

Examen Final

DOCENTE	CARRERA	CURSO
MSc. Vicente Enrique Machaca Arceda	Escuela Profesional de Ingeniería de Software	Compiladores

1. Datos de los estudiantes

- Grupo: 1
- Git Hub: <https://github.com/CrazyDani17/FinalCompiladores>
- Integrantes:
 - Guillermo Aleman
 - Marvik Del Carpio
 - Daniel Mendiguri
 - Daniela Vilchez

2. Informe

1. Introducción: Detallen la motivación del lenguaje propuesto y una breve descripción.

La programación está inmersa en todo, y cada vez se hace más necesario tener idea de qué es, o cómo se trabaja, para ello en muchas currículas de formación básica se incluyen cursos referidos, y las asignaturas de computación dejaron de tener un enfoque específicamente ofimático. Ante la necesidad de que el programar esté al alcance de los que no son especialistas ni apuntan a ello, pero requieren entender cómo se hace; y además para aquellos que buscan despertar el interés en este campo nace Llama.

La idea de este proyecto se origina en el Curso de Compiladores de la Universidad La Salle, con el fin de encontrar una forma de integrar a este campo especialmente a niños y adolescentes a través de despertar su interés por medio de lo simple, para ello nos concentramos en lograr lenguaje sencillo de entender para latino-hablantes, uno sin exceso de reglas que se concentrase en dar la lógica con la sintaxis y estructuras de control elementales.

Para poder definir el nombre, se observó la particularidad de lenguajes como python, herramientas de programación como anaconda, spyder o mysql. Todas ellas refirieron a un animal; entonces rápidamente se pensó en uno característico de Perú, y que además pueda tener sentido respecto a la operabilidad del lenguaje.

Llama es el nombre definido, y tomó un mayor sentido cuando se inició la construcción del lenguaje. Llama se desarrolla a partir de funciones secundarias las cuales pueden ejecutar sentencias e interactuar entre sí, para luego ser llamadas finalmente por la función principal "llama".

2. Lenguaje de programación LLAMA



3. Especificación léxica: Describa cada token y muestre las expresiones regulares.

a) Definición de los comentarios.

Comentarios en bloque: `~texto~`

b) Definición de los identificadores. Los identificadores deben empezar con una letra, no está permitido que un identificador empiece con un número o un subguión.

Permite:

- Los caracteres de la “A” a la “Z”.
- Los caracteres de la “a” a la “z”.
- Números y subguión.

c) Definición de las palabras clave.

si (if)
sino (else)
pinocho (bool)
numero (int)
decimal (double)
mostrar (print)
texto (string)
descansito (break)
yoyo (while)
devuelve (return)
chacha (char)

d) Definición de los literales.

- Literales enteros
1
- Literales entero flotante
2.0
- Literales booleanos
verdad, mentira
- Literales caracteres

```
nueva linea \n  
tabulador \t
```

- Literales string

```
string vacio ""  
string "asdasda"
```

e) Definición de los operadores. Los siguientes caracteres se utilizan en el código fuente como operadores:

+ - * / & %

Las siguientes combinaciones de caracteres se utilizan como operadores:

== <= >= <> #y #o

f) Expresión regular de cada componente léxico (en una tabla).

Componente Léxico	Expresión Regular
Comentario en Bloque	<code>\ ~ (\w \W)*\ ~</code>
Identificadores	<code>[a-zA-Z][\w]*</code>
String Vacio	<code>""</code>
Literal Entero	<code>0 [1-9][0-9]*</code>
Literal Flotante	<code>(0 [1-9][0-9]*)\.[0-9]*</code>
Literal Booleano	<code>verdad mentira</code>
Nueva Linea	<code>\n</code>
Tabulador	<code>\t</code>
numero	<code>"numero"</code>
texto	<code>"texto"</code>
decimal	<code>"decimal"</code>
pinocho	<code>"pinocho"</code>
chacha	<code>"chacha"</code>
misión	<code>"mision"</code>
si	<code>"si"</code>
yoyo	<code>"yoyo"</code>
verdad	<code>"verdad"</code>
mentira	<code>"mentira"</code>
devuelve	<code>"devuelve"</code>
sino	<code>"sino"</code>
+	<code>" + "</code>
-	<code>" - "</code>
*	<code>" * "</code>
<code>#y</code>	<code>"#y"</code>
<code>#o</code>	<code>"#o"</code>
/	<code>" / "</code>
>	<code>" > "</code>
<	<code>" < "</code>
(<code>" ("</code>
)	<code>") "</code>
[<code>" ["</code>
]	<code>"] "</code>
,	<code>" , "</code>
.	<code>" . "</code>
=	<code>" = "</code>
==	<code>" == "</code>
<=	<code>" <= "</code>
>=	<code>" >= "</code>
<>	<code>" <> "</code>
%	<code>" % "</code>

4. Gramática: Muestre la gramática. Para comprobar si la gramática está bien, puede utilizar esta herramienta (la gramática no debe ser ambigua y debe estar factorizada por la izquierda).

```

'''
Inicio → E FuncionPrincipal K
Inicio → FuncionPrincipal
E → DeclaracionFuncion E'
E' → DeclaracionFuncion E'
E' → ''
DeclaracionFuncion → mision identificador pizquierdo Parametros pderecho lizquierdo
    CuerpoF lderecho
FuncionPrincipal → llama pizquierdo pderecho lizquierdo Cuerpo lderecho
K → ''
K → E
Parametros → ''
Parametros → Y Y'
Y' → coma Y Y'
Y' → ''
Y → TipoDato identificador C
TipoDato → pinocho
TipoDato → numero
TipoDato → decimal
TipoDato → texto
TipoDato → chacha
C → cizquierdo cderecho
C → ''
CuerpoF → Cuerpo J
J → ''
J → devuelve Expresion
Cuerpo → ''
Cuerpo → DeclaracionVariable D'
Cuerpo → Sentencias D'
D' → DeclaracionVariable D'
D' → Sentencias D'
D' → ''
DeclaracionVariables → DeclaracionVariable Dc'
Dc' → DeclaracionVariable Dc'
Dc' → ''
Sentencias → lizquierdo MuchasSentencias lderecho
Sentencias → si pizquierdo Expresion pderecho Sentencias sino Sentencias
Sentencias → yoyo pizquierdo Expresion pderecho lizquierdo MuchasSentenciasYoyo lderecho
Sentencias → mostrar pizquierdo Expresion pderecho
Sentencias → identificador OPS
OPS → igual Expresion
OPS → cizquierdo Expresion cderecho igual Expresion
OPS → pizquierdo ParaLLamados pderecho

MuchasSentenciasYoyo → SentenciasYoyo M'
M' → SentenciasYoyo M'
M' → ''
MuchasSentenciasYoyo → ''

SentenciasYoyo → lizquierdo MuchasSentenciasYoyo lderecho
SentenciasYoyo → si pizquierdo Expresion pderecho SentenciasYoyo sino SentenciasYoyo
SentenciasYoyo → yoyo pizquierdo Expresion pderecho lizquierdo MuchasSentenciasYoyo
    lderecho

```

```
SentenciasYoyo → mostrar pizquierdo Expresion pderecho
SentenciasYoyo → identificador OPS
SentenciasYoyo → descansito

MuchasSentencias → Sentencias S'
MuchasSentencias → DeclaracionVariable S'
S' → DeclaracionVariable S'
S' → Sentencias S'
S' → ''
DeclaracionVariable → TipoDato identificador Corchetes OPI
Corchetes → cizquierdo Expresion cderecho
Corchetes → ''
OPI → igual Expresion
OPI → ''
Expresion → Ex
Ex → Tx Ex'
Ex' → Simbolo Tx Ex'
Ex' → ''
Tx → Literal
Literal → lpinocho
Literal → lnumero
Literal → ltexto
Literal → ldecimal
Literal → lchacha
Tx → identificador Op
Op → ''
Op → cizquierdo Expresion cderecho
Op → pizquierdo ParaLLamados pderecho
ParaLLamados → ''
ParaLLamados → PL PL'
PL' → coma PL PL'
PL' → ''
PL → Expresion
Tx → verdad
Tx → mentira
Tx → pizquierdo Expresion pderecho
Simbolo → mas
Simbolo → menos
Simbolo → por
Simbolo → o
Simbolo → y
Simbolo → mayor_igual
Simbolo → diferente
Simbolo → menor_igual
Simbolo → comparacion
Simbolo → menor
Simbolo → mayor
Simbolo → modulo
Simbolo → dividir
Simbolo → igual

,,,
```

5. Analizador léxico: Incluya ejemplos de código correctos y con errores léxicos (del lenguaje propuesto). Muestre el código (implementación) y capturas de pantalla de la ejecución del mismo.

```
# -----  
# tokenizer for a simple expression evaluator for  
# numbers and +,-,*,/  
# -----  
import ply.lex as lex  
# List of token names. This is always required  
tokens =  
    ('mision', 'si', 'sino', 'pinocho', 'numero', 'lnumero', 'decimal', 'ldecimal', 'mostrar',  
     'texto', 'ltexto', 'descansito', 'yoyo', 'devuelve', 'chacha', 'lchacha', 'mas', 'menos', 'por',  
     'y', 'o', 'dividir', 'mayor', 'menor', 'pizquierdo', 'pderecho', 'cizquierdo', 'cderecho', 'coma',  
     'igual', 'menor_igual', 'mayor_igual', 'diferente', 'identificador', 'lizquierdo', 'lderecho',  
     'division', 'verdad', 'mentira', 'modulo', 'llama', 'comparacion')  
  
# Regular expression rules for simple tokens  
t_mision = r'mision'  
t_si = r'si'  
t_sino = r'sino'  
t_pinocho = r'pinocho'  
t_numero = r'numero'  
t_decimal = r'decimal'  
t_mostrar = r'mostrar'  
t_texto = r'texto'  
t_descansito = r'descansito'  
t_yoyo = r'yoyo'  
t_devuelve = r'devuelve'  
t_chacha = r'chacha'  
t_llama = r'llama'  
t_comparacion = r'\=\  
  
t_mas = r'\+'  
t_menos = r'\-'  
t_por = r'\*'  
t_dividir = r'\/'  
t_modulo = r'\%'  
  
t_y = r'\#y'  
t_o = r'\#o'  
  
t_mayor = r'\>'  
t_menor = r'\<'  
  
t_pizquierdo = r'\('  
t_pderecho = r'\)'  
  
t_lizquierdo = r'\{'  
t_lderecho = r'\}'  
  
t_cizquierdo = r'\['  
t_cderecho = r'\]'  
  
t_coma = r'\,'  
t_igual = r'\='  
t_menor_igual = r'\<='
```

```
t_mayor_igual = r'\>='
t_diferente = r'\<|>'

def t_identificador(t):
    r'(?!(decimal|si|sino|pinocho|numero|mostrar|texto|descansito|yoyo|devuelve|chacha|
    mision|mentira|verdad|llama)[a-zA-Z]+[\w]*)'
    try:
        t.value = t.value
    except ValueError:
        t.value = 0
    return t

# A regular expression rule with some action code
def t_mentira(t):
    r'mentira'
    t.value = False # guardamos el valor del lexema
    return t

def t_verdad(t):
    r'verdad'
    t.value = True # guardamos el valor del lexema
    return t

def t_lnumero(t):
    r'\d+(?!\.|\.|e|E)'
    try:
        t.value = int(t.value) # guardamos el valor del lexema
    except ValueError:
        t.value = 0
    return t

def t_ldecimal(t):
    r'(0|[1-9][0-9]*)\.[0-9]*'
    try:
        t.value = float(t.value)
    except ValueError:
        t.value = 0
    return t

def t_lchacha(t):
    r'\('W|w)\)'
    t.value = t.value # guardamos el valor del lexema
    return t

def t_ltexto(t):
    r'"(\W|w)+"'
    t.value = t.value # guardamos el valor del lexema
    return t

def t_comments(t):
    r'\^(\W|W)*\^'

# Define a rule so we can track line numbers
def t_newline(t):
```



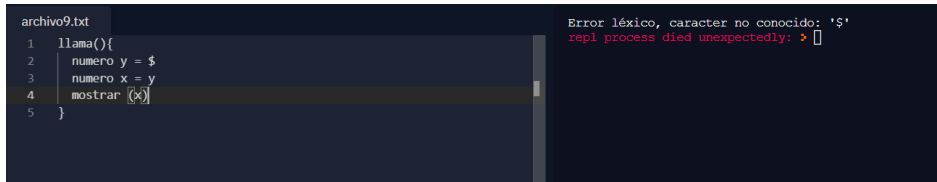
```

r'\n+'
t.lexer.lineno += len(t.value)
# A string containing ignored characters (spaces and tabs)
t_ignore = ' \t'
# Error handling rule
def t_error(t):
    print("Error léxico, caracter no conocido: '%s'" % t.value[0])
    t.lexer.skip(1)
    raise SystemExit
# Build the lexer
lexer = lex.lex()
# Test it out

file=open('archivo9.txt','r')
texto=file.readlines()
file.close()
data = texto
tokens=[]
tokens_info=[]
for renglon in texto:
    # Give the lexer some input
    lexer.input(renglon)
    # Tokenize
    while True:
        tok = lexer.token()
        if not tok:
            break # No more input
        tokens.append(tok.type)
        tokens_info.append(tok)
        #print(tok)
        #print(tok.type, tok.value, tok.lineno, tok.lexpos)

```

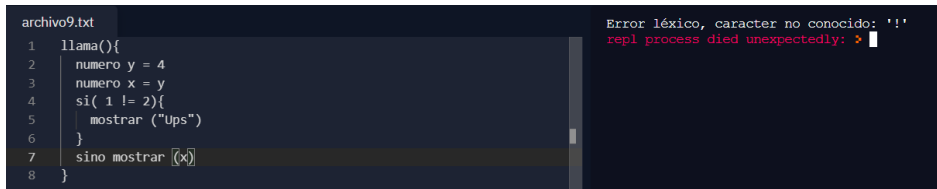
■ Ejemplos de errores léxicos:



```

archivo9.txt
1 llama(){
2     numero y = $
3     numero x = y
4     mostrar (x)
5 }
Error léxico, caracter no conocido: '$'
repl process died unexpectedly: ✖

```

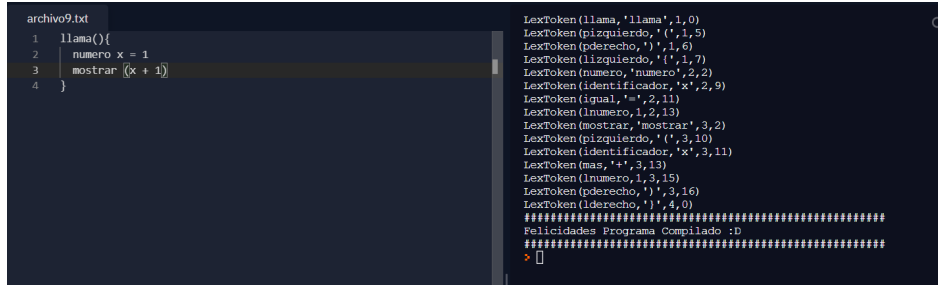


```

archivo9.txt
1 llama(){
2     numero y = 4
3     numero x = y
4     si( 1 != 2){
5         mostrar ("Ups")
6     }
7     sino mostrar (x)
8 }
Error léxico, caracter no conocido: '!'
repl process died unexpectedly: ✖

```

- Funcionamiento de analizador léxico (Ejemplo correcto):



```

archivo9.txt
1  llama(){
2      numero x = 1
3      mostrar (x + 1)
4  }

LexToken(llama,'llama',1,0)
LexToken(pizquierdo,'(',1,5)
LexToken(pderecho,')',1,6)
LexToken(ilizquierdo,'{',1,7)
LexToken(numero,'numero',2,2)
LexToken(identificador,'x',2,9)
LexToken(igual,'=',2,11)
LexToken(lnumero,1,2,13)
LexToken(mostrar,'mostrar',3,2)
LexToken(pizquierdo,'(',3,10)
LexToken(identificador,'x',3,11)
LexToken(mas,'+',3,13)
LexToken(lnumero,1,3,15)
LexToken(pderecho,')',3,16)
LexToken(iderecho,'}',4,0)
#####
Felicitades Programa Compilado :D
#####
>

```

- Analizador sintáctico: Incluya ejemplos de código correctos y con errores sintácticos (del lenguaje propuesto). Muestre el código (implementación) y capturas de pantalla de la ejecución del mismo.

```

import analizador_lexico
import xlrd
import pandas as pd
import numpy as np
from graphviz import Digraph
import arbol
import math
dot = Digraph()
filas = {"Inicio" :1 ,
"E" :2 ,
"E'" :3 ,
"DeclaracionFuncion" :4 ,
"FuncionPrincipal" :5 ,
"K" :6 ,
"Parametros" :7 ,
"Y'" :8 ,
"Y" :9 ,
"TipoDato" :10 ,
"C" :11 ,
"CuerpoF" :12 ,
"J" :13 ,
"Cuerpo" :14 ,
"D'" :15 ,
"DeclaracionVariables" :16 ,
"Dc'" :17 ,
"Sentencias" :18 ,
"OPS" :19 ,
"MuchasSentenciasYoyo" :20 ,
"M'" :21 ,
"SentenciasYoyo" :22 ,
"MuchasSentencias" :23 ,
"S'" :24 ,
"DeclaracionVariable" :25 ,
"Corchetes" :26 ,
"OPI" :27 ,
"Expresion" :28 ,
"Ex" :29 ,
"Ex'" :30 ,
"Tx" :31 ,

```

```
"Literal" :32 ,
"Op" :33 ,
"ParaLLamados" :34 ,
"PL'" :35 ,
"PL" :36 ,
"Simbolo" :37
}
columnas={"mision" :1 ,
"identificador" :2 ,
"pizquierdo" :3 ,
"pderecho" :4 ,
"lizquierdo" :5 ,
"lderecho" :6 ,
"llama" :7 ,
"coma" :8 ,
"pinocho" :9 ,
"numero" :10 ,
"decimal" :11 ,
"texto" :12 ,
"chacha" :13 ,
"cizquierdo" :14 ,
"cderecho" :15 ,
"devuelve" :16 ,
"si" :17 ,
"sino" :18 ,
"yoyo" :19 ,
"mostrar" :20 ,
"igual" :21 ,
"descansito" :22 ,
"lpinocho" :23 ,
"lnumero" :24 ,
"ltexto" :25 ,
"ldecimal" :26 ,
"lchacha" :27 ,
"verdad" :28 ,
"mentira" :29 ,
"mas" :30 ,
"menos" :31 ,
"por" :32 ,
"o" :33 ,
"y" :34 ,
"mayor_igual" :35 ,
"diferente" :36 ,
"menor_igual" :37 ,
"comparacion" :38 ,
"menor" :39 ,
"mayor" :40 ,
"modulo" :41 ,
"dividir" :42 ,
"$" :43
}
df = pd.read_excel("tablita.xlsx", 'Hoja1', header=None)
tablita_parse = df.values

pila = ["Inicio","$"]
entrada = analizador_lexico.tokens
```

```
entrada.append("$")
pila_tokens = analizador_lexico.tokens_info
linea_tokens = analizador_lexico.tokens_info.copy()

continuar=True
i=0
j=0
p=0
aux=[]
pendientes=[]
arbolito = arbol.Arbol(pila[0],0)
dot.node(str(j), pila[0])
padres=[0,-1]
#print("Inicio:")
#print(pila)
#print(entrada)

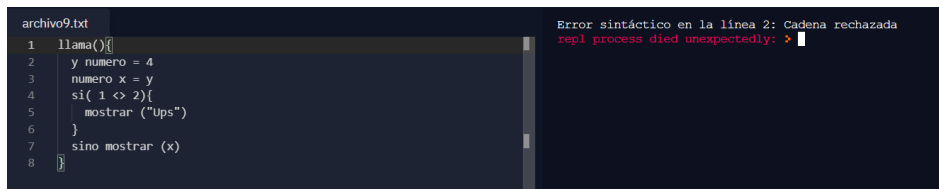
while continuar:
    #print("Vuelta",i)
    if pila[0]=="$" and entrada[0]=="$":
        continuar=False
        #print(pila)
        #print(entrada)
        #print("Cadena aceptada")
    elif pila[0] == entrada[0]:
        #print(pila)
        #print(entrada)
        pila = pila[1:]
        entrada.pop(0)
        linea_tokens.pop(0)
    elif pila[0][0]==(pila[0][0]).lower() and entrada[0][0]==(entrada[0][0]).lower():
        continuar=False
        #print(pila)
        #print(entrada)
        print("Error sintctico en la lnea " + str(linea_tokens[0].lineno) + ": Cadena
              rechazada")
        raise SystemExit
    else:
        if entrada[0] in columnas:
            reemplazo = tablita_parse[filas[pila[0]]][columnas[entrada[0]]]
            p=int(padres[0])
        else:
            continuar=False
            #print(pila)
            #print(entrada)
            print("Error sintctico en la lnea " + str(linea_tokens[0].lineno) + ": Cadena
                  rechazada")
            raise SystemExit
            break
        if reemplazo == "vacio":
            j=j+1
            h=j
            dot.node(str(h), "'')")
            dot.edge(str(p),str(h))
            arbolito.agregarhijo(arbolito,"'",p,h)
            pila=pila[1:]
```

```

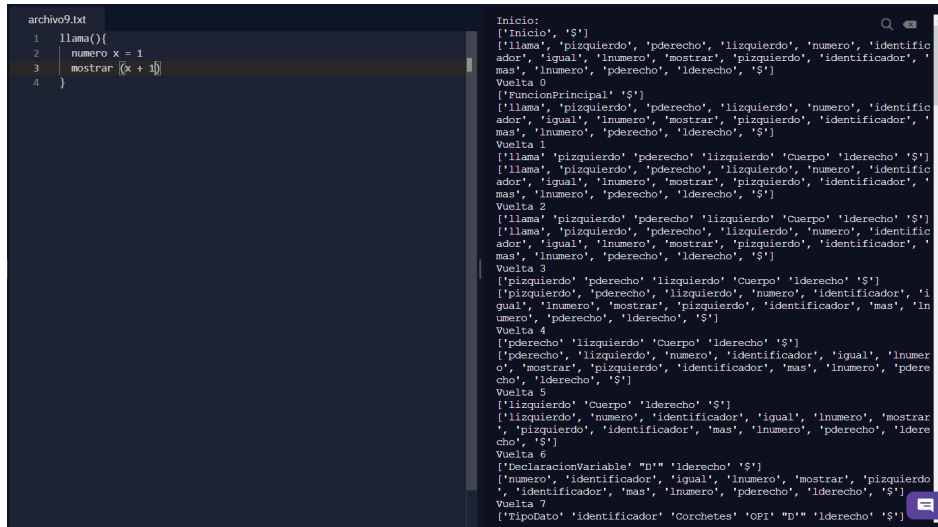
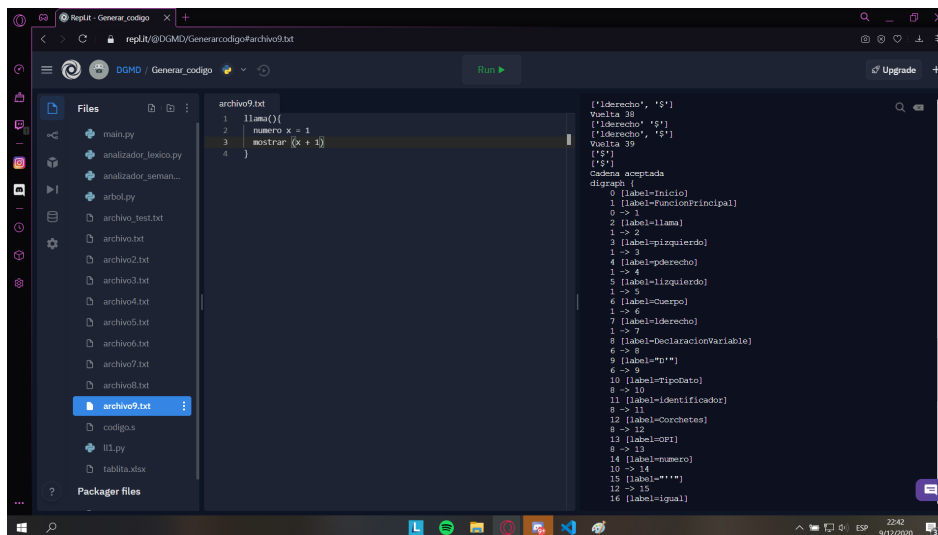
padres=padres[1:]
#print(pila)
#print(entrada)
else:
    if str(reemplazo) == 'nan':
        print("Error sintctico en la lnea " + str(linea_tokens[0].lineno) + ": Cadena
            rechazada")
        raise SystemExit
    array_aux=reemplazo.split()
    pila = np.concatenate((array_aux,pila[1:]),axis=0)
    hijos = len(array_aux)
    aux.clear()
    for e in range(hijos):
        j=j+1
        if array_aux[e][0]==array_aux[e][0].upper():
            aux.append(j)
            arbolito.agregarhijo(arbolito,array_aux[e],p,j)
        else:
            arbolito.agregarhijo(arbolito,array_aux[e],p,j)
        h=j
        dot.node(str(h), array_aux[e])
        dot.edge(str(p),str(h))
    padres = np.concatenate((aux,padres[1:]),axis=0)
    #print(pila)
    #print(entrada)
    i=i+1
#print(dot.source)
arbol.recorrerArbol(arbolito,pila_tokens)

```

■ Ejemplos de errores sintáctico:

- Funcionamiento de analizador sistántico:

- Analizador semántico: Incluya ejemplos de código correctos y con errores semánticos (del lenguaje propuesto). Muestre el código (implementación) y capturas de pantalla de la ejecución del mismo.

```
import l11
class simbolo:
    def __init__(self,t,lex,tp,pos,lin,scope,td=None):
        self.token=t
        self.lexema=lex
        self.tipo=tp
        self.posicion=pos
        self.linea=lin
        self.scope=scope
        self.tipo_dato=td
```

```
def buscar_simbolo_td(tabla,elemento):
    for i in tabla:
        if i.lexema==elemento and (i.tipo == "variable" or i.tipo == "mision"):
            return i.tipo_dato

def buscar_simbolo_tipo(tabla,elemento):
    for i in tabla:
        if i.lexema==elemento and (i.tipo == "variable" or i.tipo == "mision"):
            return i.tipo

def imprimir_tds(tabla):
    for i in tabla:
        print(i.token,i.lexema,i.tipo,i.posicion,i.linea,i.scope)

def buscar_mision(arbol,i):
    if arbol.elemento == "DeclaracionFuncion" and arbol.hijos[1].token.value == i:
        return arbol
    for subarbol in arbol.hijos:
        arbolBuscado = buscar_mision(subarbol, i)
        if (arbolBuscado != None):
            return arbolBuscado
    return None

def obtener_tipo_mision(arbol):
    if arbol.hijos[6].hijos[1].hijos[0].elemento == "''":
        return None
    else:
        tipo = verificar_tipo_Ex(arbol.hijos[6].hijos[1].hijos[1].hijos[0])
        return tipo

n=0
es_yoyo=False
tabla_de_simbolos=[]
#lll1.arbol.imprimirArbol(lll1.arbolito)
funcion_actual=None
funcion_actual_aux=None
errores = False

def verificar_tipo_de_dato_id(node_tx):
    tipo_de_dato = buscar_simbolo_td(tabla_de_simbolos,node_tx.hijos[0].token.value)
    return tipo_de_dato

def verificar_tipo_de_termino(node_t):
    tipo_de_dato = None
    if node_t.elemento == "lpinocho":
        tipo_de_dato = "pinocho"
    elif node_t.elemento == "lnumero":
        tipo_de_dato = "numero"
    elif node_t.elemento == "ltexto":
        tipo_de_dato = "texto"
    elif node_t.elemento == "ldecimal":
        tipo_de_dato = "decimal"
    elif node_t.elemento == "lchacha":
        tipo_de_dato = "chacha"
    return tipo_de_dato
```

```
def verificar_tipo_Ex_prima(node_Ex_prima):
    global errores
    tipo_de_dato_Ex_prima = "None"
    if node_Ex_prima.hijos[0].elemento=="'":
        tipo_de_dato_Ex_prima = "None"
    elif node_Ex_prima.hijos[0].elemento=="Simbolo":
        tipo_de_dato_T = verificar_tipo_Tx(node_Ex_prima.hijos[1])
        tipo_de_dato_Ex_prima = verificar_tipo_Ex_prima(node_Ex_prima.hijos[2])

    if tipo_de_dato_Ex_prima == "None":
        tipo_de_dato_Ex_prima = tipo_de_dato_T
    else:
        if tipo_de_dato_T == tipo_de_dato_Ex_prima:
            tipo_de_dato_Ex_prima = tipo_de_dato_T
        else:
            print("Error de asigancin 3 en la linea " +
                  str(node_Ex_prima.hijos[1].hijos[0].hijos[0].token.lineno) + ": los tipos no
                  coinciden")
            errores = True
    return tipo_de_dato_Ex_prima

def verificar_tipo_Tx(node_Tx):
    tipo_de_dato=None
    if node_Tx.hijos[0].elemento == "Literal":
        tipo_de_dato = verificar_tipo_de_termino(node_Tx.hijos[0].hijos[0])
    elif node_Tx.hijos[0].elemento == "identificador":
        tipo_de_dato = verificar_tipo_de_dato_id(node_Tx)
    return tipo_de_dato

def verificar_tipo_Tx_Ex_prima(node_Tx, node_Ex_prima):
    global errores
    type_Tx = verificar_tipo_Tx(node_Tx)
    type_Ex_prima= verificar_tipo_Ex_prima(node_Ex_prima)
    if str(type_Ex_prima) == "None":
        return type_Tx
    else:
        if type_Tx == type_Ex_prima:
            return type_Tx
        else:
            print("Error de asigancin 2 en la linea " +
                  str(node_Tx.hijos[0].hijos[0].token.lineno) + ": los tipos no coinciden")
            errores = True

def verificar_tipo_Ex(arbol):
    tipo_de_dato = verificar_tipo_Tx_Ex_prima(arbol.hijos[0],arbol.hijos[1])
    return tipo_de_dato

def comprobar_duplicado(valor,linea):
    global errores
    for x in reversed(tabla_de_simbolos):
        if x.lexema==valor and (x.tipo=="mision" or x.tipo == "variable"):
            print("Error error semntico en linea "+ str(linea) + ": " + x.lexema + " ya fue
                  declarado")
```



```
errores = True
break

def comprobar_existencia(valor, tipo, linea):
    global errores
    encontrado=False
    for x in reversed(tabla_de_simbolos):
        if x.lexema==valor and x.tipo==tipo:
            encontrado=True
            break
    if encontrado==False:
        print("Error semntico en lnea " + str(linea) + ": Llamaste a la " + tipo + " " +
              valor + ", pero no fue declarada")
        errores = True

def verificar_declaracion_variable(arbol):
    global errores
    comprobar_duplicado(arbol.hijos[1].token.value, arbol.hijos[1].token.lineno)

    #####:#####

    tabla_de_simbolos.append(simbolo(arbol.hijos[1].token, arbol.hijos[1].token.value,
    "variable", arbol.hijos[1].token.lexpos, arbol.hijos[1].token.lineno, funcion_actual,
    arbol.hijos[0].hijos[0].elemento))
    #####:#####

    if arbol.hijos[3].hijos[0].elemento == "igual":
        td_asignacion = verificar_tipo_Ex(arbol.hijos[3].hijos[1].hijos[0])
        td_variable = buscar_simbolo_td(tabla_de_simbolos, arbol.hijos[1].token.value)
        if td_variable != td_asignacion:
            print ("Error de asignacin en la lnea " + str(arbol.hijos[1].token.lineno) + ": no
            coinciden los tipos")
            errores = True
        else:
            arbol.hijos[3].hijos[1].tipo=td_asignacion

def obtener_parametros_lmision(arbol, par):
    if arbol.elemento=="PL":
        tipo = verificar_tipo_Ex(arbol.hijos[0].hijos[0])
        par.append(tipo)
        arbol.hijos[0].tipo=tipo
    for x in arbol.hijos:
        obtener_parametros_lmision(x, par)

def obtener_parametros(arbol, par):
    if arbol.elemento=="Y":
        par.append(arbol.hijos[0].hijos[0].elemento)
    for x in arbol.hijos:
        obtener_parametros(x, par)

def insertar_parametros(arbol):
    if arbol.elemento=="Y":
        tabla_de_simbolos.append(simbolo(arbol.hijos[1].token, arbol.hijos[1].token.value,
        "variable", arbol.hijos[1].token.lexpos, arbol.hijos[1].token.lineno, funcion_actual,
        arbol.hijos[0].hijos[0].elemento))
```

```
for x in arbol.hijos:
    insertar_parametros(x)

def verificar_mision(arbol):
    global funcion_actual
    global funcion_actual_aux
    comprobar_duplicado(arbol.hijos[1].token.value, arbol.hijos[1].token.lineno)
    funcion_actual=arbol.hijos[1].token.value
    funcion_actual_aux=funcion_actual
    if arbol.hijos[3].hijos[0]!="'':
        insertar_parametros(arbol.hijos[3])
    tipo_de_mision=obtener_tipo_mision(arbol)
    arbol.hijos[1].tipo = tipo_de_mision
    tabla_de_simbolos.append(simbolo(arbol.hijos[1].token, arbol.hijos[1].token.value, "mision",
    arbol.hijos[1].token.lexpos, arbol.hijos[1].token.lineno, "global", tipo_de_mision))

def verificar_asignacion_de_variable(arbol):
    global errores
    comprobar_existencia(arbol.hijos[0].token.value, "variable", arbol.hijos[0].token.lineno)

    td_asignacion = verificar_tipo_Ex(arbol.hijos[1].hijos[1].hijos[0])
    td_variable = buscar_simbolo_td(tabla_de_simbolos, arbol.hijos[0].token.value)

    if td_variable != td_asignacion:
        print ("Error de asignacin en la linea " + str(arbol.hijos[0].token.lineno) + ": no
            coinciden los tipos")
        errores = True
    else:
        arbol.hijos[1].hijos[1].tipo=td_asignacion

    tabla_de_simbolos.append(simbolo(arbol.hijos[0].token, arbol.hijos[0].token.value,
    "asignacion", arbol.hijos[0].token.lexpos, arbol.hijos[0].token.lineno, funcion_actual))

def verificar_llamada_de_variable(arbol):
    comprobar_existencia(arbol.hijos[0].token.value, "variable", arbol.hijos[0].token.lineno)

def verificar_llamada_de_mision(arbol):
    global errores
    parametros=[]
    parametros_lmision=[]
    comprobar_existencia(arbol.hijos[0].token.value, "mision", arbol.hijos[0].token.lineno)
    nodo_mision=buscar_mision(111, arbolito, arbol.hijos[0].token.value)
    obtener_parametros(nodo_mision.hijos[3], parametros)
    obtener_parametros_lmision(arbol.hijos[1].hijos[1], parametros_lmision)
    if len(parametros) != len(parametros_lmision):
        print("Error semntico en la linea " + str(arbol.hijos[0].token.lineno) + ": los
            parmentros no coinciden")
        errores = True
    else:
        for ra in range(len(parametros)):
            if parametros[ra] != parametros_lmision[ra]:
                print("Error semntico en la linea " + str(arbol.hijos[0].token.lineno) + ": los
                    parametros no coinciden")
                errores = True

def verificar_mostrar(arbol):
```

```
td_asignacion = verificar_tipo_Ex(arbol.hijos[2].hijos[0])
arbol.hijos[2].tipo=td_asignacion

def eliminar_scope(scope):
    #imprimir_tds(tabla_de_simbolos)
    #print("")
    for item in reversed(tabla_de_simbolos):
        if item.scope == scope:
            tabla_de_simbolos.pop(tabla_de_simbolos.index(item))
    #imprimir_tds(tabla_de_simbolos)
    #print("")

def verificar_scope_llama(arbol):
    for subarbol in arbol.hijos:
        if subarbol.elemento=="FuncionPrincipal":
            verificar_scope(subarbol)

def verificar_scope(arbol):
    global funcion_actual
    global n
    global funcion_actual_aux
    global es_yoyo

    if arbol.elemento == "FuncionPrincipal":
        funcion_actual="llama"
        funcion_actual_aux="llama"

    if arbol.elemento == "yoyo":
        es_yoyo=True

    if len(arbol.hijos)>0 and arbol.hijos[0].elemento=="mision":
        n=0
        verificar_mision(arbol)

    if arbol.elemento == "DeclaracionVariable":
        verificar_declaracion_variable(arbol)

    if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
        arbol.hijos[0].elemento=="identificador"and
        arbol.hijos[1].hijos[0].elemento=="igual":
        verificar_asignacion_de_variable(arbol)

    if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias" or
        arbol.elemento == "Tx") and arbol.hijos[0].elemento=="identificador"and
        arbol.hijos[1].hijos[0].elemento=="pizquierdo":
        verificar_llamada_de_mision(arbol)

    if arbol.elemento == "Tx" and arbol.hijos[0].elemento == "identificador" and
        arbol.hijos[1].hijos[0].elemento == "''":
        verificar_llamada_de_variable(arbol)

    if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
        arbol.hijos[0].elemento == "mostrar":
        verificar_mostrar(arbol)
```

```

if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
    arbol.hijos[0].elemento=="lizquiedo":
    n=n+1
    funcion_actual= funcion_actual_aux + str(n)

if arbol.elemento == "lderecho" and n==0:
    if es_yoyo==True:
        es_yoyo=False
    else:
        eliminar_scope(funcion_actual)

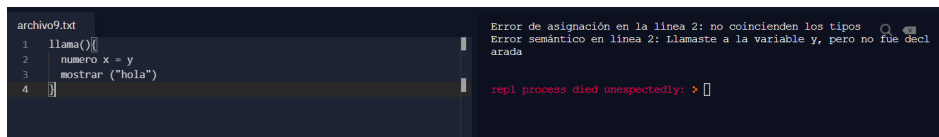
if arbol.elemento == "lderecho" and n>0:
    if es_yoyo==True:
        es_yoyo=False
    else:
        n=n-1
        eliminar_scope(funcion_actual)
    if n!=0:
        funcion_actual= funcion_actual_aux + str(n)
    else:
        funcion_actual= funcion_actual_aux

for subarbol in arbol.hijos:
    if subarbol.elemento!="FuncionPrincipal":
        verificar_scope(subarbol)

verificar_scope(ll1.arbolito)
#print("Scope Llama")
verificar_scope_llama(ll1.arbolito)
#print("Final")
#imprimir_tds(tabla_de_simbolos)
if errores == True:
    raise SystemExit

```

■ Ejemplos de errores semántico:




```

archivo9.txt
1  mision saludar(numero x, numero y){
2      numero z=1+5
3      {
4          y=1+x
5      }
6      devuelve z
7  }
8  llama(){
9      despedir()
10     saludar()
11     numero x=1
12     x=1
13     yoyo(x<-1){
14         x=2
15     }
16     {
17         x=2
18     }
19     mostrar (x)
20 }
21 mision despedir(){
22     numero y=2
23     mostrar("Adios")
24 }

```

Error semántico en la línea 10: los parámetros no coinciden
repl process died unexpectedly: > []

- Funcionamiento de analizador semántico (Ejemplo correcto):

```

archivo9.txt
1  mision suma (numero x, numero y){
2      devuelve x + y
3  }
4  llama(){
5      numero x = suma(1,2)
6      mostrar (x + 1)
7  }

```

```

LexToken(identificador,'x',1,20) x variable 20 1 suma
LexToken(identificador,'y',1,30) y variable 30 1 suma
LexToken(identificador,'suma',1,7) suma mision 7 1 global

LexToken(identificador,'suma',1,7) suma mision 7 1 global

Scope Llama
LexToken(identificador,'suma',1,7) suma mision 7 1 global
LexToken(identificador,'x',5,9) x variable 9 5 llama

LexToken(identificador,'suma',1,7) suma mision 7 1 global

Final
LexToken(identificador,'suma',1,7) suma mision 7 1 global
#####
Felicitades Programa Compilado :D
#####
>

```

8. Generación de código: Muestra su implementación y capturas de pantalla de la ejecución del compilador.

```

import analizador_semantico

print("#####")
print ("Felicitades Programa Compilado :D")
print("#####")
data = ".data\n\nendl: .asciiz " + "'" + "\\n" + "'" + "\\n"
total = 0
parametros = []

def generar_terminal(node_t):
    if node_t.elemento == "lnumero":
        return "li $a0 " + str(node_t.token.value) + "\\n"
    elif node_t.elemento == "ldecimal":
        return "li $a0 " + str(node_t.token.value) + "\\n"

def generar_Ex_prima(node_Ex_prima):
    resultado=""
    if node_Ex_prima.hijos[0].elemento=="' '":
        return ""
    elif node_Ex_prima.hijos[0].elemento=="Simbolo":
        resultado = "sw $a0 0($sp)\naddiu $sp $sp -4\\n"
        resultado += generar_Tx(node_Ex_prima.hijos[1])

```

```
resultado += "lw $t1 4($sp)\n"
if node_Ex_prima.hijos[0].hijos[0].elemento == "mas":
    resultado += "add $a0 $t1 $a0\naddiu $sp $sp 4\n"
elif node_Ex_prima.hijos[0].hijos[0].elemento == "menos":
    resultado += "sub $a0 $t1 $a0\naddiu $sp $sp 4\n"
elif node_Ex_prima.hijos[0].hijos[0].elemento == "por":
    resultado += "mul $a0 $t1 $a0\naddiu $sp $sp 4\n"
elif node_Ex_prima.hijos[0].hijos[0].elemento == "dividir":
    #resultado += "div $a0 $t1 $a0\naddiu $sp $sp 4\n"
    resultado += "div $t1 $a0\n"
    resultado += "mflo $a0\n"
    resultado += "addiu $sp $sp 4\n"

elif node_Ex_prima.hijos[0].hijos[0].elemento == "modulo":
    resultado += "div $t1 $a0\n"
    resultado += "mfhi $a0\n"
    resultado += "addiu $sp $sp 4\n"
elif node_Ex_prima.hijos[0].hijos[0].elemento == "comparacion":
    resultado += "addiu $sp $sp 4\n"
elif node_Ex_prima.hijos[0].hijos[0].elemento == "diferente":
    resultado += "addiu $sp $sp 4\n"
resultado += generar_Ex_prima(node_Ex_prima.hijos[2])
return resultado

def obtener_id_parametro (valor):
    return (len(parametros) - parametros.index(valor)) * 4

def generar_Tx(node_Tx):
    resultado=""
    if node_Tx.hijos[0].elemento == "Literal":
        resultado = generar_terminal(node_Tx.hijos[0].hijos[0])
    elif node_Tx.hijos[0].elemento == "identificador" and
        node_Tx.hijos[1].hijos[0].elemento == "''":
        if node_Tx.hijos[0].token.value in parametros:
            resultado = "lw $a0 "+ str(obtener_id_parametro(node_Tx.hijos[0].token.value)) +
                "($fp)\n"
        else:
            resultado = "lw $a0 " + node_Tx.hijos[0].token.value + "\n"
    elif node_Tx.hijos[0].elemento == "identificador" and
        node_Tx.hijos[1].hijos[0].elemento != "''":
        resultado = generar_llamada_de_mision(node_Tx)
    return resultado

def generar_Tx_Ex_prima(node_Tx, node_Ex_prima):
    resultado = generar_Tx(node_Tx)
    resultado += generar_Ex_prima(node_Ex_prima)
    return resultado

def generar_ex(node_e):
    resultado = generar_Tx_Ex_prima(node_e.hijos[0],node_e.hijos[1])
    return resultado

def generar_variable(node):
    global data
    data += str(node.hijos[1].token.value) + ": .word 0:1\n"
```

```
if node.hijos[3].hijos[0].elemento == "''":
    return ""
else:
    return generar_ex(node.hijos[3].hijos[1].hijos[0]) + "sw $a0 " +
        str(node.hijos[1].token.value) + "\n"

def generar_asignacion_de_variable(arbol):
    codigo = generar_ex(arbol.hijos[1].hijos[1].hijos[0]) + "sw $a0 " +
        str(arbol.hijos[0].token.value) + "\n"
    return codigo

def generar_yoyo(arbol):
    codigo = "label_loop:\n"
    codigo += generar_ex(arbol.hijos[2].hijos[0])
    codigo += "beq $a0 $t1 label_exit\n"
    codigo += generar_codigo(arbol.hijos[5])
    codigo += "j label_loop\n"
    codigo += "label_exit:\n"
    return codigo

def generar_if(arbol):
    codigo = generar_ex(arbol.hijos[2].hijos[0])
    codigo += "beq $a0 $t1 label_true\n"
    codigo += "label_false:\n"
    codigo += generar_codigo(arbol.hijos[6])
    codigo += "b label_end\n"
    codigo += "label_true:\n"
    codigo += generar_codigo(arbol.hijos[4])
    codigo += "label_end:\n"
    return codigo

def generar_devuelve(arbol):
    codigo = ""
    if arbol.hijos[0].elemento != "''":
        codigo = generar_ex(arbol.hijos[1].hijos[0])
    return codigo

def generar_mostrar(arbol):
    codigo=""
    if arbol.hijos[2].tipo == "numero":
        codigo = generar_ex(arbol.hijos[2].hijos[0])
        codigo += "li $v0 1\n"
        codigo += "syscall\n"
        codigo += "li $v0 4\nla $a0 endl\nsyscall\n"
    if arbol.hijos[2].tipo == "decimal":
        codigo = generar_ex(arbol.hijos[2].hijos[0])
        #codigo += "lw $f12 $a0"
        codigo += "li $v0 2\n"
        codigo += "syscall\n"
        codigo += "li $v0 4\nla $a0 endl\nsyscall\n"
    return codigo

def generar_PL_prima(PL_prima):
    codigo = ""
    if PL_prima.hijos[0].elemento != "''":
        codigo = generar_PL_PL_prima(PL_prima.hijos[1],PL_prima.hijos[2])
```

```
    return codigo

def generar_PL(PL):
    codigo = generar_ex(PL.hijos[0].hijos[0])
    codigo += "sw $a0 0($sp)\naddiu $sp $sp-4\n"
    return codigo

def generar_PL_PL_prima(PL, PL_prima):
    codigo = generar_PL (PL)
    codigo += generar_PL_prima (PL_prima)
    return codigo

def generar_parametros(arbol):
    codigo = generar_PL_PL_prima(arbol.hijos[0], arbol.hijos[1])
    return codigo

def generar_llamada_de_mision(arbol):
    codigo = "sw $fp 0($sp)\naddiu $sp $sp-4\n"
    if arbol.hijos[1].hijos[1].hijos[0].elemento != "''":
        codigo += generar_parametros(arbol.hijos[1].hijos[1])
    codigo += "jal " + arbol.hijos[0].token.value + "\n"
    return codigo

def contar_Y(Y):
    global total
    total = total + 1
    parametros.append(Y.hijos[1].token.value)

def contar_Y_prima(Y_prima):
    if Y_prima.hijos[0].elemento != "''":
        contar_Y(Y_prima.hijos[1])
        contar_Y_prima(Y_prima.hijos[2])

def contar_parametros(arbol):
    global total
    total = 0
    contar_Y(arbol.hijos[0])
    contar_Y_prima(arbol.hijos[1])
    return total

def generar_mision(arbol):
    num_param = 0
    if arbol.hijos[3].hijos[0].elemento != "''":
        num_param = contar_parametros(arbol.hijos[3])
    codigo = arbol.hijos[1].token.value + ":\n"
    codigo += "move $fp $sp\nsw $ra 0($sp)\naddiu $sp $sp -4\n"
    codigo += generar_codigo(arbol.hijos[6])
    codigo += "lw $ra 4($sp)\naddiu $sp $sp " + str((4*num_param) + 8) + "\nlw $fp\n\n"
    return codigo

def generar_llama(arbol):
    codigo = ""
    for subarbol in arbol.hijos:
```



```

        if subarbol.elemento=="FuncionPrincipal":
            codigo = generar_codigo(subarbol)
        codigo += "li $v0 10\nsyscall\n"
        return codigo

def generar_codigo(arbol):
    codigo = ""
    if arbol.elemento == "FuncionPrincipal":
        codigo += "main:\n"
    if arbol.elemento == "DeclaracionVariable":
        codigo += generar_variable(arbol)
    if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
        arbol.hijos[0].elemento == "mostrar":
        codigo += generar_mostrar(arbol)
    if arbol.elemento == "J":
        codigo += generar_devuelve(arbol)
    if (arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
        arbol.hijos[0].elemento=="identificador"and
        arbol.hijos[1].hijos[0].elemento=="igual":
        codigo += generar_asignacion_de_variable(arbol)
    if ((arbol.elemento=="SentenciasYoyo" or arbol.elemento == "Sentencias") and
        (arbol.hijos[0].elemento=="si" or arbol.hijos[0].elemento=="yoyo")) or
        arbol.elemento == "DeclaracionFuncion":
        if arbol.hijos[0].elemento=="si":
            codigo += generar_if(arbol)
        elif arbol.elemento == "DeclaracionFuncion":
            codigo += generar_mision(arbol)
        else:
            codigo += generar_yoyo(arbol)
    else:
        for subarbol in arbol.hijos:
            if subarbol.elemento!="FuncionPrincipal":
                codigo += generar_codigo(subarbol)

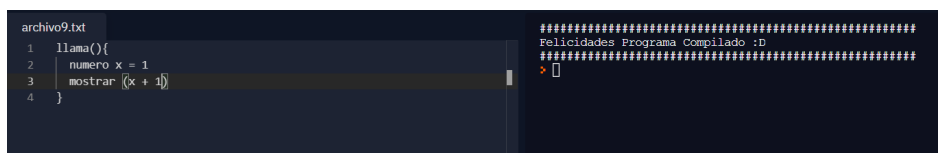
    return codigo

codigo_text = generar_llama(analizador_semantico.ll1.arbolito)
codigo_text += generar_codigo(analizador_semantico.ll1.arbolito)
codigo = data+"\n.text\n\n" + codigo_text

file=open('codigo.s','w')
file.write(codigo)
file.close()

```

- Funcionamiento de generación de código:



- Código Assembler generado:

```

codigo.s
1  .data
2
3  endl: .asciiz "\n"
4  x: .word 0:1
5
6  .text
7
8  main:
9  li $a0 1
10 sw $a0 x
11 lw $a0 x
12 sw $a0 0($sp)
13 addiu $sp $sp -4
14 li $a0 1
15 lw $t1 4($sp)
16 add $a0 $t1 $a0
17 addiu $sp $sp 4
18 li $v0 1
19 syscall
20 li $v0 4
21 la $a0 endl
22 syscall
23 li $v0 10
24 syscall
25
#####
Felicitades Programa Compilado :D
#####
>

```

9. Ejemplos de código:

- ```

mision suma (numero x) {
 numero resultado = 5 + x
 devuelve resultado
}
llama () {
 pinocho guri = mentira
 mostrar (suma (5))
}

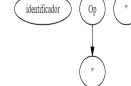
```

Tokens generados por el analizador léxico

```

mision identificador pizquierdo numero identificador pderecho lizquierdo numero
identificador igual lnumero mas identificador devuelve identificador
lderecho llama pizquierdo pderecho lizquierdo pinocho identificador igual
mentira mostrar pizquierdo identificador pizquierdo identificador pderecho
pderecho lderecho

```



```

llama ()
{
 numero x = 7
 yoyo (x == 7) {
 x = x + 1
 }
 mostrar (x)
}

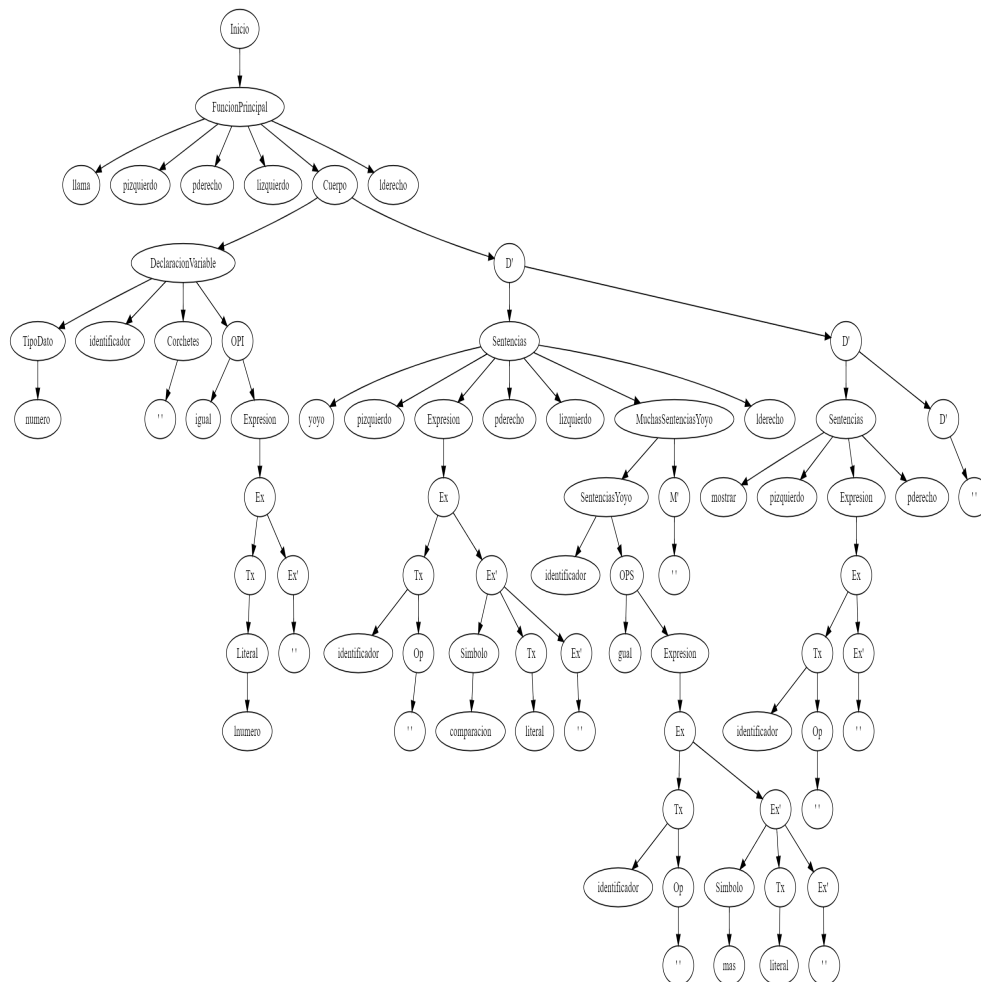
```

Tokens generados por el analizador léxico

```

llama pizquierdo pderecho lizquierdo numero identificador igual lnumero yoyo
pizquierdo identificador comparacion lnumero pderecho lizquierdo
identificador igual identificador mas lnumero lderecho mostrar pizquierdo
identificador pderecho lderecho

```



Tokens generados por el analizador léxico

[illegible]

## 10. Conclusiones

- La implementación de un lenguaje es una tarea compleja en el sentido de la cantidad de trabajo que implica, empero sencilla por la mecanización de la lógica una vez que ésta queda definida.
- El desarrollo de la especificación léxica debe considerar el nivel de sencillez que se busque para el lenguaje, evitando especialmente términos que sean muy parecidos, de forma tal que cada palabra reservada represente algo de forma exclusiva.
- La gramática puede parecer una tarea sencilla, sin embargo, implica encajar correctamente las reglas para coincidir el uso de cada término definido en la especificación léxica, esto quiere decir dar sentido real al lenguaje propuesto, para que pueda ser usado.
- Para evaluar la semántica del lenguaje, es necesario rastrear el recorrido lógico del código. Para esto se implementan métodos en el árbol sintáctico que permitan identificar los token contenidos en los nodos y lo que deben representar. De esta manera al recorrer el árbol podemos determinar la correcta significancia de cada expresión redactada dentro del código del lenguaje propuesto.
- Para completar el compilador se desarrolla el generador de código. Para ello se usa nuevamente los recorridos sobre el árbol sintáctico, para definir las funciones que retornen texto de código ensamblador. Según lo que represente el tipo de nodo que se va leyendo, debe construirse el texto ensamblador considerando la sintaxis propia del código ensamblador.
- El lenguaje propuesto es simple, y contribuye a que otros puedan entender fácilmente los preceptos de la programación. Contribuyendo al motivar el interés de nuevas personas en este campo.
- La construcción de esta presentación implicó bastante esfuerzo en horas de trabajo, recompensadas en los resultados, y el afianzamiento de los conceptos de compiladores en los participantes del grupo.