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.

<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

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
```