**源程序清单**

**#my\_lex.py**

import ply.lex as lex

#from my\_yacc\_debug import TextBrowser

import my\_yacc\_debug

tokens = ('NUM', 'ID', 'FLOAT', 'CHARACTER', 'RELOP', 'MULOP', 'ASSIGNOP', 'TOTO', 'ARRAY', 'BEGIN', 'CASE',

'DO', 'TO','DOWNTO', 'ELSE', 'FOR', 'FUNCTION', 'END', 'PROGRAM', 'CONST', 'OF', 'OR', 'NOT', 'PROCEDURE', 'RECORD',

'REPEAT', 'IF', 'THEN', 'TYPE', 'UNTIL', 'VAR', 'WHILE', 'CHAR', 'REAL', 'INTEGER', 'BOOLEAN','READ','WRITE','TRUE','FALSE')

# literals = [ '+','-', '(', ')', ',', ';', '=', '[', ']', ':', '.', ]

#keyword\_list

Key\_list=['ARRAY', 'BEGIN', 'CASE','DO', 'TO','DOWNTO', 'ELSE', 'FOR', 'FUNCTION', 'END', 'PROGRAM', 'CONST', 'OF', 'OR', 'NOT', 'PROCEDURE', 'RECORD',

'REPEAT', 'IF', 'THEN', 'TYPE', 'UNTIL', 'VAR', 'WHILE', 'CHAR', 'REAL', 'INTEGER', 'BOOLEAN','READ','WRITE','FALSE','TRUE']

#idetifier

def t\_MULOP(t):

r"\\*|/| div | mod | and "

##t.lexpos=find\_column(input\_data,t)

return t

def t\_ID(t):

r'[a-z][a-z0-9]\*'

# t.lexpos=find\_column(input\_data,t)

if t.value.upper() in Key\_list:

t.type =Key\_list[Key\_list.index(t.value.upper())]

return t

def t\_FLOAT(t):

r"\d+\.\d+e[+-]{0,1}\d+|\d+\.\d+"

##t.lexpos=find\_column(input\_data,t)

t.value=float(t.value)

return t

def t\_NUM(t):

r'\d+e[+-]{0,1}\d+|\d+'

##t.lexpos=find\_column(input\_data,t)

t.value = int(t.value)

return t

def t\_CHARACTER(t):

r'\'[\s\S]\''

##t.lexpos=find\_column(input\_data,t)

t.value=t.value[1]

return t

def t\_RELOP(t):

r"<>|<=|<|>=|>"

##t.lexpos=find\_column(input\_data,t)

return t

def t\_ASSIGNOP(t):

r':='

##t.lexpos=find\_column(input\_data,t)

return t

def t\_TOTO(t):

r"\.\."

##t.lexpos=find\_column(input\_data,t)

return t

def t\_newline(t):

r'\n+'

##t.lexpos=find\_column(input\_data,t)

t.lexer.lineno += len(t.value)

def t\_annotation(t):

r"{[\s\S]\*}"

pass

t\_ignore=' |\t'

def t\_error(t):

##t.lexpos=find\_column(input\_data,t)

print("(",t.lineno,",",t.lexpos,")","Error:Illegal expression:",t.value[0])

my\_yacc\_debug.TextBrowser.append("Error:Illegal expression: "+str(t.value[0])+" at line "+str(t.lineno))

t.lexer.skip(1)

literals = [ '+','=','-',',',';','.',':','[',']','(',')' ]

# def find\_column(input,token):

# last\_cr = input.rfind('\n',0,token.lexpos)

# if last\_cr < 0:

# last\_cr = -1

# column = (token.lexpos - last\_cr)

# return column

# with open("test.txt") as f:

# input\_data = f.read()

# lexer=lex.lex()

**#my\_yacc.py**

#-\*- coding: UTF-8 -\*-

from \_\_future\_\_ import print\_function

import ply.yacc as yacc

from my\_lex import \*

from support\_functions import \*

isRecord = False

id\_cnt = 0

def my\_print(str):

TextBrowser.append(str)

def p\_program(p):

'''program : program\_head program\_body '.' '''

fp.close()

my\_print("completed")

#action

def p\_program\_head(p):

'''program\_head : PROGRAM ID '(' identifier\_list ')' ';' '''

#action

print("#include<stdio.h>" ,file=fp)

global id\_cnt

id\_cnt = 0

def p\_program\_body(p):

'''program\_body : const\_declarations type\_declarations var\_declarations subprogram\_declarations O compound\_statement'''

#action

print(p[6].str + "}", file=fp)

def p\_O(p):

'''O : empty '''

#action

print("main()\n{", file=fp)

def p\_identifier\_list(p):

'''identifier\_list : identifier\_list ',' ID

| ID '''

#action

item={}

global isRecord

global id\_cnt

p[0] = Attribute()

if len(p) == 4:

id\_cnt += 1

p[0].str = p[1].str + "," + p[3]

item["name"] = p[3]

item["type"] = ""

item["F\_type"] = False

item["F\_const"] = False

item["declare\_line"] = p.lineno(3)

item["extend"] = []

item["var"] = False

isDefined = symboltable.find(p[3], mode=1)

if not isRecord and not isDefined:

symboltable.insert(item)

elif isRecord and not isDefined:

p[0].extend = []

p[0].extend.append(item)

else:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].str = p[1]

id\_cnt += 1

item["name"] = p[1]

item["type"] = ""

item["F\_type"] = False

item["F\_const"] = False

item["declare\_line"] = p.lineno(1)

item["extend"] = []

item["var"] = False

isDefined = symboltable.find(p[1], mode=1)

if not isRecord and not isDefined:

symboltable.insert(item)

elif isRecord and not isDefined:

p[0].extend = []

p[0].extend.append(item)

else:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + "has already been defined")

p[0].type = "type\_error"

def p\_const\_declarations(p):

'''const\_declarations : CONST const\_declaration ';'

| empty '''

#action

p[0] = Attribute()

if len(p) == 4:

print(";", file=fp)

if p[2].type == "type\_error":

p[0].type = "type\_error"

# def p\_P(p):

# ''' P : empty '''

# #action

# print("const ",end = "",file=fp)

def p\_const\_declaration(p):

'''const\_declaration : const\_declaration ';' Q ID '=' const\_variable

| ID '=' const\_variable'''

#action

p[0] = Attribute()

item={}

if len(p)==7:

print("const", p[6].type, p[4], " = ", p[6].str, end="", file=fp)

if p[6].type != "type\_error" and p[1].type != "type\_error":

if not symboltable.find(p[4]):

item["name"] = p[4]

item["type"] = p[6].type

item["F\_type"] = False

item["F\_const"] = True

item["declare\_line"] = p.lineno(4)

item["extend"] = p[6].value

item["var"] = False

symboltable.insert(item)

else:

print(p[4], " at line", p.lineno(4), "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

elif len(p) == 4:

print("const", p[3].type, p[1], " = ", p[3].str, end="", file=fp)

if p[3].type != "type\_error":

if not symboltable.find(p[1]):

item["name"] = p[1]

item["type"] = p[3].type

item["F\_type"] = False

item["F\_const"] = True

item["declare\_line"] = p.lineno(1)

item["extend"] = p[3].value

item["var"] = False

symboltable.insert(item)

else:

my\_print(str(p[1]) + " at line" + str(p.lineno(4)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_Q(p):

''' Q : empty '''

#action

print(";", file=fp)

##modified

def p\_const\_variable\_1(p):

'''const\_variable : '+' ID '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[2])

if not index:

p[0].type = "type\_error"

my\_print(str(p[2]) + " at line" + str(p.lineno(2)) + "is not a const variable")

elif symboltable.table[index]["F\_const"] == False:

my\_print("Bad const variable declared at line" + str(p.lineno(2)) + str(p[2]) + "is not a const variable")

p[0].type = "type\_error"

else:

p[0].type = symboltable.table[index]["type"]

p[0].str = "+" + p[2]

p[0].value = symboltable.table[index]["extend"]

p[0].lineno = p.lineno(2)

#modified

def p\_const\_variable\_2(p):

'''const\_variable : '-' ID '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[2])

if not index:

p[0].type = "type\_error"

my\_print(str(p[2]) + " at line" + str(p.lineno(2)) + "is not a const variable")

elif symboltable.table[index]["F\_const"] == False:

my\_print("Bad const variable declared at line" + str(p.lineno(2)) + str(p[2]) + "is not a const variable")

p[0].type = "type\_error"

else:

p[0].type = symboltable.table[index]["type"]

p[0].str = "-" + p[2]

p[0].value = symboltable.table[index]["extend"]

p[0].lineno = p.lineno(2)

def p\_const\_variable\_3(p):

'''const\_variable : '+' NUM '''

#action

p[0] = Attribute()

p[0].value = p[2]

p[0].str = "+" + str(p[2])

p[0].type = "int"

p[0].lineno = p.lineno(2)

def p\_const\_variable\_4(p):

'''const\_variable : '-' NUM '''

#action

p[0] = Attribute()

p[0].value = -p[2]

p[0].str = "-" + str(p[2])

p[0].type = "int"

p[0].lineno = p.lineno(2)

def p\_const\_variable\_5(p):

'''const\_variable : NUM '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(p[1])

p[0].type = "int"

p[0].lineno = p.lineno(1)

def p\_const\_variable\_6(p):

'''const\_variable : CHARACTER '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = "\"" + p[1] + "\""

p[0].type = "char"

p[0].lineno = p.lineno(1)

def p\_const\_variable\_7(p):

'''const\_variable : '+' FLOAT '''

#action

p[0] = Attribute()

p[0].value = p[2]

p[0].str = "+" + str(p[2])

p[0].type = "float"

p[0].lineno = p.lineno(2)

def p\_const\_variable\_8(p):

'''const\_variable : '-' FLOAT '''

#action

p[0] = Attribute()

p[0].value = -p[2]

p[0].str = "-" + str(p[2])

p[0].type = "float"

p[0].lineno = p.lineno(2)

def p\_const\_variable\_9(p):

'''const\_variable : FLOAT '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(p[1])

p[0].type = "float"

p[0].lineno = p.lineno(1)

#modified

def p\_const\_variable\_10(p):

'''const\_variable : ID '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[1])

if index == None:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + "is not a const variable")

p[0].type = "type\_error"

else:

if symboltable.table[index]["F\_const"] == False:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + "is not a const variable")

else:

p[0].type = symboltable.table[index]["type"]

p[0].str = p[1]

p[0].value = symboltable.table[index]["extend"]

p[0].lineno = p.lineno(1)

def p\_type\_declarations(p):

'''type\_declarations : TYPE R type\_declaration ';'

| empty '''

#action

p[0] = Attribute()

if len(p) == 5:

if p[3].type == "type\_error":

p[0].type = "type\_error"

def p\_R(p):

''' R : empty '''

#action

#print("typedef ", end="", file=fp)

def p\_type\_declaration(p):

'''type\_declaration : type\_declaration ';' S ID '=' type

| ID '=' type '''

#action

p[0] = Attribute()

item = {}

if len(p) == 7:

if p[6].type[0:5] != "array":

print("typedef", p[6].type,p[4] + ";", file=fp)

else:

print("typedef", p[6].type[5:], p[4], end="", file=fp)

for i in range(1, len(p[6].extend)):

print("[", p[6].extend[i][1], "]"+";", file=fp)

if p[6].type != "type\_error":

if not symboltable.find(p[4]):

item["name"] = p[4]

item["type"] = p[6].type

item["F\_type"] = True

item["F\_const"] = False

item["declare\_line"] = p.lineno(3)

item["extend"] = p[6].extend

item["var"] = False

symboltable.insert(item)

else:

my\_print(str(p[4]) + " at line" + str(p.lineno(4)) + "has been declared")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

elif len(p) == 4:

if p[3].type[0:5] != "array":

print("typedef", p[3].str," ", p[1]+";", file=fp)

else:

print("typedef", p[3].str[5:]," ", p[1], end="", file=fp)

for i in range(1, len(p[3].extend)):

print("[", p[3].extend[i][1], "]" + ";", file=fp)

if p[3].type != "type\_error":

if not symboltable.find(p[1]):

item["name"] = p[1]

item["type"] = p[3].type

item["F\_type"] = True

item["F\_const"] = False

item["declare\_line"] = p.lineno(1)

item["extend"] = p[3].extend

item["var"] = False

symboltable.insert(item)

else:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + "has been declared")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_S(p):

''' S : empty '''

#action

#print(",", end="", file=fp)

def p\_type1(p):

'''type : standard\_type

| RECORD L record\_body END

| ARRAY '[' periods ']' OF type'''

#action

p[0] = Attribute()

if len(p) == 2:

p[0].type = p[1].type

p[0].str = p[1].type

elif len(p) == 5:

p[0].str = "struct\n{\n" + p[3].str + "\n}"

if p[3].type != "type\_error":

p[0].type = "record"

p[0].extend = p[3].extend

global isRecord

isRecord = False

else:

p[0].type = "type\_error"

elif len(p) == 7:

if p[3].type != "type\_error" and p[6].type != "type\_error":

p[0].type = "array" + p[6].type

p[0].str = p[6].str

p[0].extend = [p[6].type] + p[3].extend

else:

p[0].type = "type\_error"

#modified

def p\_type2(p):

'''type : ID '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[1])

if not index:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + " is not a type")

p[0].type = "type\_error"

else:

flag1 = symboltable.find(p[1])

flag2 = symboltable.table[index]["F\_type"]

if flag1 and flag2:

p[0].name = p[1]

p[0].str = p[1]

p[0].type = p[1]

p[0].extend = ""

else:

p[0].type = "type\_error"

if not flag2:

my\_print("Bad type declaration at line" + str(p.lineno(1)) + str(p[1]) + " is not a type")

if not flag1:

my\_print(str(p[1]) + " at line" + str(p.lineno(1)) + "is not declared")

def p\_L(p):

''' L : empty '''

global isRecord

p[0] = Attribute()

p[0].extend = []

isRecord = True

# print(",", end="", file=fp)

def p\_standard\_type1(p):

'''standard\_type : INTEGER '''

p[0] = Attribute()

p[0].str = "int"

p[0].type = "int"

#action

def p\_standard\_type2(p):

'''standard\_type : REAL '''

#action

p[0] = Attribute()

p[0].str = "float"

p[0].type = "float"

def p\_standard\_type3(p):

'''standard\_type : BOOLEAN '''

#action

p[0] = Attribute()

p[0].str = "int"

p[0].type = "boolean"

def p\_standard\_type4(p):

'''standard\_type : CHAR '''

#action

p[0] = Attribute()

p[0].str = "char"

p[0].type = "char"

def p\_record\_body1(p):

'''record\_body : var\_declaration '''

#action

p[0] = Attribute()

p[0].str = p[1].str

if p[1].type != "type\_error":

p[0].extend = p[1].extend

else:

p[0].type = "type\_error"

def p\_record\_body2(p):

'''record\_body : empty '''

#action

p[0] = Attribute()

p[0].str = ""

p[0].extend = []

def p\_periods(p):

'''periods : periods ',' period

| period '''

#action

p[0] = Attribute()

if len(p) == 4:

if p[1].type != "type\_error" and p[3].type != "type\_error":

p[0].extend = p[1].extend + p[3].extend

else:

p[0].type = "type\_error"

else:

if p[1].type != "type\_error":

p[0].extend = p[1].extend

else:

p[0].type = "type\_error"

def p\_period(p):

'''period : const\_variable TOTO const\_variable'''

#action

p[0] = Attribute()

if p[1].type == "int" and p[3].type == "int" and p[1].value < p[3].value:

p[0].extend = [((p[1].value, p[3].value - p[1].value + 1))]

else:

my\_print("Bad array index declaration at line" + str( p.lineno(2)))

p[0].type = "type\_error"

#gaile

def p\_var\_declarations(p):

'''var\_declarations : VAR var\_declaration

| empty '''

#action

p[0] = Attribute()

if len(p) == 3:

print(p[2].str, file=fp)

if p[2].type == "type\_error":

p[0].type = "type\_error"

#gaile

def p\_var\_declaration(p):

'''var\_declaration : var\_declaration identifier\_list ':' type ';'

| identifier\_list ':' type ';' '''

#action

p[0] = Attribute()

global isRecord

global id\_cnt

if len(p) == 6 and not isRecord:

if p[4].type == "type\_error" or p[1].type == "type\_error":

p[0].type = "type\_error"

elif p[4].type[0:5] == "array":

#符号表操作

idx = symboltable.Top - 1

for i in range(0,id\_cnt):

symboltable.table[idx - i]["type"] = p[4].type

symboltable.table[idx - i]["extend"] = p[4].extend

#代码生成

p[0].str = p[1].str + "\n" + p[4].extend[0] + " "

temp = p[2].str.split(',', 1)

for Str in temp:

Temp = ""

for i in range(1, len(p[4].extend)):

Temp = Temp + "[" + str(p[4].extend[i][1]) + "]"

p[0].str = p[0].str + Str + Temp + ","

p[0].str = p[0].str[:-1] + ";"

else:

#符号表操作

idx = symboltable.Top - 1

for i in range(0, id\_cnt):

symboltable.table[idx-i]["type"] = p[4].type

symboltable.table[idx-i]["extend"] = p[4].extend

p[0].str = p[1].str + "\n" + p[4].str + " " + p[2].str + ";"

elif len(p) == 6 and isRecord:

list = p[2].extend

length = len(list)

for i in range(length - id\_cnt, length):

list[i]["type"] = p[4].type

list[i]["extend"] = p[4].extend

p[0].extend = p[1].extend + list

if p[4].type == "type\_error" or p[1].type == "type\_error":

p[0].type = "type\_error"

elif(p[4].type[0:5] == "array"):

p[0].str = p[1].str + "\n" + p[4].extend[0] + " "

temp = p[2].str.split(',',1)

for Str in temp:

Temp = ""

for i in range(1,len(p[4].extend)):

Temp = Temp + "[" + str(p[4].extend[i][1])+"]"

p[0].str = p[0].str + Str + Temp + ","

p[0].str = p[0].str[:-1] + ";"

else:

p[0].str = p[1].str + "\n" + p[4].str + " " + p[2].str + ";"

elif len(p) == 5 and not isRecord:

if p[3].type == "type\_error" or p[1].type == "type\_error":

p[0].type = "type\_error"

elif(p[3].type[0:5] == "array"):

#符号表操作

idx = symboltable.Top - 1

for i in range(0,id\_cnt):

symboltable.table[idx - i]["type"] = p[3].type

symboltable.table[idx - i]["extend"] = p[3].extend

p[0].str = p[3].extend[0] + " "

temp = p[1].str.split(',',1)

for Str in temp:

Temp = ""

for i in range(1,len(p[3].extend)):

Temp = Temp + "[" + str(p[3].extend[i][1]) + "]"

p[0].str = p[0].str + Str + Temp + ","

p[0].str = p[0].str[:-1] + ";"

else:

idx = symboltable.Top - 1

for i in range(0, id\_cnt):

symboltable.table[idx - i]["type"] = p[3].type

symboltable.table[idx - i]["extend"] = p[3].extend

p[0].str = p[3].str + " " + p[1].str + ";"

elif len(p) == 5 and isRecord:

list = p[1].extend

length = len(list)

for i in range(length - id\_cnt, length):

list[i]["type"] = p[3].type

list[i]["extend"] = p[3].extend

p[0].extend = list

if p[3].type == "type\_error" or p[1].type == "type\_error":

p[0].type = "type\_error"

elif p[3].type[0:5] == "array":

p[0].str = p[3].extend[0] + " "

temp = p[1].str.split(',',1)

for Str in temp:

Temp = ""

for i in range(1, len(p[3].extend)):

Temp = Temp + "[" + str(p[3].extend[i][1]) + "]"

p[0].str = p[0].str + Str + Temp + ","

p[0].str = p[0].str[:-1] + ";"

else:

p[0].str = p[3].str + " " + p[1].str + ";"

id\_cnt = 0

def p\_subprogram\_declarations(p):

'''subprogram\_declarations : subprogram\_declarations subprogram\_declaration ';'

| empty '''

#action

p[0] = Attribute()

if len(p) == 3:

symboltable.locate()

if p[1].type == "type\_error" or p[2].type == "type\_error":

p[0].type = "type\_error"

def p\_subprogram\_declaration(p):

'''subprogram\_declaration : subprogram\_head subprogram\_body'''

#action

p[0] = Attribute()

if p[1].type == "type\_error" or p[2].type == "type\_error":

p[0].type = "type\_error"

# print(symboltable.table)

symboltable.relocate()

# print(symboltable.table)

def p\_subprogram\_body(p):

'''subprogram\_body : T const\_declarations type\_declarations var\_declarations compound\_statement LC\_U'''

#action

p[0] = Attribute()

if p[-1].type != "void" :

print("return", p[-1].name + ";",file=fp)

print("}", file=fp)

if p[2].type == "type\_error" or p[3].type == "type\_error" or p[4].type == "type\_error" or p[5].type == "type\_error":

p[0].type = "type\_error"

def p\_T(p):

''' T : empty '''

# action

print("{", file=fp)

#函数返回参数声明

if(p[-1].type != 'void'):

print(p[-1].type, p[-1].name + ";", file=fp)

def p\_LC\_U(p):

''' LC\_U : empty '''

# action

print(p[-1].str, end="",file=fp)

#函数返回参数声明

def p\_subprogram\_head1(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' standard\_type ';' '''

#action

p[0] = Attribute()

hasDefined = symboltable.find(p[3], mode=1)

if not hasDefined and p[4].type != "type\_error":

fun\_name = p[3] + "\_function"

print(p[6].str, fun\_name, p[4].str, file=fp)

p[0].name = p[3]

p[0].type = p[6].type

item = symboltable.domain\_stack[1]

symboltable.table[item]["name"] = p[3]

symboltable.table[item]["type"] = "function"

symboltable.table[item]["F\_type"] = False

symboltable.table[item]["F\_const"] = False

symboltable.table[item]["declare\_line"] = p.lineno(3)

symboltable.table[item]["extend"] = [p[6].type] + p[4].parameter\_list

symboltable.table[item]["var\_list"]=["false"] + p[4].var\_list

elif hasDefined:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_X(p):

'''X : empty'''

item = {}

item["name"] = ""

item["type"] = "function"

item["F\_type"] = False

item["F\_const"] = False

item["declare\_line"] = 0

item["var"] = False

symboltable.locate()

symboltable.insert(item)

def p\_subprogram\_head2(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' array\_type ';' '''

#action

p[0] = Attribute()

hasDefined = symboltable.find(p[3], mode=1)

if not hasDefined and p[4].type != "type\_error" and p[6].type != "type\_error":

fun\_name = p[3] + "\_function"

print(p[6].str[5:] + '\*', fun\_name, p[4].str, file=fp)

p[0].name = p[3]

p[0].type = p[6].str

item = symboltable.domain\_stack[1]

symboltable.table[item]["name"] = p[3]

symboltable.table[item]["type"] = "function"

symboltable.table[item]["F\_type"] = False

symboltable.table[item]["F\_const"] = False

symboltable.table[item]["declare\_line"] = p.lineno(3)

symboltable.table[item]["extend"] = [p[6].parameter\_list] + p[4].parameter\_list

symboltable.table[item]["var"] = False

symboltable.locate()

symboltable.insert(item)

elif hasDefined:

my\_print(str(p[2]) + " at line" + str(p.lineno(2)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_subprogram\_head3(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' ID ';' '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[6])

if not index:

my\_print(str(p[6]) + " at line" + str(p.lineno(6)) + " is not a type")

p[0].type = "type\_error"

else:

if symboltable.table[index]["F\_type"] == False:

my\_print("Return type error at line" + str(p.lineno(6)) + str(p[6]) + " is not a type")

p[0].type = "type\_error"

else:

hasDefined = symboltable.find(p[3], mode=1)

if not hasDefined and p[4].type == "type\_error" :

fun\_name = p[3] + "\_function";

print(p[6], fun\_name, p[4].str, file=fp)

p[0].name = p[3]

p[0].type = p[6]

item = symboltable.domain\_stack[1]

symboltable.table[item]["name"] = p[3]

symboltable.table[item]["type"] = "function"

symboltable.table[item]["F\_type"] = False

symboltable.table[item]["F\_const"] = False

symboltable.table[item]["declare\_line"] = p.lineno(3)

symboltable.table[item]["extend"] = [p[6]] + p[4].parameter\_list

symboltable.table[item]["var\_list"] = ["false"] + p[4].var\_list

elif hasDefined:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_subprogram\_head4(p):

'''subprogram\_head : X PROCEDURE ID formal\_parameter ';' '''

#action

p[0] = Attribute()

hasDefined = symboltable.find(p[3], mode=1)

if not hasDefined and p[4].type != "type\_error":

fun\_name = p[3] + "\_function";

print("void", fun\_name, p[4].str, file=fp)

p[0].name = p[3]

p[0].type = "void"

item = symboltable.domain\_stack[1]

symboltable.table[item]["name"] = p[3]

symboltable.table[item]["type"] = "procedure"

symboltable.table[item]["F\_type"] = False

symboltable.table[item]["F\_const"] = False

symboltable.table[item]["declare\_line"] = p.lineno(3)

symboltable.table[item]["extend"] = p[4].parameter\_list

symboltable.table[item]["var\_list"] = p[4].var\_list

elif hasDefined:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + "has already been defined")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_formal\_parameter(p):

'''formal\_parameter : '(' parameter\_lists ')'

| empty'''

#action

p[0] = Attribute()

if len(p) == 4:

p[0].str = "(" + p[2].str + ")"

p[0].parameter\_list = p[2].parameter\_list

p[0].var\_list = p[2].var\_list

if p[2].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].str = "()"

p[0].parameter\_list = []

def p\_parameter\_lists(p):

'''parameter\_lists : parameter\_lists ';' parameter\_list

| parameter\_list '''

#action

p[0] = Attribute()

p[0].parameter\_list = []

if len(p) == 4:

p[0].str = p[1].str + "," + p[3].str

p[0].parameter\_list = p[1].parameter\_list + p[3].parameter\_list

p[0].var\_list = p[1].var\_list + p[3].var\_list

if p[1].type == "type\_error" and p[3].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].str = p[1].str

p[0].parameter\_list = p[1].parameter\_list[:]

p[0].var\_list = p[1].var\_list[:]

p[0].type = p[1].type

def p\_parameter\_list1(p):

'''parameter\_list : var\_parameter '''

#action

p[0] = Attribute()

p[0].str = p[1].str

p[0].parameter\_list = p[1].parameter\_list

p[0].var\_list = p[1].var\_list[:]

if p[1].type == "type\_error":

p[0].type = "type\_error"

def p\_parameter\_list2(p):

'''parameter\_list : value\_parameter '''

#action

p[0] = Attribute()

temp = p[1].str.split(',')

p[0].str = ""

for Str in temp:

if "array" in p[1].type:

p[0].str += p[1].type[5:] + " " + Str + "[],"

else:

p[0].str += p[1].type + " " + Str + ","

p[0].str = p[0].str[:-1]

p[0].parameter\_list = p[1].parameter\_list[:]

p[0].var\_list = p[1].var\_list[:]

if p[1].type == "type\_error":

p[0].type = "type\_error"

def p\_(p):

'''var\_parameter : VAR value\_parameter'''

#action

p[0] = Attribute()

p[0].str = ''

temp = p[2].str.split(',',1)

for Str in temp:

if "array" in p[2].type:

p[0].str = p[0].str + p[2].type[5:] + " " + Str + "[],"

else:

p[0].str = p[0].str + p[2].type + "\* " + Str + ","

symboltable.get(Str)["var"] = True

p[0].str = p[0].str[:-1]

p[0].parameter\_list = p[2].parameter\_list[:]

p[0].var\_list = p[2].var\_list[:]

length = len(p[0].var\_list)

for i in range(0,length):

p[0].var\_list[i] = "true"

if p[2].type == "type\_error":

p[0].type = "type\_error"

def p\_value\_parameter1(p):

'''value\_parameter : identifier\_list ':' standard\_type'''

#action

p[0] = Attribute()

global id\_cnt

if p[1].type != "type\_error":

p[0].str = p[1].str

p[0].type = p[3].type

idx = symboltable.Top

for idx in range (idx - id\_cnt, idx):

symboltable.table[idx]["type"] = p[3].type

i = id\_cnt

p[0].parameter\_list = []

while i != 0:

#此处应有类型检查

p[0].parameter\_list.append(p[3].type)

p[0].var\_list.append("false")

i -= 1

id\_cnt = 0

else:

p[0].type = "type\_error"

def p\_value\_parameter2(p):

'''value\_parameter : identifier\_list ':' array\_type'''

#action

p[0] = Attribute()

if p[1].type != "type\_error" and p[3].type != "type\_error":

p[0].str = p[1].str

p[0].type = "array" + p[3].type

global id\_cnt

i = id\_cnt

p[0].parameter\_list = []

while i != 0:

#此处应有类型检查

p[0].parameter\_list.append(p[3].parameter\_list)

p[0].var\_list.append(p[3].var\_list)

i -= 1

for index in range (symboltable.Top-id\_cnt, symboltable.Top):

symboltable.table[index]["type"] = p[0].type

symboltable.table[index]["extend"] = []

symboltable.table[index]["extend"].append(p[3].type)

symboltable.table[index]["extend"].append(None)

id\_cnt = 0

else:

p[0].type = "type\_error"

def p\_value\_parameter3(p):

'''value\_parameter : identifier\_list ':' ID '''

#action

p[0] = Attribute()

index = symboltable.find\_item(p[3])

global id\_cnt

if not index:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + " is not a type")

p[0].type = "type\_error"

id\_cnt = 0

else:

isType = symboltable.table[index]["F\_type"]

if isType and p[1].type != "type\_error":

p[0].str = p[1].str

p[0].type = p[3]

i = id\_cnt

p[0].parameter\_list = []

while i != 0:

#此处应有类型检查

p[0].parameter\_list.append(p[3])

p[0].var\_list.append("false")

i -= 1

idx = symboltable.Top

for i in range (idx-id\_cnt,idx):

symboltable.table[i]["type"] = p[3]

id\_cnt = 0

elif not isType:

my\_print(str(p[3]) + " at line" + str(p.lineno(3)) + " is not a type")

p[0].type = "type\_error"

else:

p[0].type = "type\_error"

def p\_array\_type(p):

'''array\_type : ARRAY OF standard\_type'''

#action

p[0] = Attribute()

p[0].str = "array" + p[3].str

p[0].type = p[3].type

p[0].parameter\_list = "array" + p[3].type

def p\_compund\_statement(p):

'''compound\_statement : BEGIN statement\_list END U'''

#action

p[0] = Attribute()

if p[2].type == "type\_error":

p[0].type = "type\_error"

p[0].str = p[2].str;

def p\_U(p):

''' U : empty '''

#action

#print(p[-2].str, end="", file=fp)

def p\_statement\_list(p):

'''statement\_list : statement\_list ';' statement

| statement '''

#action

p[0] = Attribute()

if len(p) == 4:

p[0].str = p[1].str + p[3].str

if p[1].type == "type\_error" or p[3].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].str = p[1].str

if p[1].type == "type\_error":

p[0].type = "type\_error"

def p\_statement1(p):

'''statement : variable ASSIGNOP expression'''

#action

p[0] = Attribute()

p[0].str = p[1].str + "=" + p[3].str + ";\n"

if p[1].type != "function":

if p[1].type != p[3].type:

if p[1].type == "float" and p[3].type == "int":

pass

elif p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print(p[1].name + " at line" + str(p[1].lineno) + "type does not match")

else:

p[0].type = "type\_error"

else:

list = symboltable.get(p[1].name)

if not list:

my\_print(p[1].name + " at line" + str(p.lineno(2)) + "does not have an item")

p[0].type = "type\_error"

else:

if list["extend"][0] != p[3].type:

if p[3].type == "int" and list["extend"][0] == "float":

pass

elif p[3].type != "type\_error":

my\_print(p[1].name + " at line" + str(p.lineno(2)) + "type does not match")

else:

p[0].type = "type\_error"

def p\_statement2(p):

'''statement : call\_procedure\_statement'''

#action

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

def p\_statement3(p):

'''statement : compound\_statement'''

#action

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

def p\_statement4(p):

'''statement : IF expression THEN statement else\_part'''

#action

p[0] = Attribute()

p[0].str = "if(" + p[2].str + ")\n{\n" + p[4].str + "}" + p[5].str

if p[2].type != "boolean":

my\_print(p[2].str + " at line " + str(p[2].lineno) + " is not a boolean variable")

p[0].type = "type\_error"

elif p[4].type == "type\_error" or p[5].type == "type\_error":

p[0].type = "type\_error"

def p\_statement5(p):

'''statement : CASE expression OF case\_body END'''

#action

p[0] = Attribute()

p[0].str = "\nswitch(" + p[2].str + ")\n{\n" + p[4].str + "}\n"

if p[2].type == "type\_error" or p[4].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].type = p[4].type

def p\_statement6(p):

'''statement : WHILE expression DO statement'''

#action

p[0] = Attribute()

p[0].str = "while(" + p[2].str + ")\n{\n" + p[4].str + '}\n'

if p[2].type != "boolean":

my\_print(p[2].str + " at line " + str(p[2].lineno) + " is not a boolean variable")

p[0].type = "type\_error"

else:

p[0].type = p[4].type

def p\_statement7(p):

'''statement : REPEAT statement\_list UNTIL expression'''

#action

p[0] = Attribute()

p[0].str = "do{\n" + p[2].str + "}\nwhile(" + "!(" + p[4].str + ")" + ")\n"

if p[4].type != "boolean":

my\_print(p[4].str + " at line " + str(p[4].lineno) + " is not a boolean type")

p[0].type = "type\_error"

else:

p[0].type = p[2].type

def p\_statement8(p):

'''statement : FOR ID ASSIGNOP expression updown expression DO statement'''

#action

p[0] = Attribute()

if p[5].str == "to":

p[0].str = "for(" + p[2] + "=" + p[4].str + ";" + p[2] + "<=" + p[6].str + ";" + p[2] + "++)\n{\n" + p[8].str + "}\n"

elif p[5].str == "downto":

p[0].str = "for(" + p[2] + "=" + p[4].str + ";" + p[2] + ">=" + p[6].str + ";" + p[2] + "--)\n{\n" + p[8].str + "}\n"

if not symboltable.find(p[2]):

my\_print(str(p[2]) + "is not declared at line " + str(p.lineno(2)))

p[0].type = "type\_error"

if symboltable.get\_type(p[2]) != p[4].type and p[6].type != p[4].type:

my\_print("expressions'types conflict at line " + str(p.lineno(1)))

p[0].type = "type\_error"

else:

p[0].type = p[8].type

def p\_statement9(p):

'''statement : empty'''

#action

p[0] = Attribute()

p[0].str = ""

def p\_variable(p):

'''variable : ID id\_varparts'''

#action

p[0] = Attribute()

p[0].lineno = p.lineno(1)

if not symboltable.find(p[1]):

my\_print(str(p[1]) + " not defined at line " + str(p.lineno(1)))

p[0].type = "type\_error"

else:

if symboltable.get(p[1])["var"]:

p[0].str = "\*" + p[1] + p[2].str

else:

p[0].str = p[1] + p[2].str

p[0].type = p[2].type

p[0].name = p[1]

p[0].parameter\_list = []

p[0].parameter\_list.append((p[0].str, p[0].type))

p[0].var\_list = []

p[0].var\_list.append("False")

def p\_id\_varparts(p):

'''id\_varparts : id\_varparts id\_varpart

| empty '''

#action

p[0] = Attribute()

if len(p) == 3:

p[0].dimension = p[2].dimension

idtype = symboltable.get\_real\_type(p[-1])

if idtype == None:

my\_print(str(p[-1]) + " at line " + str(p[2].lineno) + " does not have an additional part")

p[0].type = "type\_error"

elif idtype[0:5] != "array" and idtype != "record":

my\_print(str(p[-1]) + " at line " + str(p[2].lineno) + " does not have an additional part")

p[0].type = "type\_error"

else:

p[0].sublist = p[2].sublist[:]

p[0].str = p[1].str + p[2].str

if p[1].type == "type\_error" or p[2].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].type = p[2].type

else:

p[0].str = ""

if symboltable.find(p[-1]):

list = symboltable.get(p[-1])

if not list:

p[0].type = "type\_error"

else:

p[0].type = list["type"]

p[0].sublist = symboltable.get\_extend(p[-1])

if p[0].sublist == None:

p[0].sublist = []

else:

p[0].type = "type\_error"

def p\_id\_varpart(p):

'''id\_varpart : '[' expression\_list ']'

| '.' ID '''

#action

p[0] = Attribute()

if len(p) == 4:

p[0].dimension = p[-1].dimension + 1

STR = ""

if(len(p[-1].sublist)!=0):

STR = p[-1].sublist[1][0]

p[0].str = "[" + p[2].str + "-(" + str(STR) + ")" + "]"

if("array" in p[-1].type):

if p[2].type != "int":

my\_print("array index type error at line " + str(p.lineno(1)))

p[0].type = "type\_error"

if len(p[-1].sublist) - 1 > 1:

p[0].sublist = p[-1].sublist[:]

p[0].sublist.pop(1)

p[0].type = p[-1].type

elif len(p[-1].sublist) - 1 == 1:

p[0].sublist = p[-1].sublist[:]

p[0].sublist.pop(1)

p[0].type = p[-1].type[5:]

else:

p[0].type = "type\_error"

p[0].lineno = p.lineno(1)

else:

if symboltable.get\_real\_type(p[-1].type) == "record":

record\_list = symboltable.get(p[-1].type)["extend"]

sublist = find\_type(record\_list,p[2])

if not sublist:

my\_print("record part not defined at line " + str(p.lineno(2)))

p[0].type = "type\_error"

else:

p[0].type = sublist["type"]

p[0].sublist = sublist["extend"]

p[0].str = "." + p[2]

p[0].lineno = p.lineno(2)

def p\_else\_part(p):

'''else\_part : ELSE statement

| empty '''

#action

p[0] = Attribute()

if len(p) == 3:

p[0].str = "else\n{" + p[2].str + "\n}"

p[0].type = p[2].type

else:

p[0].str = ""

#add a ';'

def p\_case\_body(p):

'''case\_body : branch\_list ';'

| empty '''

#action

p[0] = Attribute()

if len(p) == 3:

p[0].str = p[1].str

p[0].type = p[1].type

else:

p[0].str = ""

p[0].type = ""

def p\_branch\_list(p):

'''branch\_list : branch\_list ';' LZ branch

| LX branch '''

#action

p[0] = Attribute()

if len(p) == 5:

p[0].str = p[1].str + p[4].str

if p[1].type == "type\_error" or p[4].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].str = p[2].str

p[0].type = p[2].type

def p\_LX(p):

''' LX : empty '''

p[0] = Attribute()

p[0].type = p[-2].type

def p\_LZ(p):

''' LZ : empty '''

p[0] = Attribute()

p[0].type = p[-4].type

def p\_branch(p):

'''branch : const\_list ':' statement'''

#action

p[0] = Attribute()

p[0].str = "case " + p[1].str + ":\n" + p[3].str + "break;\n"

# p[0].type = p[1].type

if p[1].type != "type\_error" and p[1].type == p[-1].type:

p[0].type = p[3].type

else:

my\_print("types conflict in casebody, at line " + str(p[1].lineno))

p[0].type = "type\_error"

def p\_const\_list(p):

'''const\_list : const\_list ',' const\_variable

| const\_variable '''

#action

p[0] = Attribute()

if len(p) == 4:

p[0].str = p[1].str + "," + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type == "type\_error":

p[0].type = "type\_error"

elif p[1].type != p[3].type:

my\_print("types conflict at line " + str(p[3].lineno))

p[0].type = "type\_error"

else:

p[0].type = p[1].type

else:

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

def p\_updown1(p):

'''updown : TO'''

#action

p[0] = Attribute()

p[0].str = "to"

p[0].lineno = p.lineno(1)

def p\_updown2(p):

'''updown : DOWNTO '''

#action

p[0] = Attribute()

p[0].str = "downto"

p[0].lineno = p.lineno(1)

def p\_call\_procedure\_statement1(p):

'''call\_procedure\_statement : ID'''

#action

p[0] = Attribute()

p[0].str = p[1] + "\_function();\n"

if not symboltable.find(p[1]):

my\_print(str(p[1]) + " at line " + str(p.lineno(1)) + "is not defined")

p[0].type = "type\_error"

elif symboltable.get\_type(p[1]) != "procedure":

my\_print(str(p[1]) + " at line " + str(p.lineno(1)) + " is not a procedure")

p[0].type = "type\_error"

elif(symboltable.get(p[1])["extend"]!=None):

my\_print(str(p[1]) + " at line " + str(p.lineno(1)) + " should have arguement " +str(symboltable.get(p[1])["extend"]))

p[0].type = "type\_error"

def p\_call\_procedure\_statement2(p):

'''call\_procedure\_statement : ID '(' expression\_list ')' '''

#action

p[0] = Attribute()

if(symboltable.find(p[1])):

#重组参数列表

str1 = p[3].str

item\_list = str1.split(",")

# if( symboltable.get(p[1])!=None):

Now\_type = symboltable.get(p[1])["type"]

var\_list = symboltable.get(p[1])["var\_list"]

if(Now\_type == "procedure"):

for i in range(0,len(item\_list)):

if(var\_list[i] == "true"):

if "array" not in symboltable.get\_type(item\_list[i]):

item\_list[i] = "&" + item\_list[i]

else:

item\_list[i] = item\_list[i]

p[3].str = ""

for i in item\_list:

p[3].str += "," + i

p[3].str = p[3].str[1:]

elif(Now\_type == "function"):

for i in range(0,len(item\_list)):

if(var\_list[i+1] == "true"):

if "array" not in symboltable.get\_type(item\_list[i]):

item\_list[i] = "&" + item\_list[i]

else:

item\_list[i] = item\_list[i]

p[3].str = ""

for i in range(0,len(item\_list)):

p[3].str += "," + item\_list[i]

p[3].str = p[3].str[1:]

p[0].str = p[1] + "\_function(" + p[3].str + ");\n"

if not symboltable.find(p[1]):

my\_print("function " + str(p[1]) + " not defined at line " + str(p.lineno(1)))

p[0].type = "type\_error"

elif symboltable.get\_type(p[1]) == "function":

if p[3].type != "".join(symboltable.get(p[1])['extend'][1:]):

my\_print("The parameters do not match at line " + str(p.lineno(1)))

p[0].type = "type\_error"

elif symboltable.get\_type(p[1]) == "procedure":

if p[3].type != "".join(symboltable.get(p[1])['extend']):

my\_print("The parameters do not match at line " + str(p.lineno(1)))

p[0].type = "type\_error"

else:

my\_print(str(p[1]) + " at line " + str(p.lineno(1)) + " is not a function or procedure")

p[0].type = "type\_error"

def p\_call\_procedure\_statement3(p):

'''call\_procedure\_statement : WRITE '(' expression\_list ')' '''

#action

p[0] = Attribute()

format\_str = ""

id\_str = ""

for i in range(0, len(p[3].parameter\_list)):

if p[3].parameter\_list[i][1] == "int":

format\_str += "%d"

elif p[3].parameter\_list[i][1] == "boolean":

format\_str += "%d"

elif p[3].parameter\_list[i][1] == "char":

format\_str += "%c"

else:

format\_str += "%f"

id\_str += "," + p[3].parameter\_list[i][0]

p[0].str = "printf(\"" + format\_str+ "\"" + id\_str + ");\n"

def p\_call\_procedure\_statement4(p):

'''call\_procedure\_statement : READ '(' variables ')' '''

#action

format\_str = ""

id\_str = ""

for i in range(0, len(p[3].parameter\_list)):

if p[3].parameter\_list[i][1] == "int":

format\_str += "%d"

elif p[3].parameter\_list[i][1] == "boolean":

format\_str += "%d"

elif p[3].parameter\_list[i][1] == "char":

format\_str += "%c"

else:

format\_str += "%f"

id\_str += ",&" + p[3].parameter\_list[i][0]

p[0] = Attribute()

p[0].str = "scanf(\"" + format\_str + "\"" + id\_str + ");\n"

if p[3].type == "type\_error":

p[0].type = "type\_error"

my\_print("read parameters type error at line " + str(p.lineno(1)))

def p\_variables(p):

'''variables : variables ',' variable

| variable '''

#action

p[0] = Attribute()

if len(p) == 4:

p[1].parameter\_list.append((p[3].str, p[3].type))

p[0].parameter\_list = p[1].parameter\_list[:]

if p[1].type == "type\_error" or p[3].type == "type\_error":

p[0].type = "type\_error"

else:

p[0].parameter\_list = p[1].parameter\_list[:]

p[0].type = p[1].type

def p\_expression\_list(p):

'''expression\_list : expression\_list ',' expression

| expression '''

#action

if len(p) == 4:

p[0] = Attribute()

p[0].str = p[1].str + ','+ p[3].str

#如果作为数组下标检查常量

p[0].type = str(p[1].type) + str(p[3].type)

p[0].parameter\_list = []

p[0].var\_list = p[1].var\_list + p[3].var\_list

p[1].parameter\_list.append((p[3].str, p[3].type))

for i in p[1].parameter\_list:

p[0].parameter\_list.append(i)

if p[0].type == "type\_error" or p[3].type == "type\_error":

p[0].type = "type\_error"

else:

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

p[0].parameter\_list =[]

p[0].parameter\_list.append((p[1].str,p[1].type))

def p\_expression1(p):

'''expression : simple\_expression RELOP simple\_expression

| simple\_expression '''

#action

if len(p) == 4:

p[0] = Attribute()

p[0].str = p[1].str + p[2] + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type == "type\_error" or p[3].type == "type\_error":

p[0].type = "type\_error"

elif p[1].type != p[3].type:

my\_print("types don't match at line " + str(p.lineno(2)))

p[0].type = "type\_error"

else:

p[0].type = "boolean"

else:

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

p[0].name = p[1].name

def p\_expression2(p):

'''expression : simple\_expression '=' simple\_expression'''

#action

p[0] = Attribute()

p[0].str = p[1].str + "=" + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type == "type\_error" or p[3].type == "type\_error":

p[0].type = "type\_error"

elif p[1].type != p[3].type:

my\_print("types don't match at line " + str(p.lineno(2)))

p[0].type = "type\_error"

else:

p[0].type = "boolean"

def p\_simple\_expression1(p):

'''simple\_expression : term'''

#actionb+

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

p[0].name = p[1].name

def p\_simple\_expression2(p):

'''simple\_expression : '+' term'''

#action

p[0] = Attribute()

p[0].str = "+" + p[2].str

if p[2].type == "float" or p[2].type == "int":

p[0].type = p[2].type

else:

if p[2].type != "type\_error":

my\_print("Opreator '+' can not deal with type " + p[2].type + " at line " + str(p.lineno(1)))

p[0].type = "type\_error"

p[0].lineno = p.lineno(1)

def p\_simple\_expression3(p):

'''simple\_expression : '-' term '''

#action

p[0] = Attribute()

p[0].str = "-" + p[2].str

if p[2].type == "float" or p[2].type == "int":

p[0].type = p[2].type

else:

if p[2].type != "type\_error":

my\_print("Opreator '-' can not deal with type " + p[2].type + " at line " + str(p.lineno(1)))

p[0].type = "type\_error"

p[0].lineno = p.lineno(1)

def p\_simple\_expression4(p):

'''simple\_expression : simple\_expression '+' term '''

#action

p[0] = Attribute()

p[0].str = p[1].str + "+" + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type in ["int", "float"] and p[3].type in ["int", "float"]:

if p[1].type == "int" and p[3].type == "int":

p[0].type = "int"

else:

p[0].type = "float"

else:

my\_print("Type " + p[1].type + " can not add with type " + p[3].type + " at line " + str(p.lineno(2)))

p[0].type = "type\_error"

def p\_simple\_expression5(p):

'''simple\_expression : simple\_expression '-' term '''

#action

p[0] = Attribute()

p[0].str = p[1].str + "-" + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type in ["int", "float"] and p[3].type in ["int", "float"]:

if p[1].type == "int" and p[3].type == "int":

p[0].type = "int"

else:

p[0].type = "float"

else:

my\_print("Type " + p[1].type + " can not sub with type " + p[3].type + " at line " + str(p.lineno(2)))

p[0].type = "type\_error"

def p\_simple\_expression6(p):

'''simple\_expression : simple\_expression OR term '''

#action

p[0] = Attribute()

if p[1].type == "boolean" and p[3].type == "boolean":

p[0].str = p[1].str + "||" + p[3].str

else:

p[0].str = p[1].str + "|" + p[3].str

p[0].lineno = p.lineno(2)

if p[1].type == p[3].type:

if p[1].type == "int":

p[0].type = "int"

elif p[1].type == "boolean":

p[0].type = "boolean"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Type error at line " + str(p.lineno(2)) + " , the operands must be integer or boolean")

p[0].type = "type\_error"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Types conflict at line " + str(p.lineno(2)) + " , and the operands must be integer or boolean")

p[0].type = "type\_error"

def p\_term(p):

'''term : term MULOP factor

| factor '''

#action

p[0] = Attribute()

if len(p) == 4:

p[0].lineno = p.lineno(2)

if p[2] == 'mod':

STR = '%'

if p[1].type == p[3].type:

if p[1].type == "int":

p[0].type = "int"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Type error at line " + str(p.lineno(2)) + ", the operands must be integer or boolean")

p[0].type = "type\_error"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Types conflict at line " + str(p.lineno(2)) + ", and the operands must be integer or boolean")

p[0].type = "type\_error"

elif p[2] == 'div':

STR = '/'

if p[1].type == p[3].type:

if p[1].type == "int":

p[0].type = "int"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Type error at line " + str(p.lineno(2)) + ", the operands must be integer")

p[0].type = "type\_error"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Types conflict at line " + str(p.lineno(2)) + ", and the operands must be integer")

p[0].type = "type\_error"

elif p[2] == 'and':

STR = '&'

if p[1].type == p[3].type:

if p[1].type == "int":

p[0].type = "int"

elif p[1].type == "boolean":

STR = "&&"

p[0].type = "boolean"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Type error at line " + str(p.lineno(2)) + ", the operands must be integer or boolean")

p[0].type = "type\_error"

else:

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Types conflict at line " + str(p.lineno(2)) + ", and the operands must be integer or boolean")

p[0].type = "type\_error"

else:

STR = p[2]

if p[1].type in ["int", "float"] and p[3].type in ["int", "float"]:

p[0].type = "float"

else:

p[0].type = "type\_error"

if p[1].type != "type\_error" and p[3].type != "type\_error":

my\_print("Type error at line " + str(p.lineno(2)) + ", the operands must be integer or real")

p[0].str = p[1].str + STR + p[3].str

else:

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

p[0].name = p[1].name

def p\_factor1(p):

'''factor : unsign\_const\_variable '''

#action

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

def p\_factor2(p):

'''factor : variable '''

#action

p[0] = Attribute()

p[0].str = p[1].str

p[0].type = p[1].type

p[0].lineno = p[1].lineno

p[0].name = p[1].name

def p\_factor3(p):

'''factor : ID '(' expression\_list ')' '''

#action

p[0] = Attribute()

p[0].lineno = p.lineno(1)

p[0].name = p[1]

#函数调用翻译

str1 = p[3].str

item\_list = str1.split(",")

var\_list = symboltable.get(p[1])["var\_list"]

for i in range(0,len(item\_list)):

if(var\_list[i+1] == "true"):

if "array" not in symboltable.get\_type(item\_list[i]):

item\_list[i] = "&" + item\_list[i]

else:

item\_list[i] = item\_list[i]

p[3].str = ""

for i in item\_list:

p[3].str += "," + i

p[3].str = p[3].str[1:]

p[0].str = p[1] + "\_function(" + p[3].str + ")"

#函数调用翻译结束

if not symboltable.find(p[1]):

my\_print("function " + str(p[1]) + " not defined at line " + str(p.lineno(1)))

p[0].type = "type\_error"

elif symboltable.get\_type(p[1]) != "function":

my\_print(str(p[1]) + " at line " + str(p.lineno(1)) + " is not a function")

p[0].type = "type\_error"

elif p[3].type != "".join(symboltable.get(p[1])['extend'][1:]):

my\_print("function's parameters do not match at line " + str(p.lineno(1)))

p[0].type = "type\_error"

else:

p[0].type = symboltable.get(p[1])["extend"][0]

def p\_factor4(p):

'''factor : '(' expression ')' '''

#action

p[0] = Attribute()

p[0].str = "(" + p[2].str + ")"

p[0].type = p[2].type

p[0].lineno = p[2].lineno

def p\_factor5(p):

'''factor : NOT factor '''

#action

p[0] = Attribute()

p[0].str = "~" + p[2].str

p[0].type = p[2].type

p[0].lineno = p.lineno(1)

def p\_unsigned\_const\_variable1(p):

'''unsign\_const\_variable : NUM '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(p[1])

p[0].lineno = p.lineno(1)

p[0].type = "int"

def p\_unsigned\_const\_variable2(p):

'''unsign\_const\_variable : CHARACTER '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = "\""+p[1]+"\""

p[0].lineno = p.lineno(1)

p[0].type = "char"

def p\_unsigned\_const\_variable3(p):

'''unsign\_const\_variable : FLOAT '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(p[1])

p[0].lineno = p.lineno(1)

p[0].type = "float"

def p\_unsigned\_const\_variable\_11(p):

'''unsign\_const\_variable : TRUE '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(1)

p[0].type = "boolean"

p[0].lineno = p.lineno(1)

def p\_unsigned\_const\_variable\_12(p):

'''unsign\_const\_variable : FALSE '''

#action

p[0] = Attribute()

p[0].value = p[1]

p[0].str = str(0)

p[0].type = "boolean"

p[0].lineno = p.lineno(1)

def p\_empty(p):

'''empty :'''

pass

def p\_error(p):

print("SyntaxError!", p)

def p\_program\_head\_error0(p):

'''program\_head : PROGRAM ID error ')' ';' '''

#action

my\_print(p[3])

if p[3].type == 'ID':

my\_print("Expected a '(' in line " + str(p.lineno(3)))

def p\_program\_head\_error1(p):

'''program\_head : PROGRAM ID '(' error ')' ';' '''

# action

my\_print("Expected a 'ID' in line " + str(p.lineno(4)))

def p\_program\_head\_error2(p):

'''program\_head : PROGRAM ID '(' identifier\_list error ';' '''

# action

my\_print("Expected a ')' in line "+ str(p.lineno(5)))

def p\_program\_head\_error3(p):

'''program\_head : PROGRAM ID '(' identifier\_list ')' error '''

# action

my\_print("Expected a ';' before line " + str(p.lineno(6)))

def p\_identifier\_list\_error(p):

'''identifier\_list : error ',' ID'''

#action

my\_print("Expected a 'ID' in line " + str(p.lineno(1)))

def p\_const\_declaration\_error0(p):

'''const\_declaration : error ';' Q ID '=' const\_variable'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Const variable defined error in line ", str(p.lineno(1)))

def p\_const\_declaration\_error1(p):

'''const\_declaration : error '=' const\_variable'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'ID' before '=' in line " + str(p.lineno(1)))

def p\_const\_declaration\_error2(p):

'''const\_declaration : const\_declaration ';' Q error '=' const\_variable'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'ID' before '=' in line " + str(p.lineno(4)))

def p\_const\_declaration\_error3(p):

'''const\_declaration : const\_declaration ';' Q ID '=' error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a variable assign to ID in line " + str(p.lineno(6)))

def p\_const\_declaration\_error4(p):

'''const\_declaration : ID '=' error'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a variable assign to ID in line " + str(p.lineno(3)))

def p\_const\_declaration\_error5(p):

'''const\_declaration : const\_declaration error Q ID '=' const\_variable'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(1)))

def p\_type\_declaration\_error0(p):

'''type\_declaration : error ';' S ID '=' type'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Type definition error in line " + str(p.lineno(1)))

def p\_type\_declaration\_error1(p):

'''type\_declaration : error '=' type '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'ID' before '=' in line " + str(p.lineno(1)))

def p\_type\_declaration\_error2(p):

'''type\_declaration : type\_declaration ';' S error '=' type'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'ID' before '=' in line " + str(p.lineno(4)))

def p\_type\_declaration\_error3(p):

'''type\_declaration : type\_declaration ';' S ID '=' error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'type' assign to ID before '=' in line " + str(p.lineno(6)))

def p\_type\_declaration\_error4(p):

'''type\_declaration : ID '=' error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'type' assign to ID before '=' in line " + str(p.lineno(3)))

def p\_type\_declaration\_error5(p):

'''type\_declaration : type\_declaration error S ID '=' type'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(2)))

def p\_type\_error0(p):

'''type : ARRAY error OF type'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Array index definition error in line " + str(p.lineno(2)))

def p\_type\_error1(p):

'''type : ARRAY '[' periods ']' OF error'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Array type definition error in line " + str(p.lineno(6)))

def p\_periods\_error(p):

'''periods : error ',' period'''

# action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Array index definition error in line " + str(p.lineno(1)))

def p\_period\_error(p):

'''period : error TOTO const\_variable'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Array index definition error in line " + str(p.lineno(1)))

def p\_var\_declaration\_error0(p):

'''var\_declaration : var\_declaration error ':' type ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Variable definition error in line " + str(p.lineno(1)))

def p\_var\_declaration\_error1(p):

'''var\_declaration : var\_declaration identifier\_list ':' error ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a correct type behind ':' in line " + str(p.lineno(4)))

def p\_var\_declaration\_error2(p):

'''var\_declaration : error ':' type ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Variable definition error in line " + str(p.lineno(1)))

def p\_var\_declaration\_error3(p):

'''var\_declaration : identifier\_list ':' error ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a correct type behind ':' in line " + str(p.lineno(3)))

def p\_var\_declaration\_error4(p):

'''var\_declaration : var\_declaration identifier\_list ':' type error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(5)))

def p\_var\_declaration\_error5(p):

'''var\_declaration : identifier\_list ':' type error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(4)))

def p\_subprogram\_head\_error0(p):

'''subprogram\_head : X FUNCTION ID '(' error ':' standard\_type ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Parameter list definition error in line " + str(p.lineno(5)))

def p\_subprogram\_head\_error1(p):

'''subprogram\_head : X FUNCTION ID '(' error ':' ID ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Parameter list definition error in line " + str(p.lineno(5)))

def p\_subprogram\_head\_error2(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' error ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Function return type error in line " + str(p.lineno(6)))

def p\_subprogram\_head\_error3(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' standard\_type error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(7)))

def p\_subprogram\_head\_error4(p):

'''subprogram\_head : X FUNCTION ID formal\_parameter ':' ID error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(7)))

def p\_subprogram\_head\_error5(p):

'''subprogram\_head : X PROCEDURE ID '(' error ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Parameter list definition error in line " + str(p.lineno(4)))

def p\_subprogram\_head\_error6(p):

'''subprogram\_head : X PROCEDURE ID formal\_parameter error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(7)))

def p\_parameter\_lists\_error0(p):

'''parameter\_lists : error ';' parameter\_list '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Parameter list definition error in line " + str(p.lineno(1)))

def p\_parameter\_lists\_error1(p):

'''parameter\_lists : parameter\_lists error parameter\_list '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(2)))

def p\_parameter\_lists\_error2(p):

'''parameter\_lists : parameter\_lists ';' error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Parameter list definition error in line " + str(p.lineno(3)))

def p\_statement\_error0(p):

'''statement : error ASSIGNOP expression'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a variable ID before ':=' in line " + str(p.lineno(1)))

def p\_statement\_error1(p):

'''statement : IF expression error statement else\_part'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'then' behind 'if' in line " + str(p.lineno(3)))

def p\_statement\_error3(p):

'''statement : CASE expression OF case\_body error'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected an 'end' behind 'case' in line " + str(p.lineno(5)))

def p\_statement\_error4(p):

'''statement : WHILE expression error statement'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'do' behind 'while' in line " + str(p.lineno(3)))

def p\_statement\_error5(p):

'''statement : FOR error updown expression DO statement'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected an initial varible behind 'for' in line " + str(p.lineno(2)))

def p\_statement\_error6(p):

'''statement : FOR ID ASSIGNOP expression error expression DO statement'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'to' or 'down to' behind 'for' in line "+ str(p.lineno(5)))

def p\_statement\_error7(p):

'''statement : FOR ID ASSIGNOP expression updown expression error statement'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'do' behind 'for' in line"+ str(p.lineno(7)))

def p\_id\_varpart\_errpr(p):

'''id\_varpart : error ']' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a '[' or index error in line " + str(p.lineno(1)))

def p\_branch\_error(p):

'''branch : error ':' statement'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a const variable before ':' in line " + str(p.lineno(1)))

def p\_variables\_error(p):

'''variables : error ',' variable '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Record or Array error in line " + str(p.lineno(1)))

def p\_expression\_error0(p):

'''expression : error RELOP simple\_expression '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_expression\_error2(p):

'''expression : error '=' simple\_expression '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_simple\_expression\_error0(p):

'''simple\_expression : error '+' term '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_simple\_expression\_error1(p):

'''simple\_expression : error '-' term '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_simple\_expression\_error2(p):

'''simple\_expression : error OR term '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_term\_error(p):

'''term : error MULOP factor '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Incorrect object before relational operation in line " + str(p.lineno(1)))

def p\_factor\_error(p):

'''factor : error ')' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a '(' or a function in line " + str(p.lineno(1)))

def p\_compund\_statement\_error0(p):

'''compound\_statement : error END U'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'begin' before line " + str(p.lineno(1)))

def p\_compund\_statement\_error1(p):

'''compound\_statement : BEGIN statement\_list error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a 'end' to match the 'begin' in line " + str(p.lineno(1)))

def p\_compund\_statement\_error2(p):

'''compound\_statement : BEGIN error END U '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

# my\_print("Expected a 'end' to match the 'begin' in line " + str(p.lineno(2)))

def p\_program\_error(p):

'''program : program\_head program\_body '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a '.' at the end of program")

my\_print("completed")

def p\_statement\_list\_error(p):

'''statement\_list : statement\_list error ';' '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(2)))

def p\_const\_declarations\_error(p):

'''const\_declarations : CONST const\_declaration error '''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(3)))

def p\_subprogram\_declarations\_error(p):

'''subprogram\_declarations : subprogram\_declarations subprogram\_declaration error'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' before line " + str(p.lineno(3)))

def p\_program\_body\_error(p):

'''program\_body : const\_declarations type\_declarations var\_declarations subprogram\_declarations subprogram\_declaration error'''

#action

my\_print("Cannot find main function ")

def p\_variable\_error(p):

'''variable : ID id\_varparts error'''

#action

p[0] = Attribute()

p[0].type = "type\_error"

my\_print("Expected a ';' or operation near line " + str(p.lineno(3)))

symboltable = SymbolTable()

fp = None

TextBrowser =None

def my\_yacc(input\_data,textBrowser):

symboltable.rollback()

lexer = lex.lex()

global TextBrowser,fp

fp = open("output.c", "w")

TextBrowser = textBrowser

parser = yacc.yacc()

result = parser.parse(input\_data.lower(), debug=True)

# result = parser.parse(input\_data.lower())

# if \_\_name\_\_ == "\_\_main\_\_":

# symboltable = SymbolTable()

# fp = open("output.c", "w")

# with open("test.txt") as f:

# input\_data = f.read()

# print(input\_data)

# print("---------------------------------------------------------")

# parser = yacc.yacc()

#

# result = parser.parse(input\_data.lower())

my\_highlighter.py

import sys

from PyQt4.QtGui import \*

from PyQt4.QtCore import \*

import re

from test import \*

class MyHighlighter( QSyntaxHighlighter ):

def \_\_init\_\_( self, parent ):

QSyntaxHighlighter.\_\_init\_\_( self, parent )

self.parent = parent

self.highlightingRules = []

keyword = QTextCharFormat()

keyword.setForeground( Qt.darkBlue )

keyword.setFontWeight( QFont.Bold )

keywords = ['ARRAY', 'BEGIN', 'CASE','DO', 'TO','DOWNTO', 'ELSE', 'FOR', 'FUNCTION', 'END', 'PROGRAM',

'CONST', 'OF', 'OR', 'NOT', 'PROCEDURE', 'RECORD','REPEAT', 'IF', 'THEN', 'TYPE','UNTIL',

'VAR', 'WHILE', 'CHAR', 'REAL', 'INTEGER', 'BOOLEAN','READ','WRITE','array', 'begin',

'case','do', 'to','downto', 'else', 'for', 'function', 'end', 'program', 'const', 'of',

'or', 'not', 'procedure', 'record','repeat', 'if', 'then', 'type','until', 'var', 'while',

'char', 'real', 'integer', 'boolean','read','write','true','false','TRUE','FALSE']

for word in keywords:

pattern = QRegExp("\\b" + word + "\\b")

rule = (pattern,keyword)

self.highlightingRules.append( rule )

keyword = QTextCharFormat()

keyword.setForeground( Qt.red )

keyword.setFontWeight( QFont.Bold )

keywords = ["-",":=","=","\+","\\*","\.\.",">","<",">=","<=","\(","\)","\[","\]",":",";"]

for word in keywords:

pattern = QRegExp( word )

rule = (pattern,keyword)

self.highlightingRules.append( rule )

def highlightBlock( self, text ):

for rule in self.highlightingRules:

expression = QRegExp( rule[0] )

index = expression.indexIn( text )

while index >= 0:

length = expression.matchedLength()

substr = text[index:index+length]

self.setFormat( index, length, rule[1] )

index = text.find(substr,index + length)

self.setCurrentBlockState( 0 )

**#QT.py**

# -\*- coding: utf-8 -\*-

# Form implementation generated from reading ui file 'QT.ui'

#

# Created by: PyQt4 UI code generator 4.11.4

#

# WARNING! All changes made in this file will be lost!

from PyQt4 import QtCore, QtGui

try:

\_fromUtf8 = QtCore.QString.fromUtf8

except AttributeError:

def \_fromUtf8(s):

return s

try:

\_encoding = QtGui.QApplication.UnicodeUTF8

def \_translate(context, text, disambig):

return QtGui.QApplication.translate(context, text, disambig, \_encoding)

except AttributeError:

def \_translate(context, text, disambig):

return QtGui.QApplication.translate(context, text, disambig)

class Ui\_MainWindow(object):

def setupUi(self, MainWindow):

MainWindow.setObjectName(\_fromUtf8("MainWindow"))

MainWindow.resize(818, 553)

self.centralwidget = QtGui.QWidget(MainWindow)

self.centralwidget.setObjectName(\_fromUtf8("centralwidget"))

self.textEdit = QtGui.QTextEdit(self.centralwidget)

self.textEdit.setGeometry(QtCore.QRect(0, 0, 821, 371))

self.textEdit.setStyleSheet(\_fromUtf8("font: 75 14pt \"Bell MT\";"))

self.textEdit.setObjectName(\_fromUtf8("textEdit"))

self.textBrowser = QtGui.QTextBrowser(self.centralwidget)

self.textBrowser.setGeometry(QtCore.QRect(0, 370, 821, 151))

self.textBrowser.setStyleSheet(\_fromUtf8("font: 75 12pt \"Adobe Gothic Std B\";"))

self.textBrowser.setObjectName(\_fromUtf8("textBrowser"))

self.lineEdit = QtGui.QLineEdit(self.centralwidget)

self.lineEdit.setGeometry(QtCore.QRect(710, 0, 113, 20))

self.lineEdit.setObjectName(\_fromUtf8("lineEdit"))

MainWindow.setCentralWidget(self.centralwidget)

self.menubar = QtGui.QMenuBar(MainWindow)

self.menubar.setGeometry(QtCore.QRect(0, 0, 818, 23))

self.menubar.setObjectName(\_fromUtf8("menubar"))

self.menuCaidan = QtGui.QMenu(self.menubar)

self.menuCaidan.setObjectName(\_fromUtf8("menuCaidan"))

self.menuRun = QtGui.QMenu(self.menubar)

self.menuRun.setObjectName(\_fromUtf8("menuRun"))

self.menuJump = QtGui.QMenu(self.menubar)

self.menuJump.setObjectName(\_fromUtf8("menuJump"))

MainWindow.setMenuBar(self.menubar)

self.statusbar = QtGui.QStatusBar(MainWindow)

self.statusbar.setObjectName(\_fromUtf8("statusbar"))

MainWindow.setStatusBar(self.statusbar)

self.actionOpen = QtGui.QAction(MainWindow)

self.actionOpen.setObjectName(\_fromUtf8("actionOpen"))

self.actionNew = QtGui.QAction(MainWindow)

self.actionNew.setObjectName(\_fromUtf8("actionNew"))

self.actionRun = QtGui.QAction(MainWindow)

self.actionRun.setObjectName(\_fromUtf8("actionRun"))

self.actionSave = QtGui.QAction(MainWindow)

self.actionSave.setObjectName(\_fromUtf8("actionSave"))

self.actionJump\_to\_line = QtGui.QAction(MainWindow)

self.actionJump\_to\_line.setObjectName(\_fromUtf8("actionJump\_to\_line"))

self.menuCaidan.addAction(self.actionOpen)

self.menuCaidan.addAction(self.actionNew)

self.menuCaidan.addAction(self.actionSave)

self.menuRun.addAction(self.actionRun)

self.menuJump.addAction(self.actionJump\_to\_line)

self.menubar.addAction(self.menuCaidan.menuAction())

self.menubar.addAction(self.menuRun.menuAction())

self.menubar.addAction(self.menuJump.menuAction())

self.retranslateUi(MainWindow)

QtCore.QMetaObject.connectSlotsByName(MainWindow)

def retranslateUi(self, MainWindow):

MainWindow.setWindowTitle(\_translate("MainWindow", "MainWindow", None))

self.menuCaidan.setTitle(\_translate("MainWindow", "File", None))

self.menuRun.setTitle(\_translate("MainWindow", "Run", None))

self.menuJump.setTitle(\_translate("MainWindow", "jump", None))

self.actionOpen.setText(\_translate("MainWindow", "Open", None))

self.actionNew.setText(\_translate("MainWindow", "New", None))

self.actionRun.setText(\_translate("MainWindow", "Compaile", None))

self.actionSave.setText(\_translate("MainWindow", "Save", None))

self.actionJump\_to\_line.setText(\_translate("MainWindow", "jump to line", None))

**#support\_function.py**

# -\*- coding: utf-8 -\*-

# Form implementation generated from reading ui file 'QT.ui'

#

# Created by: PyQt4 UI code generator 4.11.4

#

# WARNING! All changes made in this file will be lost!

from PyQt4 import QtCore, QtGui

try:

\_fromUtf8 = QtCore.QString.fromUtf8

except AttributeError:

def \_fromUtf8(s):

return s

try:

\_encoding = QtGui.QApplication.UnicodeUTF8

def \_translate(context, text, disambig):

return QtGui.QApplication.translate(context, text, disambig, \_encoding)

except AttributeError:

def \_translate(context, text, disambig):

return QtGui.QApplication.translate(context, text, disambig)

class Ui\_MainWindow(object):

def setupUi(self, MainWindow):

MainWindow.setObjectName(\_fromUtf8("MainWindow"))

MainWindow.resize(818, 553)

self.centralwidget = QtGui.QWidget(MainWindow)

self.centralwidget.setObjectName(\_fromUtf8("centralwidget"))

self.textEdit = QtGui.QTextEdit(self.centralwidget)

self.textEdit.setGeometry(QtCore.QRect(0, 0, 821, 371))

self.textEdit.setStyleSheet(\_fromUtf8("font: 75 14pt \"Bell MT\";"))

self.textEdit.setObjectName(\_fromUtf8("textEdit"))

self.textBrowser = QtGui.QTextBrowser(self.centralwidget)

self.textBrowser.setGeometry(QtCore.QRect(0, 370, 821, 151))

self.textBrowser.setStyleSheet(\_fromUtf8("font: 75 12pt \"Adobe Gothic Std B\";"))

self.textBrowser.setObjectName(\_fromUtf8("textBrowser"))

self.lineEdit = QtGui.QLineEdit(self.centralwidget)

self.lineEdit.setGeometry(QtCore.QRect(710, 0, 113, 20))

self.lineEdit.setObjectName(\_fromUtf8("lineEdit"))

MainWindow.setCentralWidget(self.centralwidget)

self.menubar = QtGui.QMenuBar(MainWindow)

self.menubar.setGeometry(QtCore.QRect(0, 0, 818, 23))

self.menubar.setObjectName(\_fromUtf8("menubar"))

self.menuCaidan = QtGui.QMenu(self.menubar)

self.menuCaidan.setObjectName(\_fromUtf8("menuCaidan"))

self.menuRun = QtGui.QMenu(self.menubar)

self.menuRun.setObjectName(\_fromUtf8("menuRun"))

self.menuJump = QtGui.QMenu(self.menubar)

self.menuJump.setObjectName(\_fromUtf8("menuJump"))

MainWindow.setMenuBar(self.menubar)

self.statusbar = QtGui.QStatusBar(MainWindow)

self.statusbar.setObjectName(\_fromUtf8("statusbar"))

MainWindow.setStatusBar(self.statusbar)

self.actionOpen = QtGui.QAction(MainWindow)

self.actionOpen.setObjectName(\_fromUtf8("actionOpen"))

self.actionNew = QtGui.QAction(MainWindow)

self.actionNew.setObjectName(\_fromUtf8("actionNew"))

self.actionRun = QtGui.QAction(MainWindow)

self.actionRun.setObjectName(\_fromUtf8("actionRun"))

self.actionSave = QtGui.QAction(MainWindow)

self.actionSave.setObjectName(\_fromUtf8("actionSave"))

self.actionJump\_to\_line = QtGui.QAction(MainWindow)

self.actionJump\_to\_line.setObjectName(\_fromUtf8("actionJump\_to\_line"))

self.menuCaidan.addAction(self.actionOpen)

self.menuCaidan.addAction(self.actionNew)

self.menuCaidan.addAction(self.actionSave)

self.menuRun.addAction(self.actionRun)

self.menuJump.addAction(self.actionJump\_to\_line)

self.menubar.addAction(self.menuCaidan.menuAction())

self.menubar.addAction(self.menuRun.menuAction())

self.menubar.addAction(self.menuJump.menuAction())

self.retranslateUi(MainWindow)

QtCore.QMetaObject.connectSlotsByName(MainWindow)

def retranslateUi(self, MainWindow):

MainWindow.setWindowTitle(\_translate("MainWindow", "MainWindow", None))

self.menuCaidan.setTitle(\_translate("MainWindow", "File", None))

self.menuRun.setTitle(\_translate("MainWindow", "Run", None))

self.menuJump.setTitle(\_translate("MainWindow", "jump", None))

self.actionOpen.setText(\_translate("MainWindow", "Open", None))

self.actionNew.setText(\_translate("MainWindow", "New", None))

self.actionRun.setText(\_translate("MainWindow", "Compaile", None))

self.actionSave.setText(\_translate("MainWindow", "Save", None))

self.actionJump\_to\_line.setText(\_translate("MainWindow", "jump to line", None))

**#GUI.py**

import sys

from PyQt4 import \*

from my\_yacc\_debug import\*

from myhighter import MyHighlighter

from QT import \*

class MyWidget(QtGui.QMainWindow):

ui = Ui\_MainWindow()

def \_\_init\_\_(self, parent=None):

super(MyWidget, self).\_\_init\_\_(parent)

self.ui.setupUi(self)

self.highlighter = MyHighlighter(self.ui.textEdit.document())

@QtCore.pyqtSlot()

def on\_actionOpen\_triggered(self):

fg = QtGui.QFileDialog()

dir = QtCore.QDir()

filename = fg.getOpenFileName(fg,"Open File",dir.currentPath(),"Pascal Files (\*.pas);;Text Files (\*.txt)")

if(filename==""):

pass

else:

fp = open(filename)

content = fp.read()

self.ui.textEdit.setPlainText(content)

self.setWindowTitle("Pascal--"+filename)

@QtCore.pyqtSlot()

def on\_actionRun\_triggered(self):

self.ui.textBrowser.clear()

inout\_data = my\_yacc(self.ui.textEdit.toPlainText(),self.ui.textBrowser)

@QtCore.pyqtSlot()

def on\_actionNew\_triggered(self):

if(self.ui.textEdit.document().isModified()):

message = QtGui.QMessageBox()

message.information(message,"Error Message","The current file has been modified,Please save the current File")

else:

self.ui.textEdit.clear()

self.setWindowTitle("Untitled")

@QtCore.pyqtSlot()

def on\_actionSave\_triggered(self):

fg = QtGui.QFileDialog()

dir = QtCore.QDir()

filename = fg.getSaveFileName(fg,"Open File",dir.currentPath())

if(filename==""):

pass

fp = open(filename,'w')

content = self.ui.textEdit.toPlainText()

print(content,file = fp)

self.ui.textEdit.document().setModified(False)

@QtCore.pyqtSlot()

def on\_actionJump\_to\_line\_triggered(self):

input = self.ui.lineEdit.text()

self.ui.lineEdit.clear()

TextLine = self.ui.textEdit.document().lineCount()

if (int(input)>TextLine):

message = QtGui.QMessageBox()

message.information(message,"Error Message","The linenumer too big!")

else:

linenumber = int(input)

block = self.ui.textEdit.document().findBlockByLineNumber(linenumber-1)

self.ui.textEdit.setTextCursor(QtGui.QTextCursor(block))

if \_\_name\_\_ == "\_\_main\_\_":

app = QtGui.QApplication(sys.argv)

mainwindow = MyWidget()

mainwindow.show()

app.exec\_()