Отчет по лабораторной работе № 2

«Разработка синтаксического анализатора»

Выполнил: студент группы P3317

Плюхин Д.А.

Преподаватель: Лаздин Артур Вячеславович

# **Цель работы**

Разработать и отладить синтаксический анализатор для заданной грамматики.

# **БНФ реализуемого языка**

Исходная грамматика

<Программа> ::= <Объявление переменных> <Описание вычислений>

<Описание вычислений> ::= Begin < Список операторов > End**.**

<Объявление переменных> ::= Var <Список переменных>

<Список переменных> ::= <Идент>; | <Идент> , <Список переменных> | <Идент> ; <Список переменных>

<Список операторов> ::= <Оператор> | <Оператор> <Список операторов>  
<Оператор>::=<Присваивание> |<Сложный оператор>|<Составной оператор>   
<Составной оператор>::= Begin < Список операторов > End

<Присваивание> ::= <Идент> = <Выражение> ;

<Выражение> ::= <Ун.оп.><Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд> | <Подвыражение> <Бин.оп.> <Подвыражение>

<Ун.оп.> ::= - | not

<Бин.оп.> ::= - | + | \* | / | \*\* | > | < | ==

<Операнд> ::= <Идент> | <Const>

<Сложный оператор>::= IF ( < Выражение> ) Оператор

<Идент> ::= <Буква> <Идент> | <Буква>

<Const> ::= <Цифра> <Const> | <Цифра>

Грамматика после устранения левой рекурсии и ввода левой факторизации

<Программа> ::= <Объявление переменных> <Описание вычислений>

<Описание вычислений> ::= Begin < Список операторов > End**.**

<Объявление переменных> ::= Var <Список переменных>

<Список переменных> ::= <Идент> <Продолжение списка переменных>

<Продолжение списка переменных> ::= , <Список переменных> | ; <Конец списка переменных>

<Конец списка переменных> ::= <Список переменных> | <Пустая цепочка>

<Список операторов> ::= <Оператор> <Продолжение списка операторов>

<Продолжение списка операторов> ::= <Список операторов> | <Пустая цепочка>  
<Оператор> ::= <Присваивание> | <Сложный оператор> | <Составной оператор>   
<Составной оператор>::= Begin < Список операторов > End

<Присваивание> ::= <Идент> = <Выражение> ;

<Выражение> ::= <Ун.оп.><Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) <Продолжение подвыражения> | <Операнд> <Продолжение подвыражения>

<Продолжение подвыражения> ::= <Бин.оп.> <Подвыражение> <Продолжение подвыражения> | <Пустая цепочка>

<Ун.оп.> ::= - | not

<Бин.оп.> ::= - | + | \* | / | \*\* | > | < | ==

<Операнд> ::= <Идент> | <Const>

<Сложный оператор> ::= IF ( < Выражение> ) < Оператор >

<Идент> ::= <Буква> <Идент> | <Буква>

<Const> ::= <Цифра> <Const> | <Цифра>

Для начала проверим, является ли полученная грамматика LL(1) – грамматикой с целью синтеза наиболее эффективного анализатора

# **Программа проверки на соответствие классу LL(1)**

def get\_key(item):

if item[1] != None:

key = item[0] + item[1]

else:

key = item[0]

return key

def get\_nullable(grammar):

nullable = {}

changed = False

for rule in grammar:

nullable[rule[0]] = False

for alternative in rule[1]:

for item in alternative:

if (item[0] != None):

nullable[get\_key(item)] = False

changed = True

while (changed == True):

changed = False

for rule in grammar:

for alternative in rule[1]:

all\_nullable = True

for item in alternative:

if (item[0] != None):

if (nullable[get\_key(item)]) == False:

all\_nullable = False

break

if (all\_nullable == True) and (nullable[rule[0]] == False):

nullable[rule[0]] = True

changed = True

return nullable

def get\_terminals(grammar):

non\_terminals = [rule[0] for rule in grammar]

terminals = []

for rule in grammar:

for alternative in rule[1]:

for item in alternative:

try:

non\_terminals.index(item[0])

except Exception as e:

if (item[0] != None):

terminals.append(get\_key(item))

return list(set(terminals))

def not\_contains(lst, values):

for value in values:

try:

lst.index(value)

except:

return True

return False

def get\_first(grammar, nullable):

terminals = get\_terminals(grammar)

non\_terminals = [rule[0] for rule in grammar]

first = {}

for terminal in terminals:

first[terminal] = [terminal]

for non\_terminal in non\_terminals:

first[non\_terminal] = []

#print(first)

changed = True

while (changed == True):

changed = False

for rule in grammar:

for alternative in rule[1]:

if (alternative[0][0] != None) and (not\_contains(first[rule[0]], first[get\_key(alternative[0])])):

first[rule[0]] = list(set(first[rule[0]] + first[get\_key(alternative[0])]))

changed = True

for i in range(1, len(alternative) - 1):

if (nullable[get\_key(alternative[i])] == True) and (not\_contains(first[rule[0]], first[get\_key(alternative[i + 1])])):

first[rule[0]] = list(set(first[rule[0]] + first[get\_key(alternative[i + 1])]))

changed = True

else:

break

return first

def get\_first\_for\_chain(beta\_chain, first):

if beta\_chain == []:

return beta\_chain

result = []

if (get\_key(beta\_chain[0]) != None):

result += first[get\_key(beta\_chain[0])]

for i in range(1, len(beta\_chain) - 1):

if (get\_key(beta\_chain[i]) == None):

continue

if (nullable[get\_key(beta\_chain[i])] == True):

result += first[get\_key(beta\_chain[i + 1])]

else:

break

return result

def is\_chain\_nullable(beta\_chain, nullable):

if beta\_chain == []:

return True

for item in beta\_chain:

if item[0] == None:

continue

if nullable[get\_key(item)] == False:

return False

return True

def get\_follow(grammar, nullable, first):

non\_terminals = [rule[0] for rule in grammar]

follow = {}

for non\_terminal in non\_terminals:

follow[non\_terminal] = []

changed = True

while changed:

changed = False

for non\_terminal in non\_terminals:

#print(non\_terminal)

for rule in grammar:

for alternative in rule[1]:

matched = False

for item in alternative:

if (item[0] == non\_terminal):

matched = True

if matched:

#print(alternative)

beta\_chain = alternative[[get\_key(item) for item in alternative].index(non\_terminal) + 1:]

#print(beta\_chain)

first\_for\_beta\_chain = get\_first\_for\_chain(beta\_chain, first)

if (not\_contains(follow[non\_terminal], first\_for\_beta\_chain)):

follow[non\_terminal] = list(set(follow[non\_terminal] + first\_for\_beta\_chain))

changed = True

if (is\_chain\_nullable(beta\_chain, nullable)) and (not\_contains(follow[non\_terminal], follow[rule[0]])):

#print("before (original): ",follow[non\_terminal])

#print("before (connected): ",follow[rule[0]])

follow[non\_terminal] = list(set(follow[non\_terminal] + follow[rule[0]]))

changed = True

return follow

def contains(lst, value):

try:

lst.index(value)

except ValueError:

return False

return True

nullable = get\_nullable(grammar)

#print(nullable)

first = get\_first(grammar, nullable)

#print(first)

follow = get\_follow(grammar, nullable, first)

#print(follow)

#print(nullable)

terminals = get\_terminals(grammar)

non\_terminals = [rule[0] for rule in grammar]

for x in non\_terminals:

for c in terminals:

counter = 0

#print(x,c)

for rule in grammar:

if (rule[0] != x):

continue

for alternative in rule[1]:

if contains(get\_first\_for\_chain(alternative, first), c) or (contains(follow[x], c) and is\_chain\_nullable(alternative, nullable)):

#print(follow[x])

#print(x,c)

#print(rule[0], " ::= ", alternative)

if counter >= 1:

print("Found collision for non-terminal %s and terminal %s" % (x,c))

#print("===================================================")

counter += 1

# **Результат проверки на соответствие классу LL(1)**

Found collision for non-terminal continuation of subexpression and terminal binary comparasion operator

Found collision for non-terminal continuation of subexpression and terminal binary simple operator

Found collision for non-terminal continuation of subexpression and terminal binary complex operator

Так, грамматика не является LL(1) – грамматикой. Попробуем пойти другим путем - построим простой восходящий синтаксический анализатор.

# **Рекурсивная процедура восходящего анализа**

def replace(program, deep):

if deep == 0:

return

print(program)

global grammar

single = []

double = []

triple = []

quadro = []

penta = []

#print(program)

for i in range(len(program)):

for rule in grammar:

for alternative in rule[1]:

if (len(alternative) == 1):

if (get\_key(alternative[0]) == program[i]):

single.append([i, rule[0]])

if (len(alternative) == 2) and (i < len(program) - 1):

if (get\_key(alternative[1]) == program[i + 1]):

double.append([i, rule[0]])

if (len(alternative) == 3) and (i < len(program) - 2):

if (get\_key(alternative[2]) == program[i + 2]):

triple.append([i, rule[0]])

if (len(alternative) == 4) and (i < len(program) - 3):

if (get\_key(alternative[3]) == program[i + 3]):

quadro.append([i, rule[0]])

if (len(alternative) == 5) and (i < len(program) - 4):

if (get\_key(alternative[4]) == program[i + 4]):

penta.append([i, rule[0]])

for s in penta:

thread = Thread(target = replace, args = (rep(program, s[0], s[1], 5), deep - 1))

thread.start()

for s in quadro:

thread = Thread(target = replace, args = (rep(program, s[0], s[1], 4), deep - 1))

thread.start()

for s in triple:

thread = Thread(target = replace, args = (rep(program, s[0], s[1], 3), deep - 1))

thread.start()

for s in double:

thread = Thread(target = replace, args = (rep(program, s[0], s[1], 2), deep - 1))

thread.start()

for s in single:

thread = Thread(target = replace, args = (rep(program, s[0], s[1], 1), deep - 1))

thread.start()

Процедура работает неприемлемо долго даже для небольшой программы, значит такой анализатор тоже не подходит. Попробуем построить простой нисходящий синтаксический анализатор.

# **Рекурсивная процедура нисходящего анализа**

def threadeded\_function(program, deep):

print(program)

global stats

global leng

if deep == 0:

return

while True:

for sentence in program:

for rule in grammar:

if rule[0] == sentence:

for alternative in rule[1]:

if (len(rule[1]) == 1):

program = insert(program, alternative, program.index(sentence))

else:

thread = Thread(target = threadeded\_function, args = (insert(program, alternative, program.index(sentence)), deep - 1))

thread.start()

if (len(rule[1]) > 1):

return

Результат запуска аналогичен предыдущему, значит такой анализатор тоже не подходит.

# **Вывод**

Таким образом, синтез оптимального синтаксического анализатора для произвольной грамматики не является такой простой задачей, как это может показаться на первый взгляд.