную запись данного выражения. Отношение приоритета получается из функции, построенной на основе матрицы операторного предшествия. В этой матрице пустая клетка символизирует различные синтаксические ошибки. Также это перенесено в программную реализацию, поэтому исключения возбуждаются именно при поиске отношения приоритета.

Матрица отношений операторного предшествия

	(	)	const	*	+	>	$
(	<.	=.	<.	<.	<.	<.	
)		>.		>.	>.	>.	>.
const		>.		>.	>.	>.	>.
*	<.	>.	<.	>.	>.	>.	>.
+	<.	>.	<.	<.	>.	>.	>.
>	<.	>.	<.	<.	<.		>.
$	<.		<.	<.	<.	<.	

Тесты

Для проверки правильности работы программы были проведены следующие тесты.

Тест 1

```
'( 1 + 2 ) * 4 > 7 * ( 2 - 5 / 2 )'
```

Вывод: `rpn: 1 2 + 4 * 7 2 5 2 / - * >`

Этот текст программы соответствует грамматике.

Тест 2

```
') 1 * 7'
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: Right bracket is not balanced`

Тест 3

```
'( 3 * 4 ) / 5 * ('
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: Right bracket is missed`

Тест 4

```
'( 3 * 4 ) / 5 4'
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: Operator is missed`

Тест 5

```
'3 + 4 + 7 * ( 9 - 3 / ( 3 + 8 ) )'
```

Вывод: `rpn: 3 4 + 7 9 3 3 8 + / - * +`

Таким образом, по результатам проведенных тестов можно сделать вывод, что программа работает корректно.

Заключение

В результате лабораторной работы была разработана программа на языке Python, реализующая синтаксический анализатор операторного предшествия. По результатам тестов можно сделать вывод, что программа работает корректно.

Листинг

main.py

```
from parser import Parser

if __name__ == "__main__":
    parser = Parser()

    test1 = '( 1 + 2 ) * 4 > 7 * ( 2 - 5 / 2 )' # 1 2 + 4 * 7 2 5 2 / - * >
    test2 = ') 1 * 7' # Right bracket is not balanced
    test3 = '( 3 * 4 ) / 5 * (' # Right bracket is missed
    test4 = '( 3 * 4 ) / 5 4' # Operator is missed
    test5 = '3 + 4 + 7 * ( 9 - 3 / ( 3 + 8 ) )' # 3 4 + 7 9 3 3 8 + / - * +

    test = test5
    parser.read_test(test)
    parser.analyze()
```

parser.py

```
class ParserSyntaxError(Exception):
    def __init__(self, *args):
        if args:
               self.message = args[0]
        else:
               self.message = None

    def __str__(self):
        if self.message:
               return f'ParserSyntaxError: {self.message}'
        else:
               return f'ParserSyntaxError was raised'

class Parser:
    def __init__(self):
        pass

    def read_test(self, string_of_symbols: str):
        self.symbols = string_of_symbols.split()
        self.symbols.append('$')
        self.number_of_current_symbol = 0
        self.length_of_list = len(self.symbols)
```

```
        self.current_symbol = self.symbols[self.number_of_current_symbol]
        self.stack = ['$']
        self.rpn = ''

    def read_program_from_file(filename: str):
        with open(filename) as file:
               data = file.read()
               return Parser(string_of_symbols = data)

    def next(self):
        self.number_of_current_symbol += 1
        self.current_symbol = self.symbol[self.number_of_current_symbol]

    def print_symbols(self):
        print(f'Symbols: {self.symbols}')

    def symbol_to_rpn(self, c: str):
        if c not in ('(', ')'):
               self.rpn = self.rpn + c + ' '

    def __get_precedence(self, a: str, b: str):
        constant = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
        operation_of_multiplication = ['*', '/']
        operation_of_addition = ['+', '-']
        operation_of_relation = ['<', '<=', '==', '<>', '>', '>=']

        if a == ')':
               if b == '(' or b in constant:
                      raise ParserSyntaxError('Operator is missed')
               if     b in operation_of_multiplication or b in operation_of_addition
or b ==')' or b == '$' \
                      or b in operation_of_relation:
                      return '>'

        if a in constant:
               if b == '(' or b in constant:
                      raise ParserSyntaxError('Operator is missed')
               if     b in operation_of_multiplication or b in operation_of_addition
or b ==')' or b == '$' \
                      or b in operation_of_relation:
                      return '>'

        if a in operation_of_multiplication:
               if b == '(' or b in constant:
                      return '<'
               if     b in operation_of_multiplication or b in operation_of_addition
or b ==')' or b == '$' \
               or b in operation_of_relation:
                      return '>'

        if a in operation_of_addition:
               if b == '(' or b in constant or b in operation_of_multiplication:
                      return '<'
```

```
            if     b in operation_of_addition or b ==')' or b == '$' or b in
operation_of_relation:
                    return '>'

        if a == '(':
            if b == '(' or b in constant or b in operation_of_multiplication or b
in operation_of_addition \
            or b in operation_of_relation:
                    return '<'
            if b == ')':
                    return '='
            if b == '$':
                    raise ParserSyntaxError('Right bracket is missed')

        if a == '$':
            if b == '(' or b in constant or b in operation_of_multiplication or b
in operation_of_addition \
            or b in operation_of_relation:
                    return '<'
            if b == ')':
                    raise ParserSyntaxError('Right bracket is not balanced')
            if b == '$':
                    raise ParserSyntaxError('Operand is missed')

        if a in operation_of_relation:
            if b == '(' or b in constant or b in operation_of_multiplication or b
in operation_of_addition:
                    return '<'
            if b == ')' or b == '$':
                    return '>'
            if b in operation_of_relation:
                    raise ParserSyntaxError('Two comparisons in one expression are
forbidden')


    def analyze(self):
        if self.stack[-1] == '$' and self.symbols[0] == '$':
            raise ParserSyntaxError('Operand is missed')
        while 100:
            if self.stack[-1] == '$' and self.symbols[0] == '$':
                break
            precedence = self.__get_precedence(self.stack[-1], self.symbols[0])
            if precedence == '<' or precedence == '=':
                self.stack.append(self.symbols[0])
                self.symbols = self.symbols[1:]
            elif precedence == '>':
                while True:
                    pop_stack = self.stack[-1]
                    self.stack = self.stack[:-1]
                    self.symbol_to_rpn(pop_stack)
                    if self.__get_precedence(self.stack[-1], pop_stack) ==
'<':
                        break
            print(f'stack: {self.stack}\nsymbols: {self.symbols}')
```

```
print(f'rpn: {self.rpn}')
```