

Compilador para C-

Doc. Victor de la Cueva

Diseño de compiladores

Isaias Martinez Vieyra

A00988525

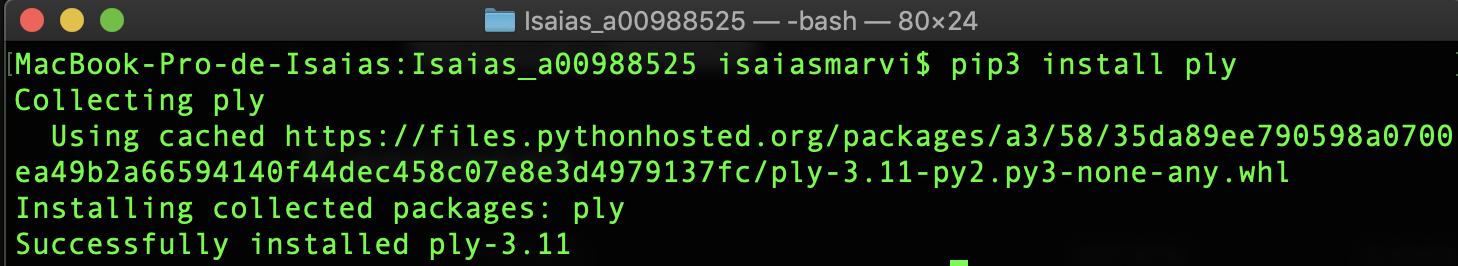
Hemos creado un compilador el cual está escrito en Python, el cual recibe un archivo de extencion .C- y genera codigo MIPS. Este archivo pasa por 3 etapas lexico,sintactica y semantica. En estas se valida que el archivo esté bien escrito en el lenguaje C-. En esté documento nos enfocaremos en la etapa posterior a la validacion del lexico, sintactica y semantica la cual es la generacion de código. Se ha decidido ocupar como lenguaje de salida MIPS por su basta documentación y por ser el recomendado por el profesor.

Instrucciones para utilizar el Compilador

Para poder utilizar el compilador es necesario tener en su maquina instalado Python 3 y PLY.

Para instalar python es necesario ingresar a la siguiente liga <https://www.python.org/downloads/> y descargar la version correspondiente a su maquina.

Posteriormente instalamos ply, para esto abrimos una terminal y corremos la siguiente linea “pip install ply”.



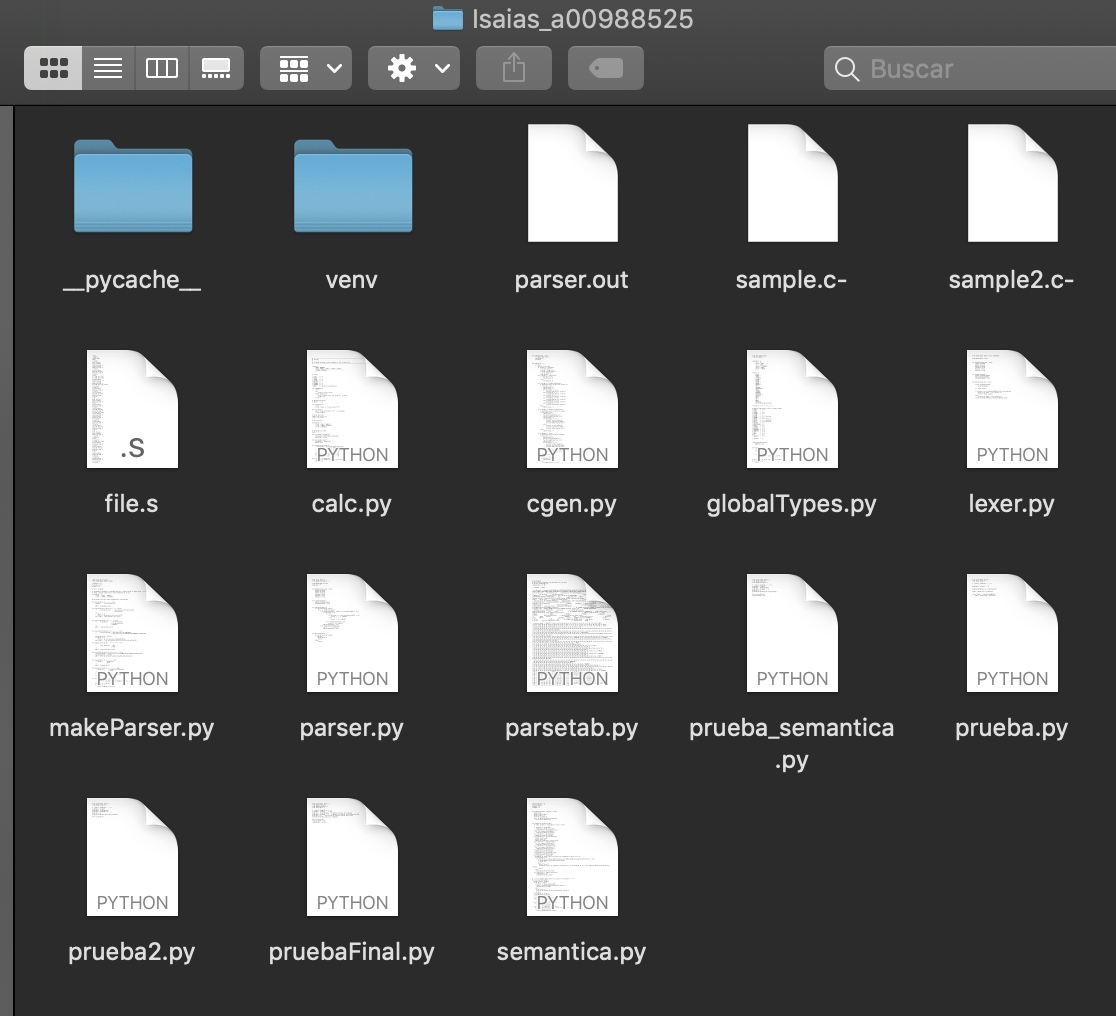
Una vez instalado lo anterior ya podemos hacer uso de nuestro compilador, para esto es necesario tener un archivo en el lenguaje C- en la misma carpeta de nuestro compilador.

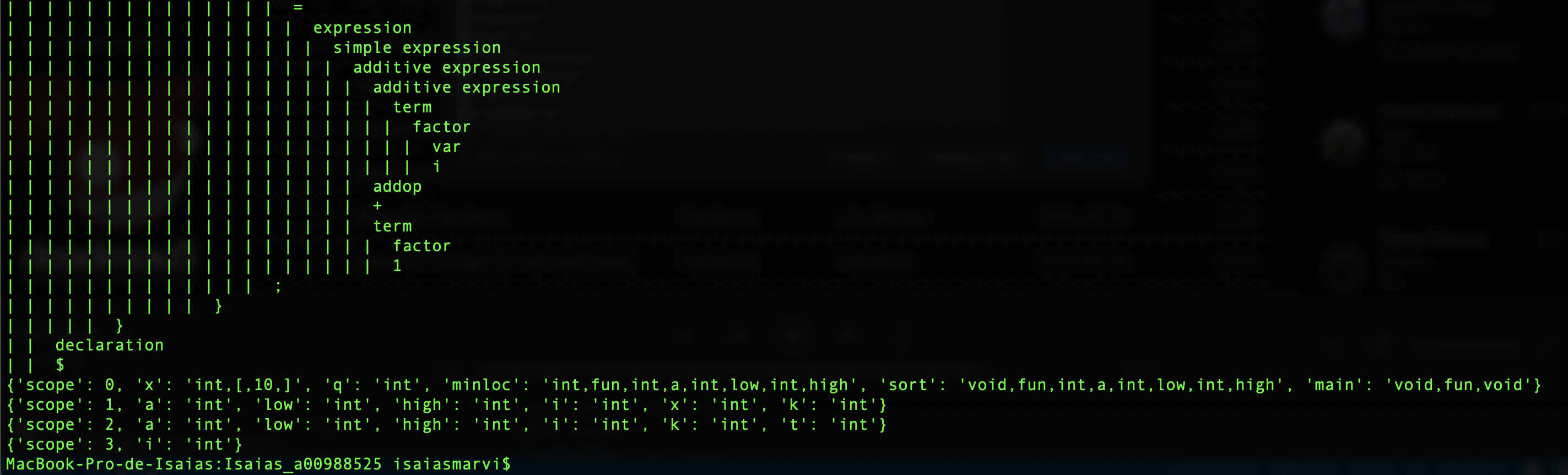
Posteriormente puedes crear un archivo como el siguiente:

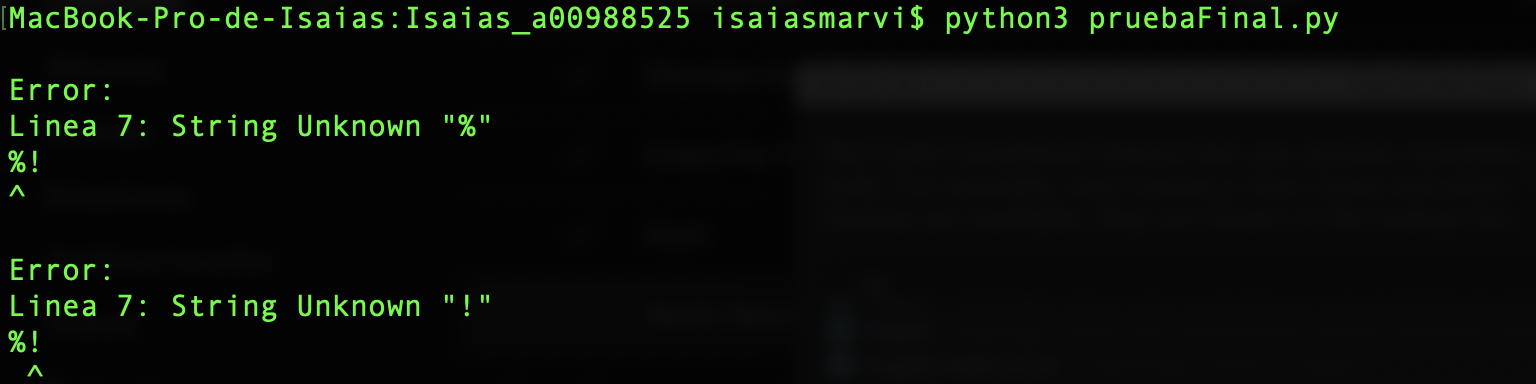
from globalTypes import \*  
from parser import \*  
from semantica import \*  
from cgen import \*  
  
f = open('./sample2.c-', 'r')  
programa = f.read() # lee  
progLong = len(programa) # longitud original del programa  
programa = programa + '$' # agregar un caracter $ que represente EOF  
posicion = 0 # posición del caracter actual del string  
globales(programa, posicion, progLong)  
  
AST = parse(True)  
semantica(AST, True)  
codeGen(AST, 'file.s')

Es necesario cambiar el nombre del archivo .C- dependiendo del nombre que le haya puesto al suyo. Una vez que tenga el archivo como el que mostramos arriba, lo corre en terminal con la siguiente linea. “ python ‘nombre.py’ ”

Si el archivo que tiene en C- no contiene fallas, vera desplegado en terminal el arbol AST y unos objetos JSON los cuales especifican las variables y los scopes identificados dentro de su codigo C-, ademas vera un archivo llamado “file.s” el cual usted puede modificar el nombre para que se llame como mas desee, solo que la extencion debera de ser .S para que lo pueda probar en un simulador. En el caso de que el archivo escrito en C- contenga errores, en la consola vera instrucciones de donde se encuentra el error en su codigo.







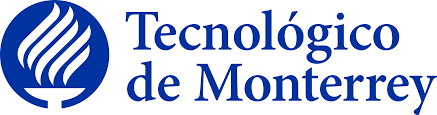
Bibligrafia:

<http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>

<https://www.cs.ucsb.edu/~franklin/64/lectures/mipsassemblytutorial.pdf>

<https://courses.cs.washington.edu/courses/cse378/03wi/lectures/mips-asm-examples.html>

Apendice:



Compilador Lexico

Doc. Victor de la Cueva

Diseño de compiladores

Isaias Martinez Vieyra

A00988525

Tokens:

Se componen de palabras reservadas e identificadores

reserved = {

'if': 'IF',

'else': 'ELSE',

'int': 'INT',

'return': 'RETURN',

'void': 'VOID',

'while': 'WHILE'

}

tokens = [

'PLUS',

'MINUS',

'TIMES',

'DIVIDE',

'LTHAN',

'LOEQU',

'MTHAN',

'MOEQU',

'EQUALS',

'DIFF',

'ASSIGN',

'SEMICOLON',

'COMA',

'LPAREN',

'RPAREN',

'LBRACKET',

'RBRACKET',

'LKEY',

'RKEY',

'COMMENTS',

'COMMENTE',

'ID',

'NUM',

'LETTER',

'DIGT',

'ENDFILE'

]+list(reserved.values())

Mis tokens se revisan con las siguientes expresiones regulares.

# Regular expression rules for simple tokens

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_TIMES = r'\\*'

t\_DIVIDE = r'/'

t\_LTHAN = r'<'

t\_LOEQU = r'<='

t\_MTHAN = r'>'

t\_MOEQU = r'>='

t\_EQUALS = r'=='

t\_DIFF = r'!='

t\_ASSIGN = r'='

t\_SEMICOLON = r';'

t\_COMA = r'\,'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

t\_LBRACKET = r'\['

t\_RBRACKET = r'\]'

t\_LKEY = r'\{'

t\_RKEY = r'\}'

eof = '$'

t\_ENDFILE = r'\$'

t\_ignore = ' \t'

def t\_NUM(t):

r'\d+'

try:

t.value = int(t.value)

return t

except ValueError:

print("Integer value too large %d", t.value)

# Quitamos los espacios vacios y tabs

def t\_newline(t):

r'\n+'

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

# Detectamos Errores, los impimimos y nos saltamos al sig token

def t\_error(t):

print("\nError: ")

print(f"Linea {t.lexer.lineno}: String Unknown \"{t.value[0]}\" ")

arr = t.lexer.lexdata.split('\n')

linea = arr[t.lexer.lineno-1].strip()

print(linea)

print(" "\*linea.find(t.value[0])+"^")

t.lexer.skip(1)

def t\_ID(t):

r'[a-zA-Z][a-zA-Z]\*\d+'

t.type = reserved.get(t.value, 'ID')

return t

def t\_COMMENT(t):

r'(/\\*(.|\n+)\*?\\*/)'

pass

DFA para deteccion de errores:

Estoy ocupando una Libreria similar a Lex, se llama ply, y es Lex adaptado a python ( <https://www.dabeaz.com/ply/> ), por lo que no hay necesidad de hacer otra implementacion o adaptar.

Dentro de la libreria de ply, existe el metodo para detectar errores “t\_error(t)”. El cual recibe un parametro tipo Token, este metodo se manda a llamar cuando no pasa ninguno de las expresiones regulares previamente definidas.

def t\_error(t):

print("\nError: ")

print(f"Linea {t.lexer.lineno}: String Unknown \"{t.value[0]}\" ")

arr = t.lexer.lexdata.split('\n')

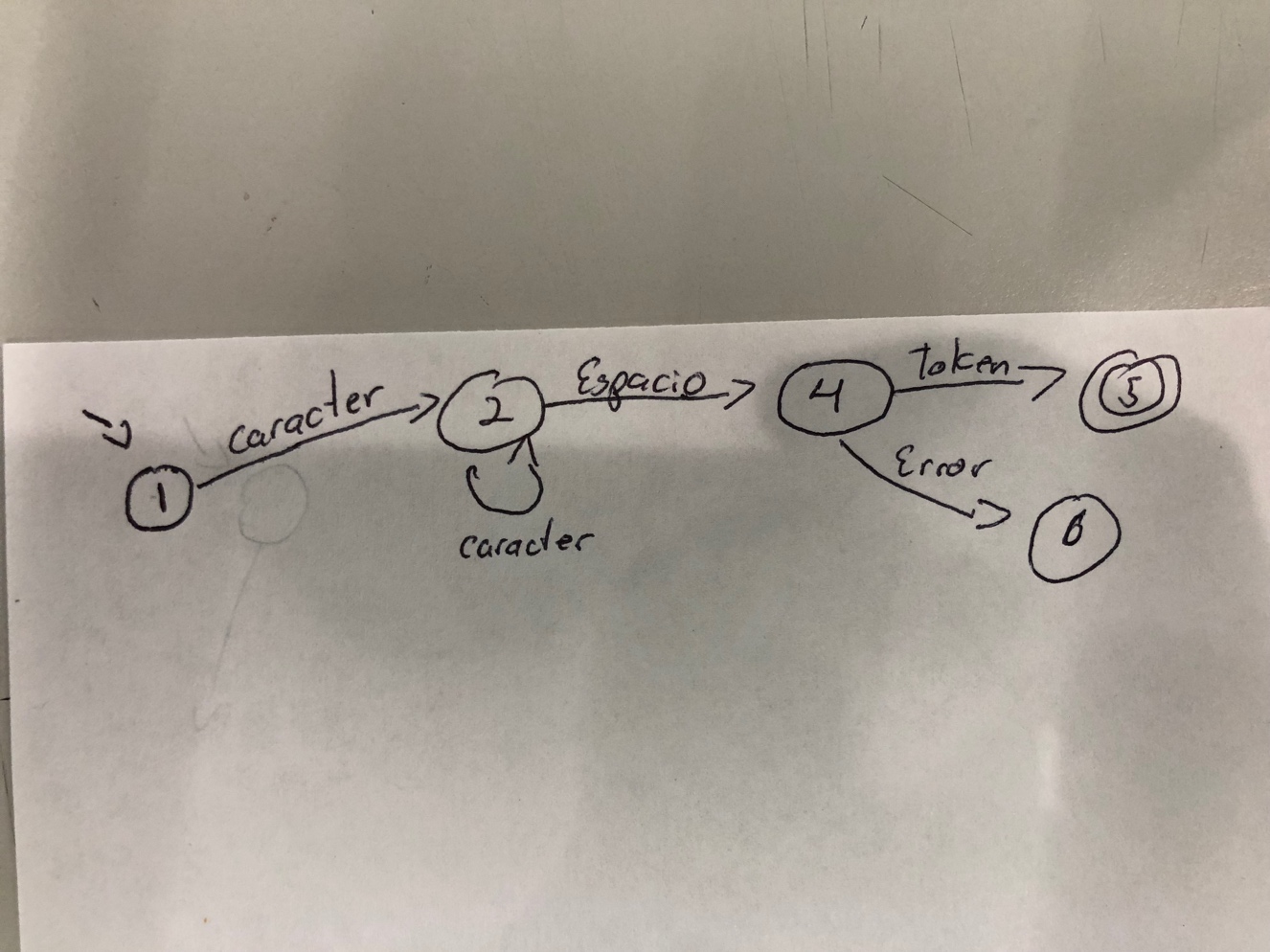
linea = arr[t.lexer.lineno-1].strip()

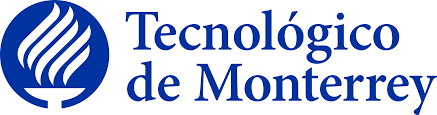
print(linea)

print(" "\*linea.find(t.value[0])+"^")

t.lexer.skip(1)

Lo que hace ply para reconocer es tomar la string mas larga hasta que exista un espacio y evalua si es Token lo acepta si no sale error.





Analizador Sintáctico

Doc. Victor de la Cueva

Diseño de compiladores

Isaias Martinez Vieyra

A00988525

Para el Analizador sintactico se ocupo una gramátrica BNF, se utilizo en un 99% la gramatica BNF definida en la definicion del lenguaje C-. Sin embargo se modificaron unas decalraciones ya que en nuestro lexer agregamos el token ENDFILE el cual habia que manejar.

Se está ocupando una librería para python llamada PLY, la cual trae un modulo llamado YACC, el cual nos ayuda a simplificar nuestro analizador sintactico, ya que lo define en funciónes, comenzando con p\_programa. Estas reglas gramaticales son pasadas a ply.yacc el cual crea reglas en el archivo ‘parser.out’ las cuales se muestran mas abajo.

A la hora de definir nuestras reglas con ply, el arreglo p[] guarda nuestro arbol de parseo definido por la gramatica, quedandose con todos los “No terminales” y generando revisando si los tokens siguientes hacen match con un estado o regla.

Notas:

Para poder correr los scripts es necesario instalar Yacc. para esto podemos ocupar cualquiera de los dos comandos.

“pip3 install ply” o “pip install ply”

def p\_program(p):

'program : declaration\_list'

p[0] = ('programa',p[1])

def p\_declaration\_list(p):

''' declaration\_list : declaration\_list declaration

| declaration

'''

if len(p) == 3:

p[0] = ('declaration list',p[1],p[2])

else: p[0] = ('declaration list',p[1])

def p\_declaration(p):

''' declaration : var\_declaration

| fun\_declaration

| ENDFILE

'''

p[0] = ('declaration',p[1])

def p\_var\_declaration(p):

''' var\_declaration : type\_specifier ID SEMICOLON

| type\_specifier ID LBRACKET NUM RBRACKET SEMICOLON

'''

if len(p) == 4:

p[0] = ('var declaration',p[1],p[2],p[3])

else: p[0] = ('var declaration',p[1],p[2],p[3],p[4],p[5],p[6])

def p\_type\_specifier(p):

'''

type\_specifier : INT

| VOID

'''

p[0] = ('type specifier',p[1])

def p\_fun\_declaration(p):

'fun\_declaration : type\_specifier ID LPAREN params RPAREN compound\_stmt'

p[0] = ('fun declaration',p[1],p[2],p[3],p[4],p[5],p[6])

def p\_params(p):

''' params : param\_list

| VOID

'''

p[0] = ('params',p[1])

def p\_param\_list(p):

''' param\_list : param\_list COMA param

| param

'''

if len(p) == 3:

p[0] = ('param list',p[1],p[2],p[3])

else: p[0] = ('param list',p[1])

def p\_param(p):

''' param : type\_specifier ID

| type\_specifier LBRACKET RBRACKET

'''

if len(p) == 3:

p[0] = ('param',p[1])

else: p[0] = ('param', p[1],p[2],p[3])

def p\_compound\_stmt(p):

'compound\_stmt : LKEY local\_declarations statement\_list RKEY'

p[0] = ('compound stmt',p[1],p[2],p[3],p[4])

def p\_local\_declarations(p):

''' local\_declarations : local\_declarations var\_declaration

|

'''

if len(p) == 3:

p[0] = ("local declarations", p[1],p[2])

def p\_statement\_list(p):

''' statement\_list : statement\_list statement

|

'''

if len(p) == 3:

p[0] = ('statement list',p[1],p[2])

def p\_statement(p):

''' statement : expression\_stmt

| compound\_stmt

| selection\_stmt

| iteration\_stmt

| return\_stmt

'''

p[0] = ('statement',p[1])

def p\_expression\_stmt(p):

''' expression\_stmt : expression SEMICOLON

| SEMICOLON

'''

if len(p) == 3:

p[0] = ('expression stmt', p[1],p[2])

else: p[0] = ('expression stmt',p[1])

def p\_selection\_stmt(p):

''' selection\_stmt : IF LPAREN expression RPAREN statement

| IF LPAREN expression RPAREN statement ELSE statement

'''

if len(p) == 6:

p[0] = ('selection stmt',p[1],p[2],p[3],p[4],p[5])

else: p[0] = ('selection stmt',p[1],p[2],p[3],p[4],p[5],p[6],p[7])

def p\_iteration\_stmt(p):

' iteration\_stmt : WHILE LPAREN expression RPAREN statement'

p[0] = ('iteration stmt',p[1],p[2],p[3],p[4],p[5])

def p\_return\_stmt(p):

''' return\_stmt : RETURN SEMICOLON

| RETURN expression SEMICOLON

'''

if len(p) == 4:

p[0] = ('return stmtm',p[1],p[2])

else: p[0] = ('return stmt',p[1],p[2],p[3])

def p\_expression(p):

''' expression : var ASSIGN expression

| simple\_expression

'''

if len(p) == 4:

p[0] = ('expression',p[1],p[2],p[3])

else: p[0] = ('expression',p[1])

def p\_var(p):

''' var : ID

| ID LBRACKET expression RBRACKET

'''

if len(p) == 5:

p[0] = ('var',p[1],p[2],p[3],p[4])

else: p[0] = ('var', p[1])

def p\_simple\_expression(p):

''' simple\_expression : additive\_expression relop additive\_expression

| additive\_expression

'''

if len(p) == 4:

p[0] = ('simple expression',p[1],p[2],p[3])

else: p[0] = ('simple expression',p[1])

def p\_relop(p):

''' relop : LOEQU

| LTHAN

| MTHAN

| MOEQU

| EQUALS

| DIFF

'''

p[0] = ('relop',p[1])

def p\_additive\_expression(p):

''' additive\_expression : additive\_expression addop term

| term

'''

if len(p) == 4:

p[0] = ('additive expression',p[1],p[2],p[3])

else: p[0] = ('additive expression',p[1])

def p\_addop(p):

''' addop : PLUS

| MINUS

'''

p[0] = p[1]

def p\_term(p):

''' term : term mulop factor

| factor

'''

if len(p) == 4:

p[0] = ('term',p[1],p[2],p[3])

else: p[0] = ('term',p[1])

def p\_mulop(p):

''' mulop : TIMES

| DIVIDE

'''

p[0] = ('mulop',p[1])

def p\_factor(p):

''' factor : LPAREN expression RPAREN

| ID

| call

| NUM

'''

if len(p) == 4:

p[0] = ('factor',p[1],p[2],p[3])

else: p[0] = ('factor',p[1])

def p\_call(p):

' call : ID LPAREN args RPAREN '

p[0] = ('call',p[1],p[2],p[3],p[4])

def p\_args(p):

''' args : arg\_list

|

'''

if len(p) == 2:

p[0] = ('args',p[1])

def p\_arg\_list(p):

''' arg\_list : arg\_list COMA expression

| expression

'''

if len(p) == 4:

p[0] = ('arg list',p[1],p[2],p[3])

else: p[0] = ('arg list',p[1])

# Error rule for syntax errors

def p\_error(p):

if not p:

print("End of File!")

return

while True:

# Get the next token

tok = parser.token()

print("next token = ",tok)

if not tok or tok.type == 'SEMICOLON':

break

parser.restart()

# Build the parser

parser = yacc.yacc(debug=True)

Rule 0 S' -> program

Rule 1 program -> declaration\_list

Rule 2 declaration\_list -> declaration\_list declaration

Rule 3 declaration\_list -> declaration

Rule 4 declaration -> var\_declaration

Rule 5 declaration -> fun\_declaration

Rule 6 declaration -> ENDFILE

Rule 7 var\_declaration -> type\_specifier ID SEMICOLON

Rule 8 var\_declaration -> type\_specifier ID LBRACKET NUM RBRACKET SEMICOLON

Rule 9 type\_specifier -> INT

Rule 10 type\_specifier -> VOID

Rule 11 fun\_declaration -> type\_specifier ID LPAREN params RPAREN compound\_stmt

Rule 12 params -> param\_list

Rule 13 params -> VOID

Rule 14 param\_list -> param\_list COMA param

Rule 15 param\_list -> param

Rule 16 param -> type\_specifier ID

Rule 17 param -> type\_specifier LBRACKET RBRACKET

Rule 18 compound\_stmt -> LKEY local\_declarations statement\_list RKEY

Rule 19 local\_declarations -> local\_declarations var\_declaration

Rule 20 local\_declarations -> <empty>

Rule 21 statement\_list -> statement\_list statement

Rule 22 statement\_list -> <empty>

Rule 23 statement -> expression\_stmt

Rule 24 statement -> compound\_stmt

Rule 25 statement -> selection\_stmt

Rule 26 statement -> iteration\_stmt

Rule 27 statement -> return\_stmt

Rule 28 expression\_stmt -> expression SEMICOLON

Rule 29 expression\_stmt -> SEMICOLON

Rule 30 selection\_stmt -> IF LPAREN expression RPAREN statement

Rule 31 selection\_stmt -> IF LPAREN expression RPAREN statement ELSE statement

Rule 32 iteration\_stmt -> WHILE LPAREN expression RPAREN statement

Rule 33 return\_stmt -> RETURN SEMICOLON

Rule 34 return\_stmt -> RETURN expression SEMICOLON

Rule 35 expression -> var ASSIGN expression

Rule 36 expression -> simple\_expression

Rule 37 var -> ID

Rule 38 var -> ID LBRACKET expression RBRACKET

Rule 39 simple\_expression -> additive\_expression relop additive\_expression

Rule 40 simple\_expression -> additive\_expression

Rule 41 relop -> LOEQU

Rule 42 relop -> LTHAN

Rule 43 relop -> MTHAN

Rule 44 relop -> MOEQU

Rule 45 relop -> EQUALS

Rule 46 relop -> DIFF

Rule 47 additive\_expression -> additive\_expression addop term

Rule 48 additive\_expression -> term

Rule 49 addop -> PLUS

Rule 50 addop -> MINUS

Rule 51 term -> term mulop factor

Rule 52 term -> factor

Rule 53 mulop -> TIMES

Rule 54 mulop -> DIVIDE

Rule 55 factor -> LPAREN expression RPAREN

Rule 56 factor -> ID

Rule 57 factor -> call

Rule 58 factor -> NUM

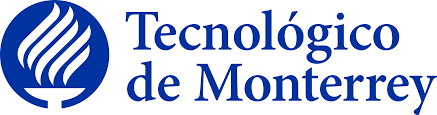
Rule 59 call -> ID LPAREN args RPAREN

Rule 60 args -> arg\_list

Rule 61 args -> <empty>

Rule 62 arg\_list -> arg\_list COMA expression

Rule 63 arg\_list -> expression



Analizador Semántico

Doc. Victor de la Cueva

Diseño de compiladores

Isaias Martinez Vieyra

A00988525

Para el Analizador semantico se utilizo el AST generado por el programa pasado. El AST nos ayudara a crear la tabla de simbolos y posteriormente evaluar los valores de las variables y que sean correctos y correspondan a sus declaraciones.

Para esto se ocuparon diversos metodos los cuales de forma recursiva y con estructuras de control se formo la tabla de simbolos y posteriormente se validaron los valores de las operaciones y de las asignaciones.

La estructura de control utilizada para manejar la tabla, fue un arreglo. Ya que C- no permite tener mas de dos scopes de profundidad por lo que siempre estara en evaluacion el global y un local, por lo que unicamente es necesario ir monitoreando en que scope te encuentras. Una vez encontrado una expression o una assignacion, es necesario checar que las partes correspondan.

Es importante recalcar que el analizador semantico no tiene recuperacíon de errores como tal, en cuanto encuentre una incongruencia, se mostrara el error y se dejara de correr el programa.

Notas:

Para poder correr los scripts es necesario instalar PLY. para esto podemos ocupar cualquiera de los dos comandos.

“pip3 install ply” o “pip install ply”

tabla\_simbolos = []

actual\_scope=0

funval = []

evalNum = 2

def semantica(arbol, imprime = True):

tabla(arbol)

global actual\_scope

actual\_scope = 0

semantica\_1(arbol,0)

def semantica\_1(arbol,scope):

if arbol is None or type(arbol) is int: return

if arbol[0] == 'programa':

semantica\_1(arbol[1],scope)

elif arbol[0] == 'declaration list':

for x in range(1,len(arbol)):

semantica\_1(arbol[x],scope)

elif arbol[0] == 'declaration':

semantica\_1(arbol[1],scope)

elif arbol[0] == 'fun declaration':

global actual\_scope

actual\_scope +=1

semantica\_1(arbol[6], actual\_scope)

elif arbol[0] == 'compound stmt':

for x in range(1,len(arbol)):

semantica\_1(arbol[x],scope)

elif arbol[0] == 'statement list':

for x in range(1,len(arbol)):

semantica\_1(arbol[x],scope)

elif arbol[0] == 'statement':

semantica\_1(arbol[1],scope)

elif arbol[0] == 'expression stmt':

semantica\_1(arbol[1],scope)

elif arbol[0] == 'expression':

# debemos de checar que la variable sea signada a un valor int o arr

if len(arbol) == 4:

if arbol[1][1] in tabla\_simbolos[scope] and tabla\_simbolos[scope][arbol[1][1]] == 'int':

if checkType(arbol[3],arbol[1][1],scope):

return True

else:

print("Error:")

print(f"El valor de asignacion {arbol[1][1]}, no corresponde al de la var {tabla\_simbolos[scope][arbol[1][1]]}")

quit()

else:

semantica\_1(arbol[1],scope)

elif arbol[0] == 'simple expression':

if len(arbol) == 2:

checkFun(arbol[1],scope)

def checkType(xpr, var1, scope):

if xpr[0] == 'expression':

return checkType(xpr[1],var1,scope)

elif xpr[0] == 'simple expression':

for x in range(1,len(xpr)):

return checkType(xpr[x],var1,scope)

elif xpr[0] == 'additive expression':

# checamos que los dos valores de al suma sean int

for x in range(1,len(xpr)):

return checkType(xpr[x],var1,scope)

elif xpr[0] == 'term':

# checamos que los dos valores sean int

for x in range(1,len(xpr)):

return checkType(xpr[x],var1,scope)

elif xpr[0] == 'factor':

# verificamos que var1 y xpr sean int

if type(xpr[1]) is int:

if var1 in tabla\_simbolos[scope]:

if tabla\_simbolos[scope][var1] == 'int':

return True

elif var1 in tabla\_simbolos[0]:

if tabla\_simbolos[0][var1] == 'int':

return True

else:

print("Error:")

print(f"variable no exite \"{xpr[1]}\"")

quit()

else:

if len(xpr) == 4:

return checkType(xpr[2],var1,scope)

else:

return checkType(xpr[1],var1,scope)

elif xpr[0] == 'call':

if xpr[1] in tabla\_simbolos[0]:

aux = tabla\_simbolos[0][xpr[1]].split(",")

if aux[0] == tabla\_simbolos[scope][var1]:

return True

else:

print("Error:")

print(f" Variables no corresponden \"{aux[0]} y {tabla\_simbolos[scope][var1]}\" ")

quit()

else:

print("Error:")

print(f" Metodo no declarado \"{xpr[1]}\" ")

quit()

elif xpr[0] == 'var':

if xpr[1] in tabla\_simbolos[scope]:

if tabla\_simbolos[scope][xpr[1]] == tabla\_simbolos[scope][var1]:

return True

else:

# Imprimimos Error

return False

elif xpr[1] in tabla\_simbolos[0]:

if tabla\_simbolos[0][xpr[1]] == tabla\_simbolos[0][var1]:

return True

else:

# Imprimimos Error

return False

else:

print("No encontrada")

return False