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 5 представлена на рисунках 1 и 2.

<выражение> ->
<отношение> { <логическая операция> <отношение> }

<отношение> ->
<простое выражение> [<операция отношения> <простое выражение>]

<простое выражение> ->
[<унарная аддитивная операция>] <слагаемое> { <бинарная аддитивная операция> <слагаемое> }

<слагаемое> ->

Рисунок 1 – Исходная грамматика

<множитель> { <мультипликативная операция> <множитель> }

<множитель> ->
<первичное> { ** <первичное> } |
abs <первичное> |
not <первичное>

<первичное> ->
<числовой литерал> |
<имя> |
(<выражение>)

<логическая операция> ->
and | **or** | **xor**

<операция отношения> ->
< | <= | = | /> | > | >=

<бинарная аддитивная операция> ->
+ | - | &

<унарная аддитивная операция> ->
+ | -

<мультипликативная операция> ->
* | / | **mod** | **rem**

<операции высшего приоритета> ->
** | **abs** | **not**

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.

<программа> ->
    <блок>

<блок> ->
    { <список операторов> }

<список операторов>
    <оператор> <хвост>

<хвост> ->
    ; <оператор> <хвост> | ε

<оператор> ->
    <идентификатор> = <выражение> |
    <блок>

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.

```
Входная цепочка: {x=-1>(pand(1**(-p)=abs(-1)));x=p**p;;x=p}

идентификатор
унарная аддитивная операция
первичное
множитель
слагаемое
простое выражение
операция отношения
первичное
множитель
слагаемое
простое выражение
отношение
логический оператор
первичное
унарная аддитивная операция
первичное
множитель
слагаемое
простое выражение
отношение
выражение
первичное
слагаемое
простое выражение
операция отношения
унарная аддитивная операция
первичное
```

Рисунок 4 – Результат работы программы

```
                        идентификатор
множитель               первичное
слагаемое               первичное
простое выражение       слагаемое
отношение               простое выражение
выражение               отношение
первичное               выражение
множитель               оператор
слагаемое
простое выражение       идентификатор
отношение               первичное
выражение               множитель
первичное               слагаемое
множитель               простое выражение
слагаемое               отношение
простое выражение       выражение
отношение               оператор
выражение
первичное               хвост
множитель               хвост
слагаемое               хвост
простое выражение       хвост
отношение               список операторов
выражение               блок
оператор                программа
```

Рисунок 5– Результат работы программы

Текст программы

run.py

```
from parser import Parser

tests = [
    '{ a := 3 ; aaa := p ; { a := p ; x := p and not p } }',
    '{ a := ( abs p and ( 1 ** 25 ) ) }',
    # '{ a? := 1 and - ( + p and 1 ** 2 ** 455 ) }',
    '{ c := not ( p mod 5 / ( 2 + 2 ) ) }',
    '{ c := abs ( - p + 2 & 4 mod 5 / ( 2 ) ) }',
    '{ ll := ( p and - 22 ) }'
     ]

     def main(input_string):
         input_string_split = list(input_string.strip().split())
         print()
         print('Tokens')
         print(input_string_split)
         parser = Parser(input_string_split)


     if __name__ == "__main__":

         for _ in tests:
             main(_)
```

Parser.py

```
class Parser:

    def __init__(self, input):
        self.input = input
        self.i = 0
        self.program()

    def program(self):
        if self.block():
                print('program')
                return True
            else:
                return False

         def error(self):
             print('Syntax error on ', self.i)
             exit()

         def get_current_token(self):
             if self.i < len(self.input):
                 return self.input[self.i]
             return None


         def block(self):
             if self.get_current_token() == '{':
                 self.i = self.i + 1
                 if self.get_current_token() == '}':
                     print('block')
                     return True
                 elif self.operators_list():
                     print('operator_list')
```

```
                    if self.get_current_token() == '}':
                        self.i = self.i + 1
                        print('block')
                        return True
                    else:
                        self.error()
                self.error()

        def operators_list(self):
            if self.operator():
                print('operator')
                if self.tail():
                    print('tail')
                    return True
                self.error()
            self.error()

        def operator(self):
            if self.get_current_token().isalpha():
                self.i = self.i + 1
                print('identifier')
                if self.get_current_token() == ':=':
                    self.i = self.i + 1
                    if self.expression():
                        print('expression')
                        return True
                    self.error()
                self.error()
            elif self.block():
                return True
            else:
                return self.error()

        def primary(self):
            if self.get_current_token() == 'p' or
  self.get_current_token().isdigit():
                self.i = self.i + 1
                return True
            if self.get_current_token() == '(':
                self.i = self.i + 1
                if self.expression():
                    print('expression')
                    if self.get_current_token() == ')':
                        self.i = self.i + 1
                        return True
            self.error()

        def factor(self):
            if self.get_current_token() in ('abs', 'not'):
                self.i = self.i + 1
                if self.primary():
                    print('primary')
                    return True
                self.error()
            if self.primary():
                print('primary')
                while True:
                    if self.get_current_token() == '**':
                        self.i = self.i + 1
                        if self.primary():
                            print('primary')
                            continue
                        self.error()
                    else:
                        break
            return True
```

```
98.
99.         def term(self):
100.            if self.factor():
101.                print('factor')
102.                while True:
103.                    if self.get_current_token() in ('*', '/',
    'mod', 'rem'):
104.                        print('mul operation')
105.                        self.i = self.i + 1
106.                        if self.factor():
107.                            print('factor')
108.                            continue
109.                        self.error()
110.                    else:
111.                        break
112.                return True
113.
114.        def simple_expression(self):
115.            if self.get_current_token() in ('-', '+'):
116.                self.i = self.i + 1
117.                print('un add operation')
118.            if self.term():
119.                print('term')
120.            while True:
121.                if self.get_current_token() in ('+', '-', '&'):
122.                    self.i = self.i + 1
123.                    print('bin add operation')
124.                    if self.term():
125.                        print('term')
126.                        continue
127.                    self.error()
128.                else:
129.                    break
130.            return True
131.
132.        def expression(self):
133.            if self.relation():
134.                print('relation')
135.                if self.get_current_token() in ('and', 'or',
    'xor'):
136.                    self.i = self.i + 1
137.                    print('logic operation')
138.                    if self.relation():
139.                        print('relation')
140.                        return True
141.                    self.error()
142.                return True
143.            else:
144.                self.error()
145.
146.        def relation(self):
147.            if self.simple_expression():
148.                print('simple expression')
149.                if self.get_current_token() in ('<', '<=', '==',
    '/>', '>=', '>'):
150.                    self.i = self.i + 1
151.                    print('relation operation')
152.                    if self.simple_expression():
153.                        print('simple expression')
154.                        return True
155.                    self.error()
156.                return True
157.            else:
158.                self.error()
159.
160.        def tail(self):
```

```
161.             if self.get_current_token() == ';':
162.                 self.i = self.i + 1
163.                 if self.operator():
164.                     print('operator')
165.                     if self.tail():
166.                         print('tail')
167.                         return True
168.                     self.error()
169.                 self.error()
170.             return True
```