Instituto Tecnológico y de   
Estudios Superiores de Monterrey



Documentación Final para el Proyecto Final

Diseño de Compiladores

Ing. Elda Guadalupe Quiroga González

8 de mayo de 2019

***Javorgini***

Javier Benjamín Saldívar Mustafa

A01192358

**Índice**

Descripción del Proyecto \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3

Descripción del Lenguaje \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 8

Descripción del Compilador \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 9

Descripción de la Máquina Virtual \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 24

Pruebas \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 25

Documentación \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 28

**Descripción del Proyecto**

**Propósito**

La visión del proyecto, es crear mi propio lenguaje de programación.

**Objetivo**

El objetivo del proyecto es crear un lenguaje, así como un “Patito ++”, en la cual se pueden utilizar variables locales o globales, métodos, funciones con parámetros, asignaciones, if-statements, while y do-while loops, y arreglos de 1 dimensión.

**Aplicación**

Un lenguaje que puede utilizar estructura de datos para análisis estadístico, en la cual se puede mandar a llamar las siguientes funciones:

* max (regresa el número más grande dentro de la estructura)
* min (regresa el número más pequeño dentro de la estructura)
* range (regresa la resta entre el número más grande y más pequeño)
* iqrange (regresa la resta entre el número que está a 75% y el número que está a 25%)
* median (regresa el número que está a 50%, en medio)
* average (regresa el promedio de todos los números dentro de la estructura)
* stdv (regresa el promedio de la distancia de todos los números al promedio)
* variance (regresa la desviación estándar al cuadrado)

Además, se pueden crear estructura de datos tipo pieChart, barGraph y plotLine, las cuales podrás mandar a llamar las siguientes funciones:

* modify (mandas parámetros como el título y valor para alimentar el tipo de grafica)
* draw (se ejecuta el código para desplegar la gráfica con los valores actuales dentro)

**Análisis de requerimientos**

* Declaración del nombre del programa
* Declaración de variables locales
* Declaración de variables globales
* Declaración de arreglos de una dimensión
* Variables y arreglos de tipo
  + Int
  + Float
  + Bool
  + Char
* Declaración de múltiples variables en una línea
* Declaración de funciones
  + Void
  + Int
  + Float
  + Bool
  + Char
* Mandar parámetros de tipo
  + Int
  + Float
  + Bool
  + Char
* Regresar un valor
* Manejar cambio de contexto
* Declaración de una función main
* Manejar asignaciones
* Manejar condiciones
  + If
  + If-else
* Manejar escritura
  + Print
* Manejar ciclos
  + While
  + Do-while
* Manejar expresiones aritméticas
  + Suma
  + Resta
  + Multiplicación
  + División
* Manjar expresiones relacionales
  + Menor que
  + Mayor que
  + Menor o igual que
  + Mayor o igual que
  + Es igual
  + No es igual
* Manejar expresiones lógicas
  + And
  + Or
* Manejar comentarios
* Manejar funciones predeterminadas
  + Max
  + Min
  + Rango
  + Rango intercuartil
  + Mediana
  + Media
  + Desviación estándar
  + Varianza
* Crear graficas
  + Pie chart
  + Bar graph
  + Plot line

**Test Cases**

1. Lex
   1. Escribir palabras que se reconocen
   2. Escribir palabras que no se reconocen
2. Yacc
   1. Escribir líneas de código estructurados
   2. Escribir líneas de código no estructurados
3. Cubo Semántico
   1. Probar compatibilidad entre valores del mismo tipo
   2. Probar compatibilidad entre valores de diferentes tipo
   3. Probar tipo de resultado de operadores aritméticos
   4. Probar tipo de resultado de operadores lógicos
   5. Probar tipo de resultado de operadores relacionales
4. Mapeo de Memoria
   1. Verificar cada contexto (global, local, temporal, constante)
   2. Verificar cada rango de tipo (int, float, char, bool)
5. Código Intermedio
   1. Verificar cuádruplos de expresiones aritméticos
   2. Verificar cuádruplos de expresiones lógicos
   3. Verificar cuádruplos de expresiones relacionales
   4. Verificar cuádruplos de asignación
   5. Verificar cuádruplos de lectura
   6. Verificar cuádruplos de decisiones
   7. Verificar cuádruplos de ciclos
   8. Verificar cuádruplos de funciones
   9. Verificar cuádruplos de arreglos
6. Máquina Virtual
   1. Verificar ejecución de expresiones aritméticos
   2. Verificar ejecución de expresiones lógicos
   3. Verificar ejecución de expresiones relacionales
   4. Verificar ejecución de asignación
   5. Verificar ejecución de lectura
   6. Verificar ejecución de decisiones
   7. Verificar ejecución de ciclos
   8. Verificar ejecución de funciones
   9. Verificar ejecución de arreglos

**Bitácora**

Jueves 25 de abril – Visualizar objetivo, keywords y tokens y empezar diagramas de sintaxis a mano.

Domingo 28 de abril – Terminar diagramas de sintaxis a mano, empezar a pasarlas a electronico y empezar expresiones gramaticales sin recursión por la izquierda y sin ambigüedad a mano. Empezar lex.

Lunes 29 de abril – Terminar diagramas de sintaxis electrónico, terminar lex y corregir keywords y tokens. Terminar propuesta y llevarla a firmar, terminar expresiones gramaticales a mano, empezar y terminar yacc. Hacer pruebas de lex y yacc y ajustar acorde. Visualizar tabla de variables a mano.

Martes 30 de abril – Recoger propuesta rechazada, corregir propuesta, coregir lex y yacc acorde. Llevar a firmar propuesta. Recoger propuesta firmada. Hacer tabla de variables y tabla de funciones.

Miercoles 1 de mayo – Visualizar cubo semántico. Hacer cubo semántico. Puntos neurálgicos al crear variables y asignarlas a sus respectivas tablas de variables y a sus respectivos funciones “on the go”.

Sabado 4 de mayo – Generación de código intermedio de expresiones aritméticas y estatutos secuenciales. Generación de código intermedio de decisiones y ciclos. (Avance entregada)

Domingo 5 de mayo – Generación de código intermedio de funciones.

Lunes 6 de mayo – Crear máquina virtual y agregar cambios de contexto. Factorial iterativa y recursiva jalando.

Martes 7 de mayo – Generación de código intermedio de arreglos. Generación de código intermedio de funciones especiales. Empezar documentación.

Miércoles 8 de mayo – Terminar documentación.

**Reflexión Individual**

A lo largo del semestre, fuimos viendo diferentes temas, los cuales entendía, pero no fue hasta que me puse a hacer el proyecto, que todo se fue conectando lentamente, dando un aprendizaje más completo y menos abstracto. El mayor impacto fue volver funcional todo lo visto en diferentes clases como Organización Computacional, Lenguajes de Programación y obviamente Diseño de Compiladores. Esta clase fue la que más tiempo y esfuerzo demandó, ya que todo lo visto termino siendo aprendizaje, desde aprender a usar Python y aprender a usar GitHub, hasta aprender los componentes internos que lleva un lenguaje de programación, incluyendo el manejo de memoria y máquina virtual.

**Descripción del Lenguaje**

**Nombre**

Javorgini

**Descripción genérica**

El lenguaje Javorgini tiene una estructura básica que se divide en 3 secciones, la declaración del programa, la declaración de funciones y la declaración del main. Dentro de cada una de estas 3 secciones puedes declarar variables. La primera instrucción que ejecuta es ir al main. Dentro de la sección del main y la de funciones, ya puedes hacer diferentes estatutos como asignación y expresiones aritméticas. Tiene un enfoque de análisis estadístico entonces existen funciones predeterminadas las cuales mediante arreglos, puedes mandar un conjunto de datos, y Javorgini te regresará el número correspondiente a la operación llamada.

**Lista de errores**

1. Illegal character
2. Variable not declared
3. Type mismatch
4. Funcion not found
5. Parameter type mismatch
6. Syntax error at token
7. Syntax error at EOF

**Descripción del compilador**

**Equipo de cómputo**

Laptop HP Pavilion G4

**Lenguaje**

Python 3.7

**Librerías relevantes**

* PLY
* Pprint
* Matplotlib

**Descripción del Análisis de Léxico**

Keywords (palabras reservadas)

1. program
2. var
3. as
4. int
5. float
6. bool
7. char
8. pieChart
9. barGraph
10. plotLine
11. func
12. void
13. main
14. if
15. else
16. do
17. while
18. print
19. input
20. call
21. return
22. max
23. min
24. range
25. iqrange
26. median
27. average
28. stdev
29. variance
30. sort
31. find

Tokens

1. id [a-z][a-zA-Z\_0-9]\*
2. cte\_int d+
3. cte\_float d+ . d+
4. cte\_bool true|false
5. cte\_char '[a-zA-Z]'
6. cte\_string "[a-zA-Z\_0-9]\*"
7. ;
8. ,
9. .
10. (
11. )
12. {
13. }
14. [
15. ]
16. =
17. +
18. -
19. \*
20. /
21. <
22. >
23. <=
24. >=
25. ==
26. !=
27. &&
28. ||
29. !
30. //

**Descripción del Análisis de Sintaxis**

PROGRAMA -> program id ; PROGRAMA’ PROGRAMA’’ MAIN  
PROGRAMA’ -> VARS PROGRAMA’ | empty  
PROGRAMA’’ -> FUNCIONES PROGRAMA’’ | empty

VARS -> var id VARS’ as TIPO VARS’’  
VARS’ -> , id VARS’ | empty  
VARS’’ -> [ cte\_int ] | empty

TIPO -> int | float | bool | char

TIPO\_CTE -> cte\_int |cte\_float | cte\_bool | cte\_char | LLAMADA | id TIPO\_CTE’  
TIPO\_CTE’ -> [ EXPRESION ] | FUNCIONES\_ARR

FUNCIONES -> func FUNCIONES’ id ( FUNCIONES’’ ) BLOQUE\_MODULAR  
FUNCIONES’ -> void | TIPO  
FUNCIONES’’ -> TIPO id FUNCIONES’’’ | empty  
FUNCIONES’’’-> , TIPO id FUNCIONES’’’ | empty

MAIN -> void main ( ) BLOQUE\_MODULAR

BLOQUE\_MODULAR -> { BLOQUE\_MODULAR’ BLOQUE\_MODULAR’’ }  
BLOQUE\_MODULAR’ -> VARS BLOQUE\_MODUAR’ | empty  
BLOQUE\_MODULAR’’ -> ESTATUTO BLOQUE\_MODULAR’’ | empty

BLOQUE -> { BLOQUE’ }  
BLOQUE’ -> ESTATUTO BLOQUE’ | empty

ESTATUTO -> ASIGNACION | CONDICION | ESCRITURA | LLAMADA | RETURN | WHILE | DO\_WHILE

ASIGNACION -> id ASIGNACION’ = ASIGNACION’’  
ASIGNACION’ -> [ EXPRESION ] | empty  
ASIGNACION’’ -> EXPRESION | LEIDA

LEIDA -> input ( )

CONDICION -> if ( EXPRESION ) BLOQUE CONDICION’  
CONDICION’ -> else BLOQUE | empty

ESCRITURA -> print ( ESCRITURA’ ) ;  
ESCRITURA’ -> EXPRESION | cte\_string

LLAMADA -> call id ( LLAMADA’ ) ;  
LLAMADA’ -> TIPO id LLAMADA’’ | empty  
LLAMADA’’ -> , TIPO id LLAMADA’’ | empty

RETURN -> return EXPRESION ;

WHILE -> while ( EXPRESION ) BLOQUE

DO\_WHILE -> do BLOQUE while ( EXPRESION )

EXPRESION -> COMPARE EXPRESION’  
EXPRESION’ -> && COMPARE | || COMPARE | empty

COMPARE -> EXP COMPARE’  
COMPARE’ -> > EXP | < EXP | >= EXP | <= EXP | == EXP | != EXP | empty

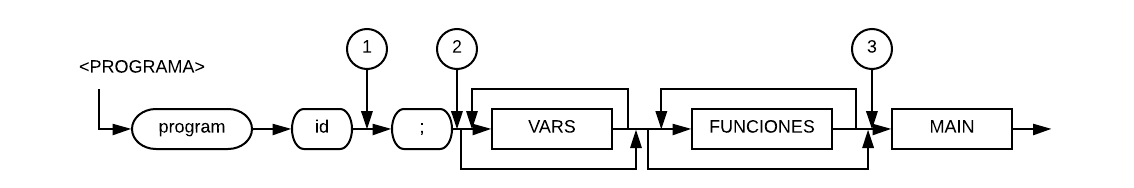
EXP ->TERMINO EXP’  
EXP’ -> + TERMINO EXP’ | - TERMINO EXP’ | empty

TERMINO -> FACTOR TERMINO’  
TERMINO’ -> \* FACTOR TERMINO’ | / FACTOR TERMINO’ | empty

FACTOR -> ( EXPRESION ) | TIPO\_CTE | - TIPO\_CTE | ! TIPO\_CTE

**Descripción de Generacion de Código Intermedio**

Diagramas de Sintaxis



1. Generar GOTO al main
2. Crear la tabla de variables y niveles de jerarquía global en la tabla de variables
3. Asignar main a la variable global de cambio de contexto, crear nivel de jerarquía main en la tabla de variables y rellenar el GOTO hecho en el punto 1 para el salto a main

'puntoCreateProgram : '

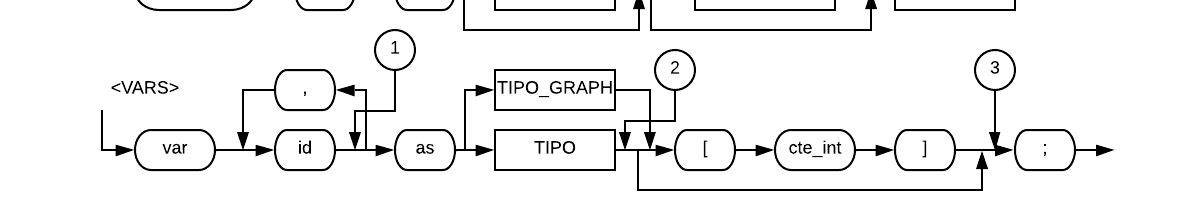
**global** programID

programID = p[-1]

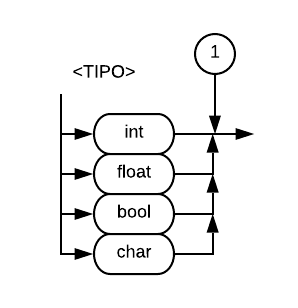
quad = ("GOTO", None, None, None)

pilaQuads.append(quad)

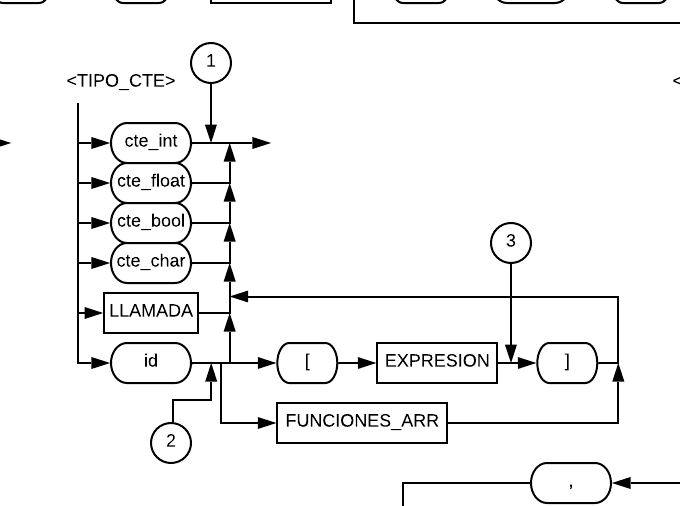
pilaSaltos.append(len(pilaQuads)-1)



1. Agregar ID anterior a la pila de variables
2. Para cada variable en la pila de variables, crear su instancia en la tabla de variables
3. Para cada variable en la pila de variables, cambiar la dimensión dentro de la tabla de variables



1. Guardar el tipo en una variable global



1. Para cada tipo constante, tiene su propio punto en la cual se saca la siguiente dirección temporal y se hace push al vector polaco.
2. Se busca en la tabla de variables, se saca la dirección y se hace push al vector polaco.
3. Se genera el VERIFY para verificar que la expresión está dentro de la dimensión del arreglo, se genera el cuádruplo de sumar la expresión a la dirección base del arreglo.

aux2 = vectorPolaco.pop()

aux1 = vectorPolaco.pop()

dimensionTemp = globalProgram[programID][currentState]['varTable'][currentIDGlobal]['Dimension']

quad = ("VER", aux2, 0, dimensionTemp)

pilaQuads.append(quad)

typeTemp = globalProgram[programID][currentState]['varTable'][currentIDGlobal]['Type']

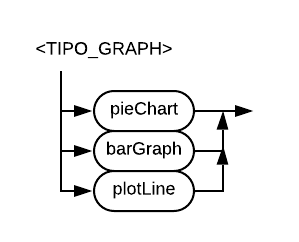
tempDir = getTempDir(typeTemp)

quad = ("+", tempInt, aux2, tempDir)

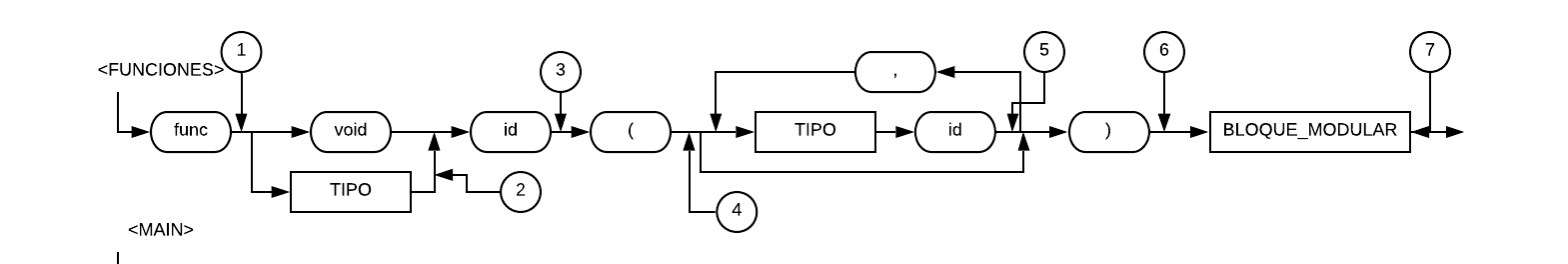
pilaQuads.append(quad)

tempListDir = []

tempListDir.append(tempDir)



No contiene puntos neurálgicos

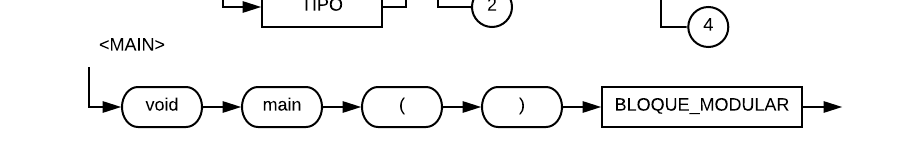


1. Reiniciar la memoria local
2. Guardar el tipo de return
3. Cambiar la variable global de contexto, y crear la tabla de variables para esta función
4. Crear tabla de parámetros bajo la jerarquía correspondiente a la función
5. Agregar cada variable a la tabla de variables bajo parámetros, guardar su tipo y la cuenta
6. Llenar la tabla de parámetros con los valores obtenidos y guardados (nombre, tipo, numero de parámetro)
7. Crear cuádruplo ENDPROC

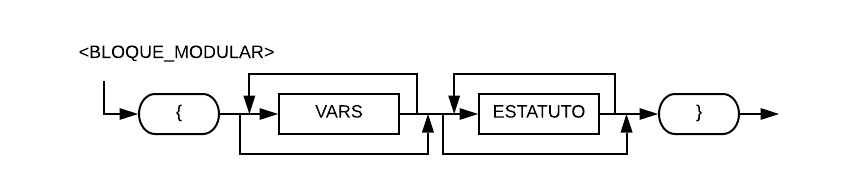
'puntoFinalFuncQuad : '

quad = ("ENDPROC", None, None, None)

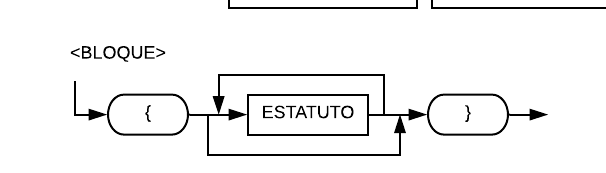
pilaQuads.append(quad)



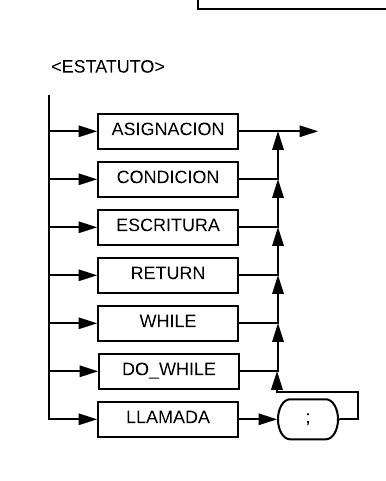
No contiene puntos neurálgicos



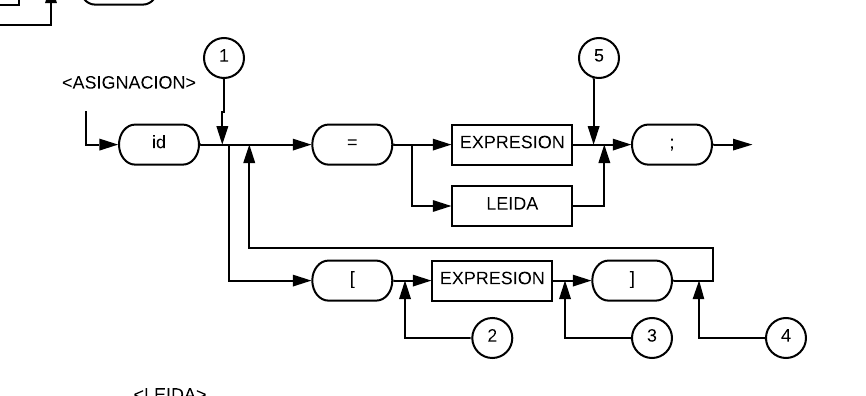
No contiene puntos neurálgicos



No contiene puntos neurálgicos



No contiene puntos neurálgicos



1. Guardar el ID, buscarlo en la tabla de variables y hacer push al vector polaco
2. Push fondo falso al vector polaco
3. Se genera el VERIFY para verificar que la expresión está dentro de la dimensión del arreglo, se genera el cuádruplo de sumar la expresión a la dirección base del arreglo.
4. Pop fondo falso
5. Se genera el cuádruplo de asignación

aux2 = vectorPolaco.pop()

aux1 = vectorPolaco.pop()

dimensionTemp = globalProgram[programID][currentState]['varTable'][currentIDGlobal]['Dimension']

quad = ("VER", aux2, 0, dimensionTemp)

pilaQuads.append(quad)

typeTemp = globalProgram[programID][currentState]['varTable'][currentIDGlobal]['Type']

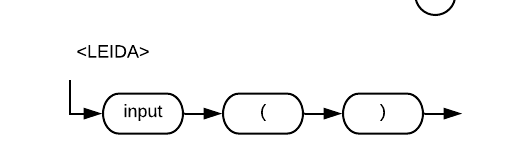
tempDir = getTempDir(typeTemp)

quad = ("+", tempInt, aux2, tempDir)

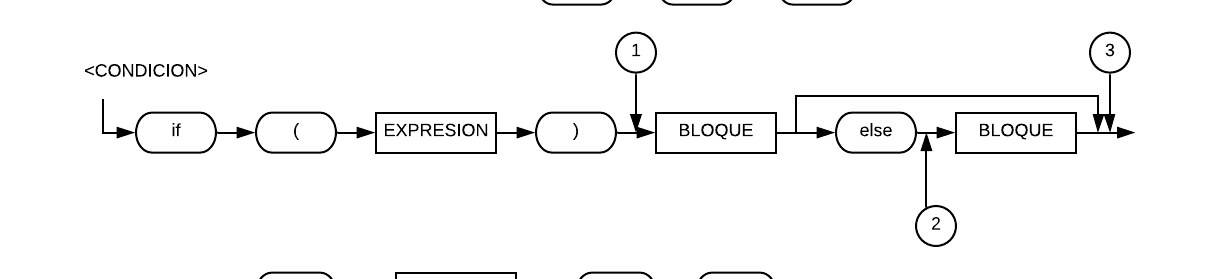
pilaQuads.append(quad)

tempListDir = []

tempListDir.append(tempDir)



No contiene puntos neurálgicos



1. Generar el cuádruplo GOTOF y hacer push a la pila de saltos
2. Generar el GOTO, hacer push a la pila de saltos y rellenar el GOTOF del IF (pop a la pila de saltos)
3. Pop a la pila de saltos, y rellenar el ultimo cuádruplo

result = vectorPolaco.pop()

quad = ("GOTOF", result, None, None)

pilaQuads.append(quad)

pilaSaltos.append(len(pilaQuads)-1)

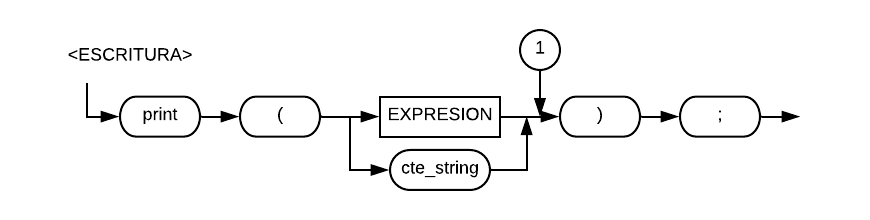
quad = ("GOTO", None, None, None)

pilaQuads.append(quad)

false = pilaSaltos.pop()

pilaSaltos.append(len(pilaQuads)-1)

fillQuad(false, len(pilaQuads)+1)

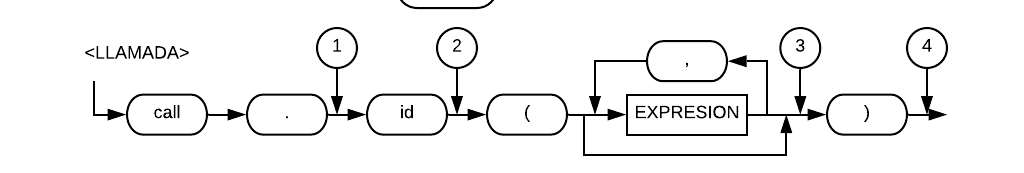


1. Generar cuádruplo de WRITE

toPrint = vectorPolaco.pop()

quad = ("write", None, None, toPrint)

pilaQuads.append(quad);



1. Push fondo falso
2. Checar si la función existe en la tabla de variables y generar cuádruplo de ERA
3. Verificar que el argumento coincida con el tipo que ya estaba creado en la tabla de parámetros y generar el cuádruplo PARAM
4. Pop fondo falso y generar cuádruplo GOSUB

quad = ("ERA", None, None, llamadaID)

pilaQuads.append(quad)

paramCounterToSend = 0

argumento = vectorPolaco.pop()

argumentoDir = globalProgram[programID][funcToCall]['paramTable']['direccion'][paramCounterToSend+1]

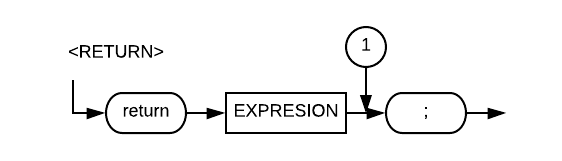
quad = ("PARAM", argumento, None, argumentoDir)

pilaQuads.append(quad)

jumpTo = goSubFunciones[funcToCall]

quad = ("GOSUB", funcToCall, None, jumpTo)

pilaQuads.append(quad)



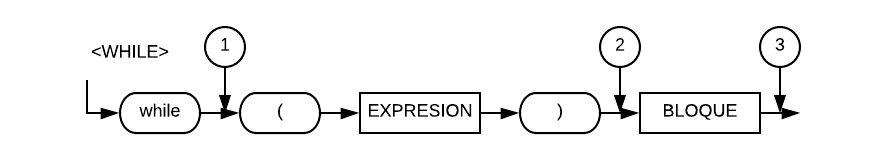
1. Generar cuádruplo de RETURN

returnTemp = vectorPolaco.pop()

returnDir = globalProgram[programID]['global']['varTable'][currentState]['Direccion']

quad = ("RETURN", returnTemp, None, returnDir)

pilaQuads.append(quad)



1. Push el número del cuádruplo a la pila de saltos
2. Checar si la expresión es booleana y generar GOTOF
3. Pop a la pila de saltos y rellenar el GOTOF y genera GOTO

result = vectorPolaco.pop()

quad = ("GOTOF", result, None, None)

pilaQuads.append(quad)

pilaSaltos.append(len(pilaQuads)-1)

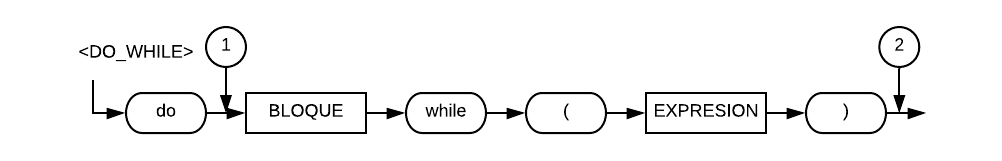
end = pilaSaltos.pop()

goBack = pilaSaltos.pop()

quad = ("GOTO", None, None, goBack)

pilaQuads.append(quad)

fillQuad(end, len(pilaQuads)+1)



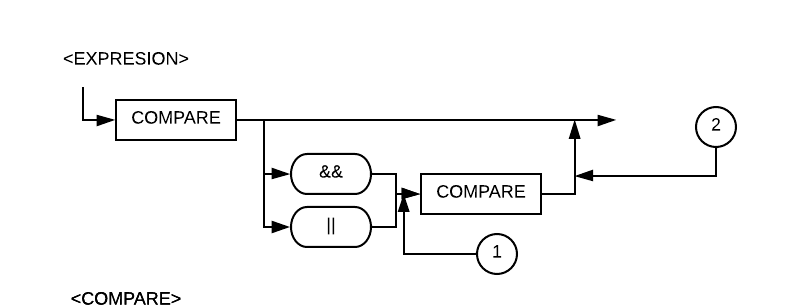
1. Push el número del cuádruplo a la pila de saltos
2. Verificar si la expresión es booleana, pop a la pila de saltos y generar GOTOT

result = vectorPolaco.pop()

goBack = pilaSaltos.pop()

quad = ("GOTOT", None, None, goBack)

pilaQuads.append(quad)



1. Push operador a la pila de operadores
2. Generar el cuádruplo de la operación

**if** pilaOper[length-1] == "&&":

createOperationQuad()

**elif** pilaOper[length-1] == "||":

createOperationQuad()

**def** createOperationQuad():

right\_operand = vectorPolaco.pop()

right\_Type = pilaTipos.pop()

left\_operand = vectorPolaco.pop()

left\_Type = pilaTipos.pop()

operator = pilaOper.pop()

operatorCube = convertSemCubeParam(operator)

leftTypeCube = convertSemCubeParam(left\_Type)

rightTypeCube = convertSemCubeParam(right\_Type)

result\_Type = semCube[operatorCube][leftTypeCube][rightTypeCube]

result\_Type = convertSemCubeParam(result\_Type)

**if** result\_Type != "Error":

result = getTempDir(result\_Type)

# print("create quad ", result\_Type)

quad = (operator, left\_operand, right\_operand, result)

# print(quad)

pilaQuads.append(quad)

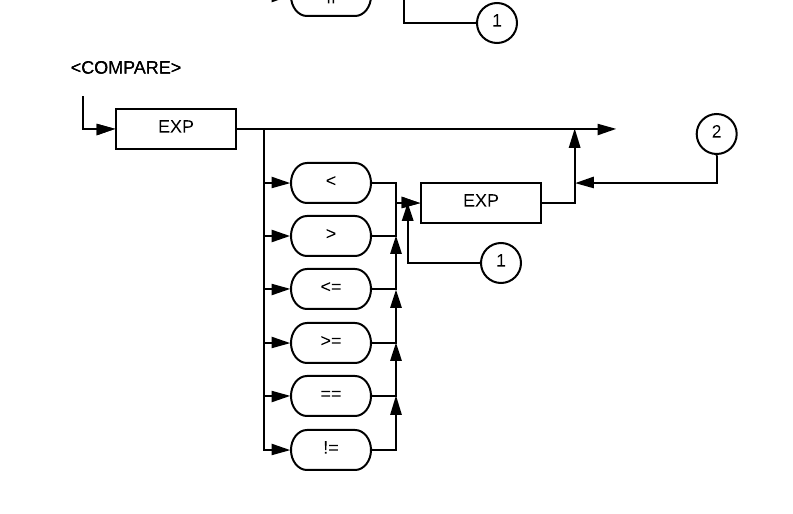
vectorPolaco.append(result)

pilaTipos.append(result\_Type)

**else**:

**print**("Error: Type mismatch", str(p.lexer.lineno))

sys.exit(0)



1. Push operador a la pila de operadores
2. Generar el cuádruplo de la operación

**if** pilaOper[length-1] == "<":

createOperationQuad()

**elif** pilaOper[length-1] == ">":

createOperationQuad()

**elif** pilaOper[length-1] == ">=":

createOperationQuad()

**elif** pilaOper[length-1] == "<=":

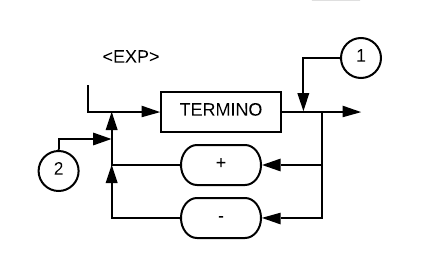
createOperationQuad()

**elif** pilaOper[length-1] == "==":

createOperationQuad()

**elif** pilaOper[length-1] == "!=":

createOperationQuad()



1. Mientras haya operadores dentro de la pila de operadores, generar el cuádruplo de la operación correspondiente
2. Push operador a la pila de operadores

length = len(pilaOper)

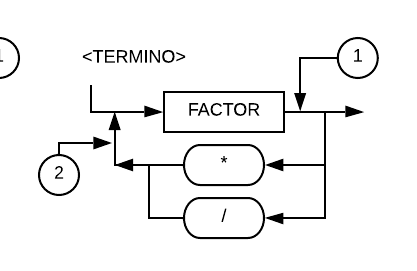
**if** length > 0:

**if** pilaOper[length-1] == "+":

createOperationQuad()

**elif** pilaOper[length-1] == "-":

createOperationQuad()



1. Mientras haya operadores dentro de la pila de operadores, generar el cuádruplo de la operación correspondiente
2. Push operador a la pila de operadores

length = len(pilaOper)

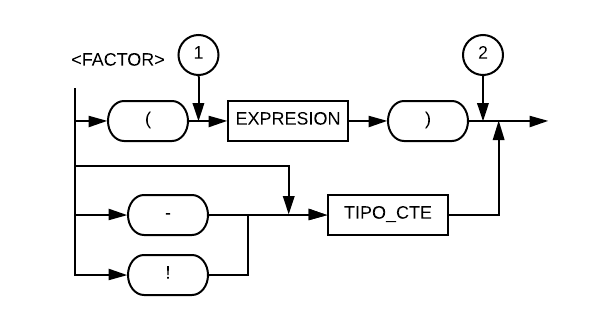
**if** length > 0:

**if** pilaOper[length-1] == "\*":

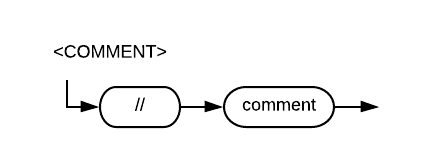
createOperationQuad()

**elif** pilaOper[length-1] == "/":

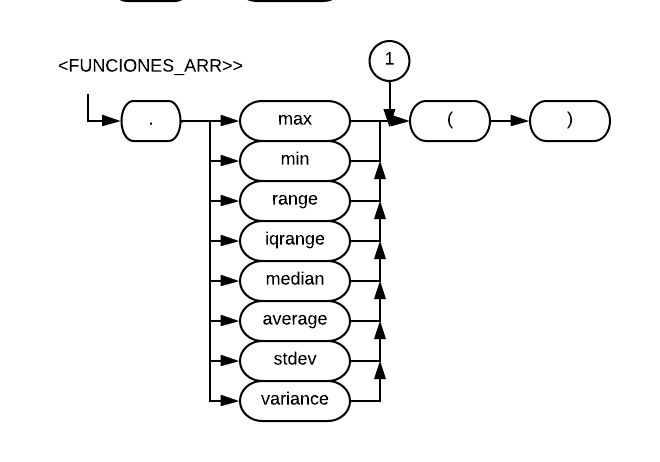
createOperationQuad()



1. Push fondo falso
2. Pop fondo falso



No tiene puntos neurálgicos



1. Pop a la pila de vector polaco, crear cuádruplo especial mandando (función, dimensión del arreglo, dirección del arreglo, dirección del resultado)

arrToSend = vectorPolaco.pop()

quad = (specialFuncToCall, arrDimToSend, arrToSend, resultDir)

pilaQuads.append(quad)

**Administración de Memoria**

La memoria se administra con puros diccionarios. Se tiene diccionarios adentro de diccionarios y la jerarquía de esta es:

Program 🡪 contexto 🡪 tabla de variables o parámetros 🡪dirección/tipo/dimensión

Ej. 1

{'factorialIterativo': {'factorial': {'paramTable': {'count': 1,

'direccion': {1: 10000},

'parametros': {1: 'x'},

'types': {1: 'int'}},

'returnType': 'int',

'varTable': {'x': {'Direccion': 10000,

'Name': 'x',

'Type': 'int'}}},

'global': {'varTable': {'factorial': {'Direccion': 1001,

'Type': 'int'}}},

'main': {'varTable': {}}}}

Ej. 2

{'ifQuadCheck': {'global': {'varTable': {}},

'main': {'varTable': {'a': {'Dimension': 0,

'Direccion': 10000,

'Name': 'a',

'Type': 'int'},

'b': {'Dimension': 0,

'Direccion': 10001,

'Name': 'b',

'Type': 'int'},

'c': {'Dimension': 0,

'Direccion': 10002,

'Name': 'c',

'Type': 'int'},

'd': {'Dimension': 0,

'Direccion': 10003,

'Name': 'd',

'Type': 'int'}}}}}

**Descripción de la Máquina Virtual**

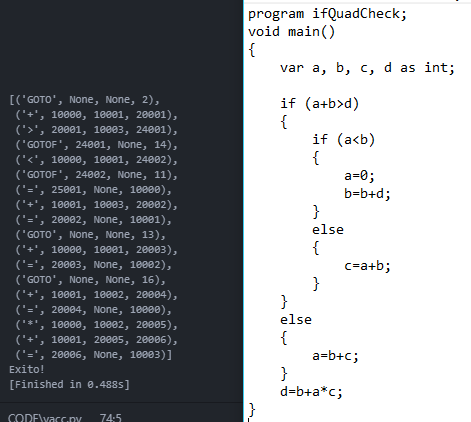
La administración de la memoria proviene del archivo de mapea de memoria virtual. En esta se encuentran las constantes donde empiezan cada variable global, local, temp y const ya sean int, float, char y bool como se muestra en la tabla siguiente

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Int | Float | Char | Bool |
| Global | 1000 | 5000 | 7000 | 9000 |
| Local | 10,000 | 15,000 | 17,000 | 19,000 |
| Temp | 20,000 | 22,000 | 23,000 | 24,000 |
| Const | 25,000 | 27,000 | 28,000 | 29,000 |

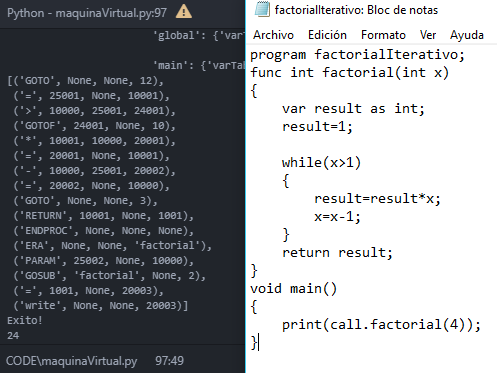
Dentro de este mismo archivo, cada contexto tiene su propia función que regresa la siguiente dirección, según el parámetro mandado, esta, siendo el tipo. Además, como la única memoria que se puede reiniciar, debido a cambio de contexto, es la local, entonces se puede llamar la función resetLocal(). Esta función anterior regresa el valor de la memoria a la inicial que se encuentra en la tabla. Para manejar cambio de contextos, en la máquina virtual, se maneja la memoria como una pila de memorias. Cuando se hace el GOSUB para ir a la función, se hace push una memoria limpia a la pila de memoria y cuando se hace return o end procedure, se hace pop a la memoria, de esta forma continuando con la memoria que se tenía un nivel antes.

**Pruebas del funcionamiento del lenguaje**

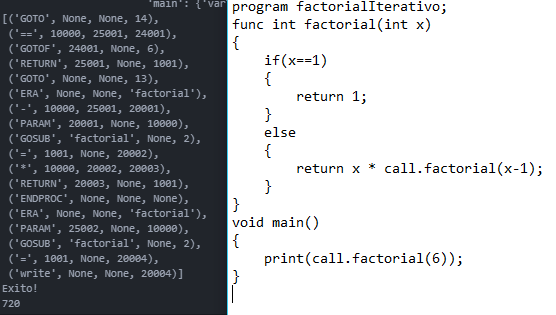
Prueba if/else



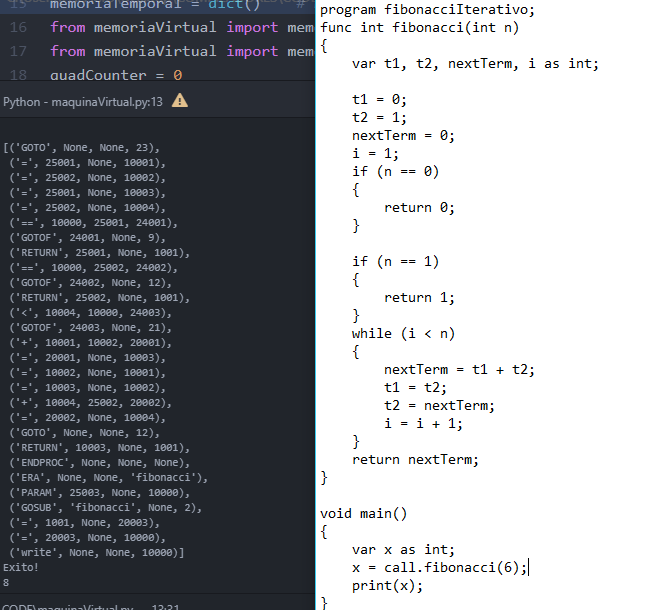
Prueba Factorial Iterativo



Prueba Factorial Recursivo



Prueba Fibonacci Iterativo



**Documentación**

**Liga con todo el código incluyendo README**

<https://github.com/javbsald/compis>

**Métodos en Máquina Virtual**

# Funcion que recibe una direccion de memoria y busca en la lista de memoria para conseguir el valor de esa direccion

**def** memoryToValue(mem):

**if** mem >= 1000 **and** mem < 10000: # Global

**if** mem **in** memoriaGlobal:

**return** memoriaGlobal[mem]

**else**:

**print**("Not found in Global Memory")

**elif** mem >= 10000 **and** mem < 20000: # Local

#print(mem, " in ", localList, " lengh=", localListLength)

**if** mem **in** localList[localListLength-1]:

**return** localList[localListLength-1][mem]

**else**:

**print**(mem, "Not found Local Memory")

**elif** mem >= 20000 **and** mem < 25000: # Temp

**if** mem **in** memoriaTemporal:

**return** memoriaTemporal[mem]

**else**:

**print**("Not found in Temporal Memory")

**elif** mem >= 25000 **and** mem < 30000: # Const

**return** memoriaConstanteDir[mem]

# Funcion que recibe result para asignar a la direccion de resultDir

**def** assignToMemory(result, resultDir):

**if** resultDir >= 1000 **and** resultDir < 10000: # Global

memoriaGlobal[resultDir] = result

**elif** resultDir >= 10000 **and** resultDir < 20000: # Local

localList[localListLength-1][resultDir] = result

**elif** resultDir >= 20000 **and** resultDir < 25000:

memoriaTemporal[resultDir] = result

#elif resultDir >= 25000 and resultDir < 30000: # Const

#memoriaConstante[resultDir] = result

# Como podemos tener direcciones de memoria que apuntan a otra direccion de memoria

# esta funcion verifica que ninguna de los otros 3 que no es el inicial sea un apuntador

**def** verifyQuadDirections(quadRecieved):

#print("Quad Recieved")

#print(quadRecieved)

#print(isinstance(quadRecieved[1], int))

#print(isinstance(quadRecieved[2], int))

#print(isinstance(quadRecieved[3], int))

newDir1=quadRecieved[1]

newDir2=quadRecieved[2]

newDir3=quadRecieved[3]

**if**(isinstance(quadRecieved[1], int) == False):

**if** quadRecieved[1] != None:

#print(quadRecieved[1], "NOT INT")

newDir1 = memoryToValue(quadRecieved[1][0])

**if**(isinstance(quadRecieved[2], int) == False):

**if** quadRecieved[2] != None:

#print(quadRecieved[2], "NOT INT")

newDir2 = memoryToValue(quadRecieved[2][0])

**if**(isinstance(quadRecieved[3], int) == False):

**if** quadRecieved[3] != None:

#print(quadRecieved[3], "NOT INT")

newDir3 = memoryToValue(quadRecieved[3][0])

newQuad = (quadRecieved[0], newDir1, newDir2, newDir3)

#print("Quad Converted", newQuad)

**return** newQuad

# Funciones Predeterminadas

**def** getArray(arr, dim):

counter = 1

arrValues = []

#print(arr, dim)

**while**(counter<=dim):

value = memoryToValue(arr+counter)

#print(value)

arrValues.append(value)

counter += 1

# print("GET ARRAY", arrValues)

**return** arrValues

**Funciones en parser**

# Se llama esta función cada vez que se quiere asignar alguna expresión aritmética, lógica o relacional

**def** createOperationQuad():

right\_operand = vectorPolaco.pop()

right\_Type = pilaTipos.pop()

left\_operand = vectorPolaco.pop()

left\_Type = pilaTipos.pop()

operator = pilaOper.pop()

operatorCube = convertSemCubeParam(operator)

leftTypeCube = convertSemCubeParam(left\_Type)

rightTypeCube = convertSemCubeParam(right\_Type)

result\_Type = semCube[operatorCube][leftTypeCube][rightTypeCube]

result\_Type = convertSemCubeParam(result\_Type)

**if** result\_Type != "Error":

result = getTempDir(result\_Type)

# print("create quad ", result\_Type)

quad = (operator, left\_operand, right\_operand, result)

# print(quad)

pilaQuads.append(quad)

vectorPolaco.append(result)

pilaTipos.append(result\_Type)

**else**:

**print**("Error: Type mismatch", str(p.lexer.lineno))

sys.exit(0)