Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Proyecto Final
Equipo #1

Open Roonie

Análisis y Diseño de Compiladores

Antonio Carlos Vargas Torres	A00813182
Rubén Eugenio Cantú Vota	A00814298

Indice

1 Descripción de proyecto

1.1 Visión del proyecto

Nuestra visión es crear ***Open Roonie***, un lenguaje de programación ligero que contenga los elementos básicos de los lenguajes imperativos procedurales actuales. Utilizaremos la herramienta PLY (Python Lex-Yacc) para implementar el léxico y sintaxis de nuestro proyecto, además, planeamos utilizar una de las herramientas de google developers, Blocky.

1.2 Objetivo del lenguaje

Como objetivo de proyecto, se enfocará en desarrollar un lenguaje de programación que pueda ser entendible por niños, jóvenes y adultos que no estén relacionados con la programación. El lenguaje ***Open Roonie*** utilizará blocky para la interfaz gráfica, de tal manera que la programación sea intuitiva y fácil de hacer.

1.3 Alcance del proyecto

El proyecto tiene como alcance el diseño de un lenguaje de programación sencillo en el cual se desarrollan las etapas de análisis léxico, sintaxis y semántica. Por otra parte, se hace uso de herramientas como Blockly y PythonAnywhere que permiten implementar una interfaz amigable y fácil de usar para el usuario final.

1.4 Análisis de requerimientos

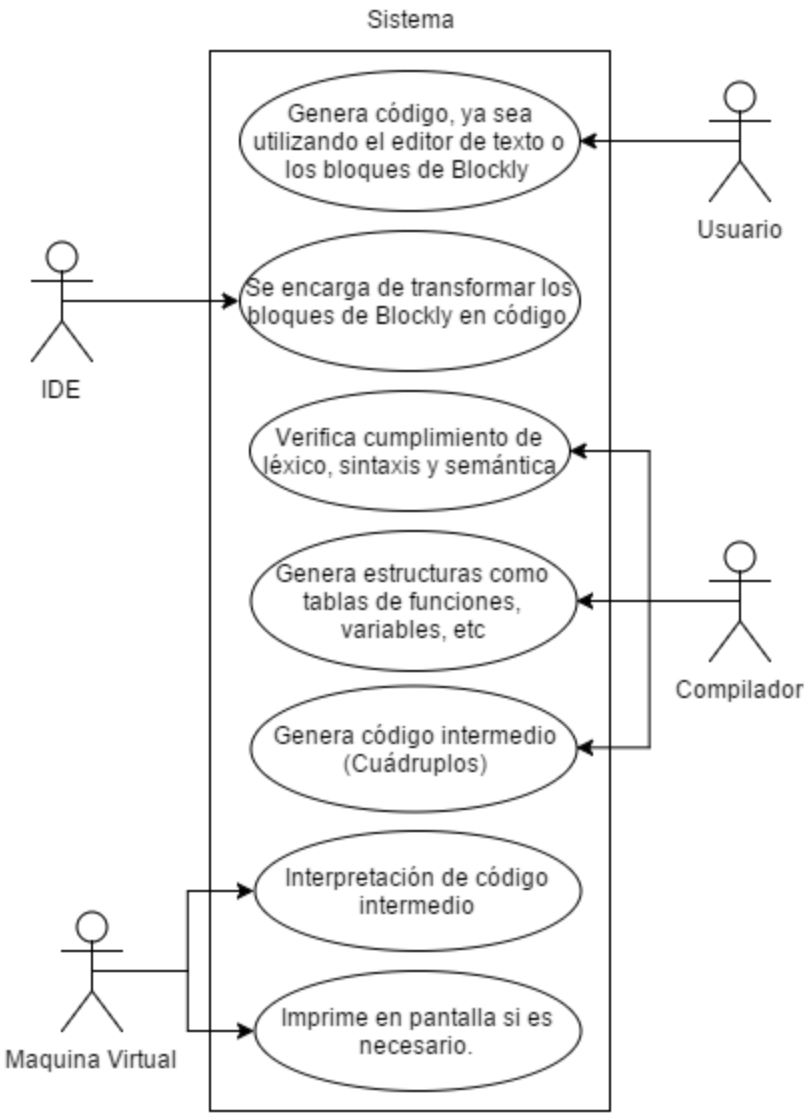
1.4.1 Requerimientos funcionales

- ➔ El compilador valida que los archivos a procesar sean escritos en el lenguaje Open Roonie (.roonie)
- ➔ El compilador detecta los errores que existan en sintaxis y semántica. A sí mismo, despliega el tipo de error al usuario.
- ➔ El compilador reconoce los tipos de datos declarados para verificar que las operaciones, condiciones, secuencias, funciones, variables y arreglos sean aceptables.
- ➔ El compilador administra funciones y variables en un directorio.
- ➔ El compilador administra asignaciones de memoria virtual para el programa que está siendo procesado.
- ➔ El compilador genera una lista de cuádruplos mediante la lectura del programa, tomando como referencia toda la información almacenado en el administrador de memorias y directorio de funciones.
- ➔ La máquina virtual tiene la capacidad de realizar operaciones aritméticas, sean sumas, restas, multiplicaciones y divisiones.
- ➔ La máquina virtual tiene la capacidad de procesar estatutos, asignaciones, secuencias, funciones, variables y arreglos.
- ➔ La máquina virtual genera números aleatorios dado un parámetro, al igual que la lectura y escritura de variables.
- ➔ La interfaz gráfica tiene opciones para importar y exportar archivos en XML para facilitar el uso de Blockly.
- ➔ La interfaz gráfica tiene opciones para importar y exportar archivos en .roonie para facilitar el uso del editor de texto en línea.
- ➔ La interfaz gráfica tiene una terminal en la cual se pueden ejecutar los programas desarrollados en blockly y/o en el editor de textos.

1.4.2 Requerimientos no funcionales

- ➔ El compilador y la máquina virtual son utilizados desde una página web.
- ➔ La interfaz gráfica es sencilla y fácil de usar.

1.5 Casos de uso



1.6 Descripción de los principales casos de uso

Los principales casos de uso nos permiten verificar que el funcionamiento del compilador y de la máquina virtual sean correctos y adecuados, tal y como se diseñaron. De tal manera que, por cada avance, se realizaron pruebas para comprobar su comportamiento. A continuación, se muestra un listado de las pruebas realizadas:

1. Validación de reglas de léxico y sintaxis.
2. Validación de semántica.
3. Generación de código de asignaciones de variables.
4. Generación de código de expresiones aritméticas.
5. Generación de código de estatutos condicionales.
6. Generación de código de estatutos secuenciales.
7. Generación de código de funciones.
8. Generación de código de parámetros.
9. Generación de código de asignaciones de argumentos.
10. Generación de código de impresión y lectura de sistema.
11. Ejecución de código de asignaciones de variables.
12. Ejecución de código de expresiones aritméticas.
13. Ejecución de código de estatutos condicionales.
14. Ejecución de código de estatutos secuenciales.
15. Ejecución de código de funciones.
16. Ejecución de código de parámetros.
17. Ejecución de código de asignaciones de argumentos.
18. Ejecución de código de impresión y lectura de sistema.
19. Generación de código de arreglos N-dimensionales.
20. Ejecucion de codigo de arreglos N-dimensionales.
21. Generación de código de números aleatorios de sistema.
22. Ejecución de código de números aleatorios de sistema.

1.7 Descripción del proceso general

La técnica ágil de desarrollo de software que se utilizó a lo largo del proyecto es *“Pair programming”*. Ambos integrantes del equipo estuvieron participando activamente durante la solución de problemas y la implementación de las soluciones. Todas las sesiones de trabajo principales fueron presenciales, tanto en el Tecnológico de Monterrey como en espacios privados de trabajo. Por otra parte, se realizaron pequeñas correcciones de errores y optimizaciones de manera remota.

1.7.1 Bitacora general

1. Léxico y Sintaxis	24/09/2016
Se implementó la primera versión de léxico y sintaxis. Ya se han establecido los tokens al igual que las reglas que debe seguir el programa. Por otra parte, las gramáticas diseñadas en la propuesta fueron corregidas.	

2. Directorio de Funciones y Tablas de Variables	06/10/2016
Implementamos una estructura llamada *‘FunctionDirectory’*, la cual se encarga de manejar el directorio de funciones con sus tablas de variables correspondientes. En esa misma estructura, se verifica que no existan variables repetidas en un mismo scope, y que no se repitan funciones en un mismo programa. Se incluye un programa de prueba en el cual, si se utiliza en *OpenRoonie*, imprimirá al final el diccionario y el directorio de funciones de dicho programa.	

3. Semántica y Expresiones	15/10/2016
Una clase *'MemoryManager'*, que se encarga de asignar una dirección virtual a un dato, según su tipo y su scope. Una clase *'SemanticCube'*, la cual se encarga de ver la compatibilidad de datos según sus operaciones, en caso de no ser compatible, tira un error de semántica, y en caso de ser compatibles, retorna el tipo de valor resultante.	

Una clase *'QuadrupleManager'*, que se encarga de generar los cuádruplos resultantes de, por el momento, solo operaciones aritméticas (+, -, *, /)
La clase que ya existía *'FunctionDirectory'* ahora se encarga también de administrar las variables temporales creadas por el *'QuadrupleManager'*, así también como las constantes que se encuentren en el source code que se de como entrada.

Finalmente, fueron agregadas las instrucciones necesarias en la sintaxis del yacc para hacer que las clases anteriormente mencionadas fueran utilizadas correctamente.

4. Generación de Código para Estatutos	24/10/2016
La generación de cuadruplos para: asignación, print, estatutos condicionales y ciclos (específicamente if, if else, while); las instrucciones necesarias fueron agregadas a la clase *'QuadrupleManager'*, así como en las reglas sintácticas correspondientes. Ademas, por cada instrucción de cuádruple que agregamos en esta entrega, hacemos que se verifiquen las reglas de semántica de nuestro lenguaje (como el tipado de variables para la asignación, que los estatutos sean booleanos (para condiciones y ciclos), etc.) En general, vamos al corriente con nuestro compilador según el calendario de entregas de avance de proyectos.	

5. Código Intermedio d Funciones	29/10/216
Asociatividad por izquierda. Verificación de tipos de datos de parámetros en las llamadas a funciones. Asignación del valor cuando se invoca una función a los parámetros correspondientes.	

6. Diseño del Mapa de Memoria para Ejecución	05/11/2016
En este avance se modificó la sintaxis del lenguaje para que obligatoriamente todas las funciones terminen con un return seguido de un valor (a excepción de las funciones void). En cuanto a lo que nos faltaba con relación a los cuádruplos de funciones: creamos el cuádruplo *'ERA'* y *'GOSUB'*, que nos ayudarán en la máquina virtual a saber donde comienza la función que se invoque, y cuanto espacio en memoria requiere para ser cargada. (En la entrega anterior hicimos *'PARAMS'*, no recuerdo en qué estado lo	

habíamos dejado, pero ya funciona correcto y completo).

En resumen: Ya terminamos nuestra fase de compilación (a excepción de la generación de código intermedio para arreglos), y comenzamos a diseñar cómo implementaremos nuestra máquina virtual (planeamos realizarla en el lenguaje Python).

7. Máquina Virtual	15/11/2016

En este avance se realizó la implementación de la máquina virtual: en este momento es capaz de ejecutar: funciones, funciones recursivas, ciclos, condiciones, lecturas, impresiones, operaciones aritmeticas y logicas. Además, se modificaron algunas cosas de la sintaxis para que fuese más cómodo y flexible el programar en Roonie. También se hicieron implementaciones de optimización en la creación de cuadruplos así como en la ejecución de estos.

Actualmente estamos trabajando en la implementación de vectores n-dimensionales, y de la interfaz facilitada por Blockly, esperamos tener ambas cosas completas para el siguiente avance.

8. Open Roonie	19/11/2016

Este es el último avance en el cual ya están implementados toda la funcionalidad del compilador y de la máquina virtual. Por otra parte, la implementación web está en línea, haciendo uso de las herramientas de Blockly y la terminal de Python Anywhere.

El proyecto está actualmente en línea, en la siguiente liga:
www.hollowlife.com/OpenRoonie/

1.7.2 Reflexiones

A lo largo del semestre, tuve muchas inquietudes e intereses en como funcionaba la herramienta de PLY. Me permitió conocer las reglas que existen en los lenguajes de programación, ya sea el léxico, sintaxis y semántica y cómo estos deben de ser procesados. Se tuvo una muy buena experiencia a lo largo del proyecto, ya que todas las sesiones de trabajo se realizaron con tiempo. Definitivamente no es una actividad sencilla de realizar, de no ser por las páginas de información con Little Luck, esto hubiera sido aún más difícil. Por otra parte, gracias a esta experiencia, me siento cómodo y capaz de poder volver a trabajar en algun compilador. De errores y faltas de optimización, se aprendieron formas eficaces de desarrollar los componentes. Finalmente, el proyecto me enseñó maneras de trabajar y maneras de pensar diferente gracias a la retroalimentación de mi compañero.

_____

Antonio Carlos Vargas Torres

Tras realizar este proyecto me di cuenta sobre todos los pasos que se realizan una vez que mando compilar mi codigo, aveces, uno como programador cree que está haciendo un buen trabajo porque obtiene los resultados que quiere, pero no se pone a pensar en todo el overhead, el trabajo extra que le esta pidiendo a la computadora. Definitivamente el desarrollar un lenguaje no es tarea fácil, pues hay que pensar desde el léxico hasta cómo generar el código objeto. Me pareció un proyecto bastante retador, aun y cuando estuvimos al tanto de todas las entregas, fue una experiencia que me ha hecho valorar mucho las funcionalidad que ofrecen los lenguajes de alto nivel de hoy en dia, y me dejan con la intriga de como realizan varias de sus implementaciones.

_____

Rubén Eugenio Cantú Vota

2 Descripción del lenguaje

2.1 Nombre del lenguaje

El lenguaje se llama Open Roonie, el cual toma significado con los siguientes motivos:
Open porque el proyecto es Open Source, aceptando pulls en gitHub.
Roonie es una palabra compuesta por Roo (pronunciado *Ru*) y nie (pronunciado *ny*), en honor a los creadores del lenguaje (**Ru**bén y To**ny**).

2.2 Descripción de las principales características del lenguaje

El lenguaje Open Roonie ha sido influenciado por distintos lenguajes de programación, entre ellos JavaScript, C, Python y Little Duck.

Acerca del léxico, se tienen las siguientes palabras reservadas del lenguaje: if, else, while, var, program, main, function y return. Por otra parte, también se tienen funcionalidades del sistema, como lo son read(var), print(vars*, constantes*) y random(minimo, maximo). Por último, se tienen los tipos de datos y tipo de función int, float, char, string y bool. Todas estas son las palabras utilizadas por el lenguaje Open Roonie. También cuenta con caracteres para los operadores aritméticos y lógicos, como lo son +, -, *, / al igual que >, <, >=, <=, ==, <> y =. Se hace uso de paréntesis, corchetes, llaves, comas, punto y coma, comillas sencillas y comillas simples.

Finalizando la fase de léxico, los identificadores se validan empezando con letras, seguidos de 0 o más letras y/o dígitos. Los números enteros pueden ser negativos, ceros o positivos. Los números flotantes similar, sin embargo tienen la alternativa de contener punto con 1 o mas digitos.

Ahora bien, la sintaxis del lenguaje tiene la siguiente estructura:
En un principio, es necesario escribir la palabra reservada ‘program’, seguido de un identificador para nombrar el programa. Una vez hecho, es posible declarar variables globales con el siguiente formato: ‘var *tipo* varname;’, donde tipo puede ser cualquier tipo de variable válido y donde varname es el identificador o nombre de una variable.

Es importante destacar que es posible declarar mas de una variable del mismo tipo por línea, solo basta con agregar una coma al final y escribir otro identificador. Finalmente, si se desea declarar un vector, se realiza de la siguiente manera: varname[N] o varname[N, M]... en donde N y/o M es la longitud del vector en la dimensión especificada.

Una vez declaradas las variables globales, es posible declarar las funciones del programa. Para crear una función, solo basta con seguir el siguiente formato general: 'function *tipo* funcname'. Esta parte es muy similar a la de variable, sin embargo no se pueden declarar funciones en una misma línea de código. Por otra parte, el tipo de la funcion tambien puede ser void, es decir, sin retorno específico. Ahora bien, las funciones pueden o no contener parámetros. Para definirlos, basta con abrir paréntesis. Si no contiene parámetros, los paréntesis se quedan vacíos. En caso de tener, se tendrá que definir el tipo y el nombre de la variable. Esto es separado por comas en caso de tener mas de 1 parametro. Una vez terminado el encabezado de la función, se prosigue a desarrollar el bloque.

Los bloques son delimitadores, identificados entre llaves {}, en el lenguaje que permiten agrupar las instrucciones que el programador desee. Dentro de los bloques, se pueden realizar asignaciones, condiciones, ciclos, escrituras, lecturas, declaración de variables y llamadas a función. Cabe destacar que las funciones creadas por el programador, deberán de contener al final la palabra reservada return. Si es una función void, basta con escribir 'return;'. Sin embargo, si la función tiene algun tipo, debera retornar una variable o constante de ese mismo tipo.

Es importante reconocer que las variables globales y las funciones son completamente opcionales. Esto se debe a que la función principal es main() y debe de ser declarada como la última función. Debido a ser una función del sistema, no se debe escribir el prefijo 'function'. Al igual que las demás funciones, main debe contener un bloque pero no debe contener la palabra reserver return al final.

Para finalizar, se comenta acerca de las funciones especiales, como lo son la escritura de variables y constantes (print), la lectura de variables (read) y la generación de números aleatorios (random). Estas funciones pueden ser llamadas desde cualquier función.

2.3 Descripción de Errores

Los errores en Open Roonie están categorizados de la siguiente manera:

ERROR I/O:

En caso de utilizar un archivo con el código fuente, el compilador espera recibir un archivo que exista, que se tenga acceso de lectura, y que finalice con el char EOF.

ERROR LÉXICO:

Si el compilador encuentra en el código fuente algun caracter no registrado en su diccionario de tokens.

ERROR SINTAXIS:

Si el compilador encuentra un token válido para el lenguaje, pero no existe ninguna variable sintáctica que lo consuma, se mostrará un error de sintaxis donde se encontró dicho token.

ERROR SEMÁNTICA:

Los errores de semántica son desplegados cuando secciones de código, sintácticamente correctas, necesitan asegurar que se cumplan ciertas reglas para darle significado al lenguaje. El mas claro ejemplo es cuando se quieren realizar operaciones entre diferentes operandos, donde es necesario verificar sus tipos de datos para ver si son compatibles, en Open Roonie, utilizamos una clase llamada '*SemanticCube'* precisamente para eso (anexado más adelante en este documento). Otro motivo por el cual pueden salir errores de semántica es si se llaman invocar funciones dándole una cantidad diferente de parámetros que los que tiene definidos en la tabla de funciones.

ERROR MEMORIA:

Estos errores pueden salir tanto en compilación como en ejecución, pueden salir por dos diferentes motivos, el primero es por intentar acceder a una dirección que no ha sido inicializada, el segundo es si se hace un 'request' y ya no hay memoria suficiente para dar.

ERROR RUNTIME:

Los errores en ejecución son provocados por cosas que están fuera del alcance de la compilación, tienen que ver con los valores que toman variables, los

ejemplos más claros son la división entre cero, y el acceder a subíndices inválidos de vectores.

3 Descripción del Compilador

3.1 Equipo de cómputo

Como el compilador fue implementado en *Python*, nos permitió utilizar diferentes sistemas operativos durante el desarrollo de este. No se utilizó particularmente una computadora potente, por lo que no es problema utilizarlo en PCs con especificaciones promedio.

3.2 Lenguaje

El único lenguaje utilizado en el desarrollo del compilador es *Python*, especificamente la version 3.5.

3.3 Utilerías especiales

Se utilizó la implementación de *Lex* / *Yacc* en *Python* (*PLY*) para la construcción de los tokens (a partir de expresiones regulares definidas en la siguiente sección de este documento) y, así como su verificación léxica (que sean tokens válidos) y cumplimiento de las reglas sintácticas (que los tokens vengan en el orden descrito por la gramática libre de contexto).

http://www.dabeaz.com/ply/

3.4 Descripción del análisis léxico

3.4.1 Patrones de construcción de los elementos principales

- Palabras reservadas: if, else, var, program, void, int, float, char, bool, True, False, string, read, print, random, function, main, while, return.
- Identificadores = [a-z | A-Z][A-Z|a-z|0-9]
- Constantes enteras = -?[0-9][0-9]*
- Constantes flotantes = -?[0-9]+.[0-9]+
- Constantes string = r'\"([^\\\n]|(\\.))*?\"'
- Constantes char = r'(L)?\'([^\\\n]|(\\.))*?\"
- Operadores aritméticos: = + - * /
- Operadores logicos: = > >= < <= == <>
- Operador asignacion: = ‘=’
- Separadores: = () {} [] , ; “” ‘’

3.4.2 Enumeracion de tokens del lenguaje y codigo asociado

<table>
<tr>
<td>Tokens de operadores aritméticos:
● PLUS ‘+'
● MINUS '-'
● TIMES '*'
● DIVIDE /'</td>
<td>Tokens de operadores de asignacion:
● EQUALS '='

Tokens de control de flujo:
● 'if' 'IF',
● 'else' 'ELSE'</td>
</tr>
<tr>
<td>Tokens de operadores logicos:
● MORETHAN '>'
● LESSTHAN '<'
● NOTEQUAL '<>'
● MORETHANEQUAL '>='
● LESSTHANEQUAL '<='
● ISEQUALTO '=='
● OR '|'
● AND '&'</td>
<td>Tokens de separadores:
● LPAREN '('
● RPAREN ')'
● LBRACKET '{'
● RBRACKET '}'
● LSQBRACKET '['
● RSQBRACKET ']'
● SEMICOLON ';'
● COMMA ','</td>
</tr>
</table>

Tokens de tipos de datos:	Tokens de programa:
● 'int' : 'INT', ● 'float' : 'FLOAT', ● 'char' : 'CHAR', ● 'bool' : 'BOOL',	● 'var' :'VAR', ● 'program' :'PROGRAM', ● ‘read’ : ‘READ’ ● ‘print’ : ‘PRINT’ ● ‘random’ : ‘RANDOM’ ● ‘return’ : ‘RETURN’

3.5 Descripción de análisis de sintaxis

A continuación, se muestra la gramática formal empleada:

```
<PROGRAMA> : program ID ; <VARS> <FUNCS> main <BLOQUE>
<PROGRAMA> : program ID ; <FUNCS> main <BLOQUE>
<PROGRAMA> : program ID ; <VARS> main <BLOQUE>
<PROGRAMA> : program ID ; main <BLOQUE>
<VARS> : var <TIPO> id [ <DIMS> ] <MASVARS> ;
<VARS> : var <TIPO> id  <MASVARS> ;
<MASVARS> : , id [ <DIMS> ] <MASVARS>
<MASVARS> : , id  <MASVARS>
<DIMS> : <EXPRESION> <MASEXPRESION> | empty
<FUNCS> : function <TIPO> id ( <ARGS> ) <BLOQUEFUNC>
<ARGS> : <TIPO> id <MASARGS> | empty
<MASARGS> : , <ARGS> | empty
<TIPO> : int | float | char | bool | string | void
<BLOQUEFUNC> : { <ESTATUTO> <MASESTATUTO> return <EXPRESION> }
<ESTATUTO> : <ASIGNACION> | <CONDICION> | <CICLO> | <ESCRITURA> | <LECTURA> | <VARS> | <LLAMA
FUNC> ;
<MASESTATUTO> : <ESTATUTO> <MASESTATUTO> | empty
<ASIGNACION> : id = <EXPRESION> ;
<ASIGNACION> : id [ <DIMS> ] = <EXPRESION> ;
<DIMS> : <EXPRESION> <MASEXPRESION>
<MASEXPRESION> : , <EXPRESION> <MASEXPRESION> | empty
<CONDICION> : if ( <EXPRESION> ) (ESTATUTO | BLOQUE)
<CONDICION> : if ( <EXPRESION> ) (ESTATUTO | BLOQUE) else (ESTATUTO | BLOQUE)
<CICLO> : while ( <EXPRESION> ) (ESTATUTO | BLOQUE)
<ESCRITURA>: print ( <EXPRESION> <MASEXPRESION> ) ;
<LECTURA> : read ( id ) ;
<LLAMA FUNC> : id ( <PARAMS> )
<PARAMS> : <EXPRESION> <MASPARAMS>
<MASPARAMS> : , <EXPRESION> <MASPARAMS> | empty
<EXPRESION> : <EXP_COMP> |  <EXP_COMP> <ANDOR> <EXP_COMP>
<ANDOR> : '&' | '|'
<EXP_COMP> : <EXP> | <EXP> <COMPARE> <EXP>
<COMPARE> : '>' | '<' | '>=' | '<=' | '==' | '<>'
<EXP> : <TERMINO> <MASTERMINO>
<MASTERMINO> : (+ | -) <TERMINO> <MASTERMINO< | empty
<TERMINO> : <FACTOR> <MASFACTOR>
<MASFACTOR> : (* | /) <FACTOR> <MASFACTOR> | empty
```

3.6 Descripción de generación de código intermedio y análisis sintáctico

3.6.1 Código de operación

Tipos de alcances que existen en la clase ‘MemoryManager’

global	0
constante	1
local	2

Tipos de datos utilizados en Open Roonie

int	0
float	1
char	2
bool	3
string	4
void	5

3.6.2 Direcciones virtuales asociadas a los elementos del código

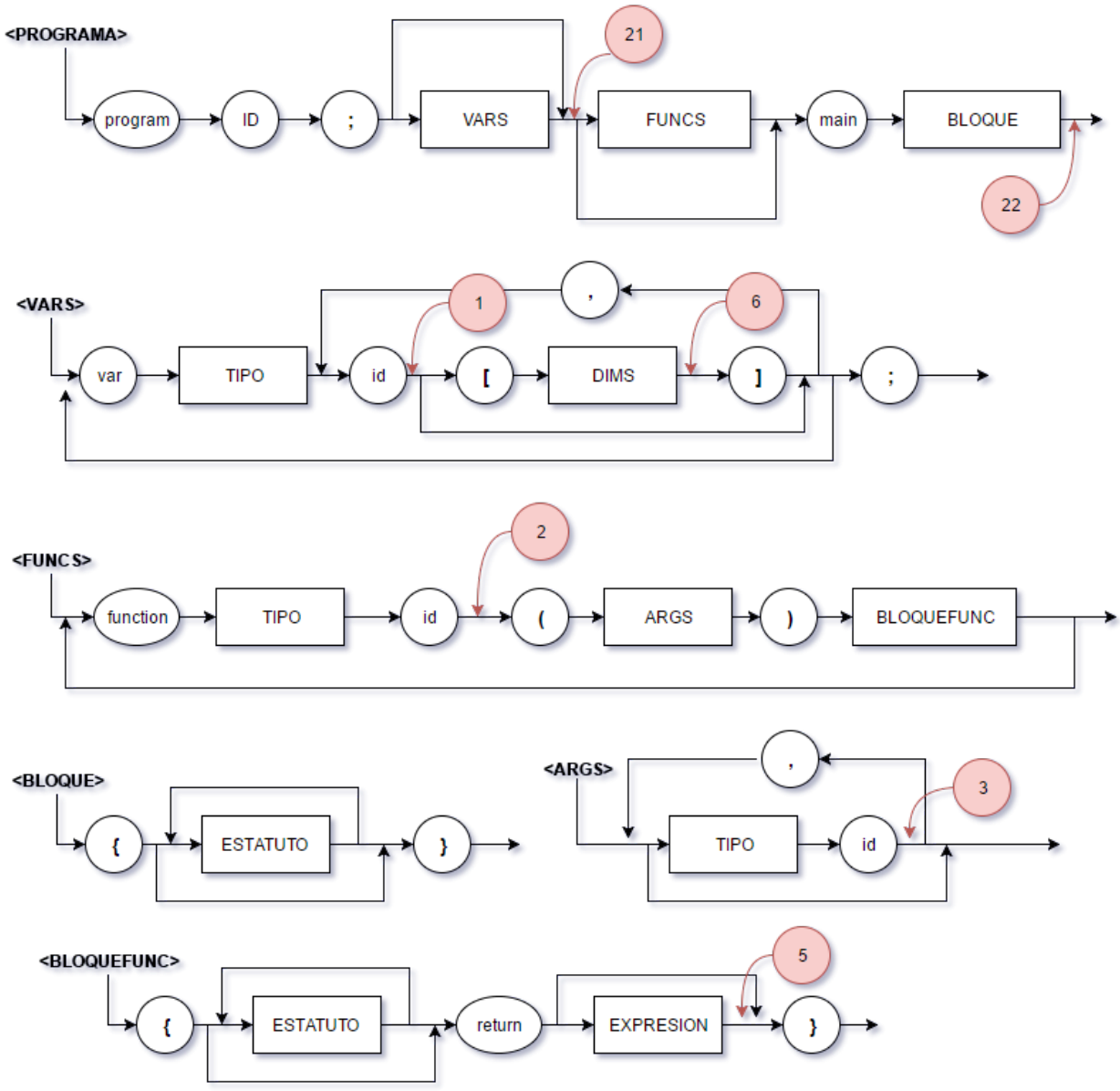
Tipo / Scope	Variables globales	Constantes	Variables locales
int	1000	7000	13000
floats	2000	8000	14000
char	3000	9000	15000
bool	4000	10000	16000
string	5000	11000	17000

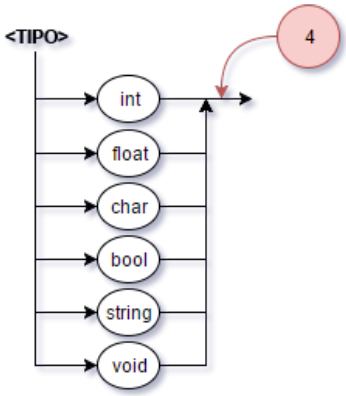
Las variables temporales que vayan siendo requeridas, serán tratadas como variables locales al contexto donde se hayan solicitado.

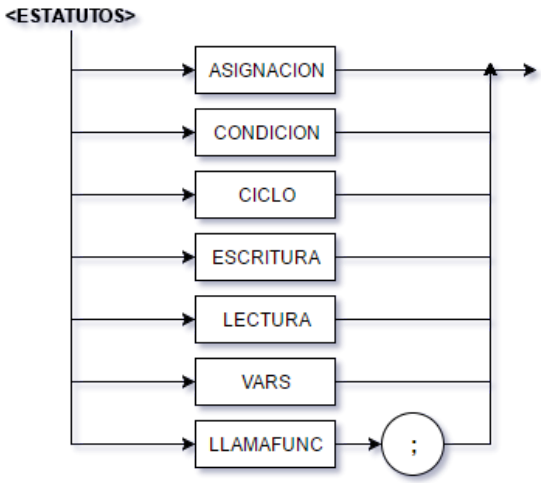
Operaciones utilizadas en la generación de cuádruplos:
(Operaciones 0 - 12 pertenecen al cubo semántico)

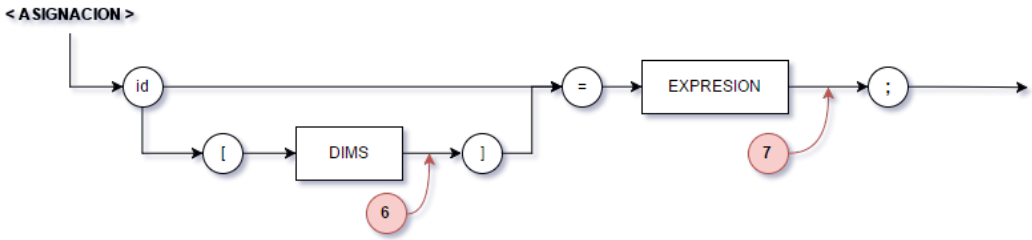
+	0	**=**	12	**end**	24
-	1	**print**	13		
*****	2	**read**	14		
/	3	**random**	15		
>	4	**gotoT**	16		
<	5	**gotoF**	17		
>=	6	**goto**	18		
<=	7	**goSub**	19		
<>	8	**params**	20		
==	9	**return**	21		
|	10	**era**	22		
&	11	**verify**	23		

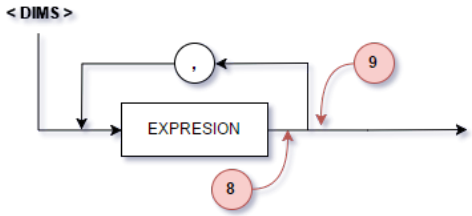
3.7 Diagramas de sintaxis

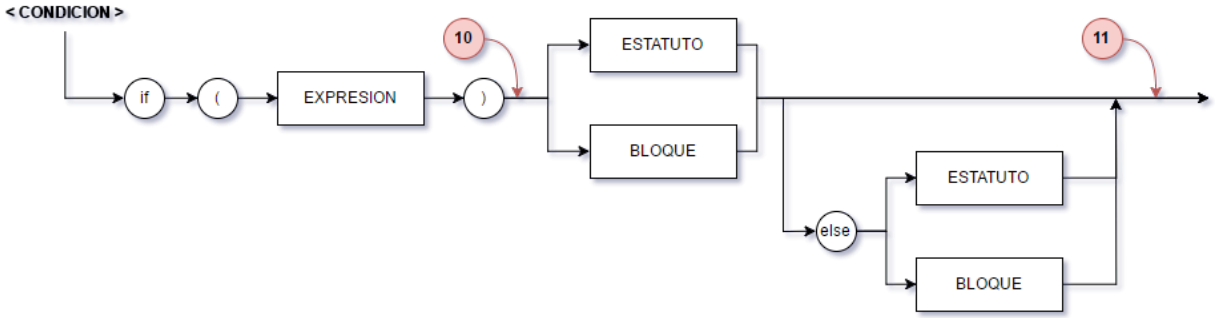


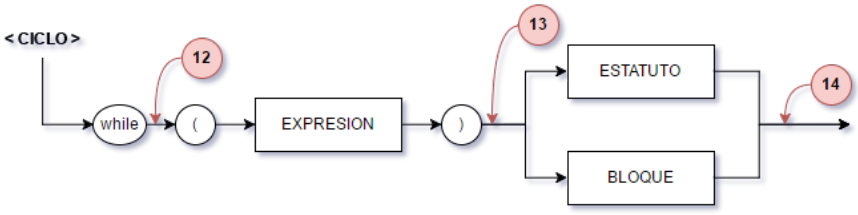

<TIPO>
4
int
float
char
bool
string
void

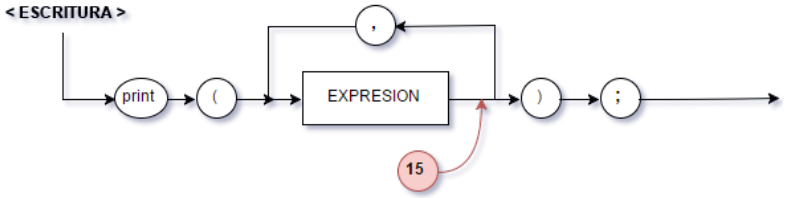

<ESTATUTOS>
ASIGNACION
CONDICION
CICLO
ESCRITURA
LECTURA
VARS
LLAMAFUNC
;

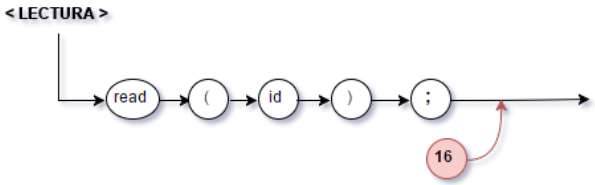

< ASIGNACION >
id
=
EXPRESION
;
[
DIMS
]
6
7

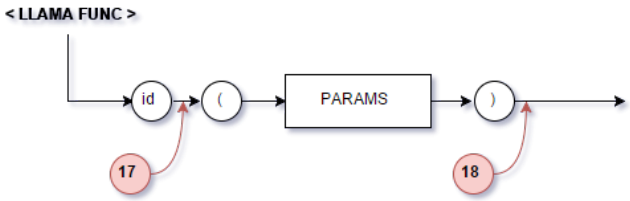

< DIMS >
,
9
EXPRESION
8

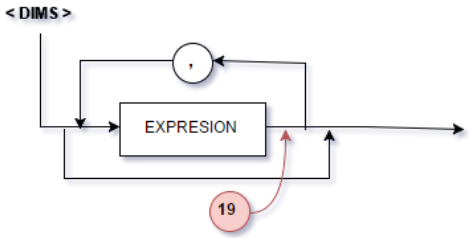

< CONDICION >
if
(
EXPRESION
)
10
ESTATUTO
BLOQUE
11
else
ESTATUTO
BLOQUE

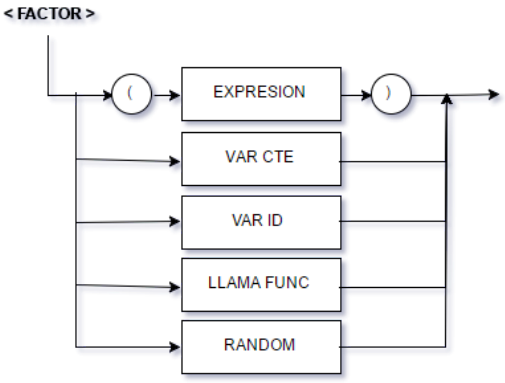

< CICLO >
12
while
(
EXPRESION
)
13
ESTATUTO
BLOQUE
14

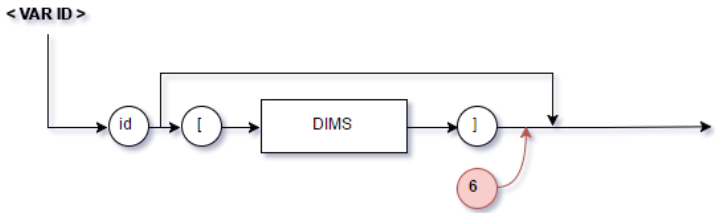

< ESCRITURA >
print
(
,
EXPRESION
)
;
15

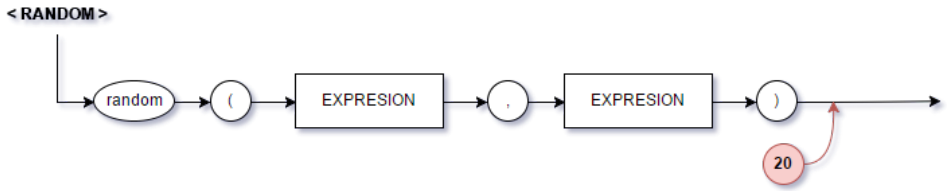

< LECTURA >
read
(
id
)
;
16

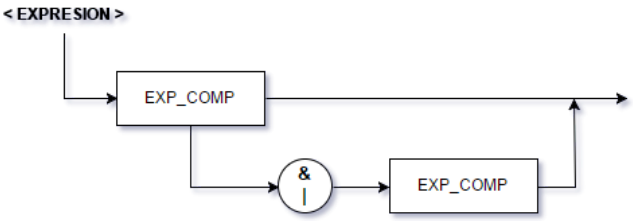

< LLAMA FUNC >
id
(
PARAMS
)
17
18

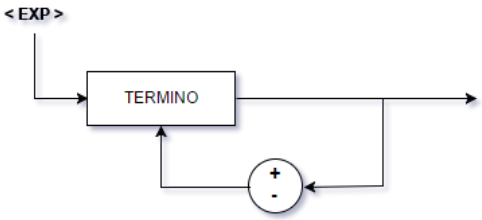

< DIMS >
,
EXPRESION
19

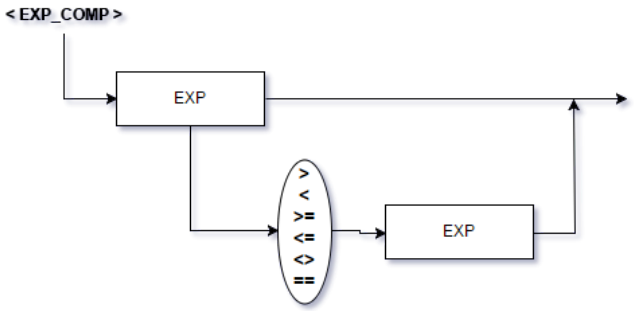

< FACTOR >
(
EXPRESION
)
VAR CTE
VAR ID
LLAMA FUNC
RANDOM

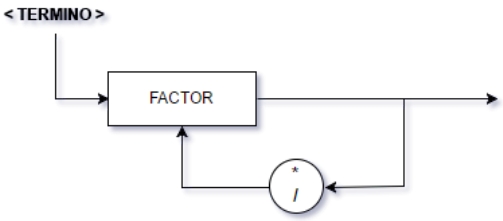

< VAR ID >
id
[
DIMS
]
6

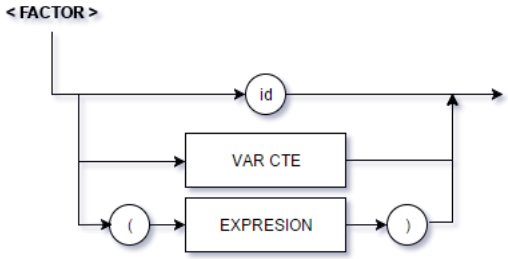

< RANDOM >
random
(
EXPRESION
,
EXPRESION
)
20

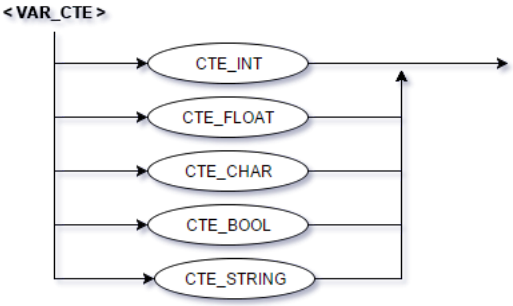

< EXPRESION >
EXP_COMP
&
|
EXP_COMP


< EXP >
TERMINO
+
-


< EXP_COMP >
EXP
>
<
>=
<=
<>
==
EXP


< TERMINO >
FACTOR
*
/


< FACTOR >
id
VAR CTE
(
EXPRESION
)


< VAR_CTE >
CTE_INT
CTE_FLOAT
CTE_CHAR
CTE_BOOL
CTE_STRING

3.8 Descripción de cada una de las acciones semánticas y de código

1. Verificar que el id no exista ya en el scope en donde está siendo declarada, y en caso de no existir, agregar un registro en la tabla de variables perteneciente a la función correspondiente.

2. Verificar que no exista un registro en el directorio de funciones con ese id, en caso de no existir agregarlo, junto con su tipo de dato de retorno y el número de cuádruplo siguiente.

3. Verificar que el id que se le de a cada argumento no se repita en la tabla de variables de la función correspondiente, y en caso de que no se repita, agregarlo a la tabla de variables de la función y modificar el registro de tipos de parámetros en el renglón de la función correspondiente (en el directorio).

4. Agregar a la pila de tipos el token que se acaba de consumir.

5. Verificar en el directorio de funciones el tipo de dato de retorno de la función correspondiente y compararlo con el tipo de dato resultante de la expresión, en caso de ser void debe seguir un ‘;’ después del return.

6. Verificar que las dimensiones sean de tipo entero y cantidad correspondiente a dicho vector (según sus dimensiones).

7. Verificar que sean compatibles los tipos de datos de las variables que planeo asignar, y en caso de serlo, generar el cuádruplo de asignación (‘=’).

8. En caso de no ser una declaración, se deberá crear un cuádruplo de ‘*’ según el subíndice proporcionado con la ‘M’ correspondiente almacenada en la tabla de variables. Si es una declaración, se deberá almacenar el tamaño del vector por cada una de sus dimensiones.

9. En caso de no ser declaración, y haber completado la instrucción 8 para todas sus dimensiones, se deberán sumar estos resultados con la dirección base del

vector, almacenada también en la tabla de variables. Si es declaración se deberá calcular la cantidad total de memoria requerida, y se hará el 'request' a memoria, adicionalmente es necesario calcular las 'M's y almacenarlas en el renglón de esta variable vector en la tabla de variables.

10. Se deberá verificar que el tipo de dato del resultado de la expresión sea de tipo booleano, además se creará un cuádruplo gotoF, donde su valor de retorno quedará pendiente.

11. Llamar a la funcion *updateReturnReference* de la clase *'MemoryManager'* y cambiar el cuádruplo gotoF que dejamos pendiente en la instrucción 10, por el número del siguiente cuádruplo.

12. Guardar el valor del cuádruplo actual en la pila de saltos.

13. Se deberá verificar que el tipo de dato del resultado de la expresión sea de tipo booleano, además se creará un cuádruplo gotoF, donde su valor de retorno quedará pendiente.

14. Crear cuádruplo goto hacia el pop de la pila de saltos (el valor almacenado en el paso 12); adicionalmente llamar a la funcion *updateReturnReference* de la clase *'MemoryManager'* y cambiar el cuádruplo gotoF que dejamos pendiente en la instrucción 13, por el número del siguiente cuádruplo.

15. Generar cuádruplo 'print' con el pop de la pila de operadores.

16. Generar cuádruplo 'read' con el token id que se le dio.

17. Generar cuádruplo 'era' de la función correspondiente.

18. Generar cuádruplo 'goSub' con el número de cuádruplo almacenado en el renglón de la función correspondiente en el directorio de funciones.

19. Por cada parámetro, verificar que sea de tipo correspondiente al de la función que se está invocando, en caso de no ser del mismo tipo, o que se manden una cantidad diferente de parámetros deberá marcar error, de caso contrario, generar el cuádruplo 'params'.

20. Los dos últimos tipos de la pila de tipos, deberán ser ‘int’, en caso de ser así, generar cuádruplo ‘random’, en caso contrario marcar error.

21. Generar cuádruplo ‘era’ y ‘goSub’ al primer cuádruplo de main.

22. Generar cuádruplo ‘end’ para señalar que se terminó el programa.

3.9 Tabla de consideraciones semánticas

Considerando el primer operador de tipo ‘int’

```
# int
#  +   -   *   /   >   <  >=  <=  <>  ==  |   &   =
[ 0,  0,  0,  0,  3,  3,  3,  3,  3,  3, -1, -1,  0]   # int
[ 1,  1,  1,  1,  3,  3,  3,  3,  3,  3, -1, -1,  1]   # float
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # char
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # bool
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # string
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # void
```

Considerando el primer operador de tipo ‘float’

```
# float
#  +   -   *   /   >   <  >=  <=  <>  ==  |   &   =
[ 1,  1,  1,  1,  3,  3,  3,  3,  3,  3, -1, -1, -1]   # int
[ 1,  1,  1,  1,  3,  3,  3,  3,  3,  3, -1, -1,  1]   # float
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # char
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # bool
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # string
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]         # void
```

Considerando el primer operador de tipo ‘char’

```
# char
#  +   -   *   /   >   <  >=  <=  <>  ==  |   &   =
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # int
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # float
[-1, -1, -1, -1, -1, -1, -1, -1,  3,  3, -1, -1,  2]   # char
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # bool
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # string
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # void
```

Considerando el primer operador de tipo ‘bool’

```
# bool
#  +   -   *   /   >   <  >=  <=  <>  ==  |   &   =
```

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # int
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # float
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # char
[-1, -1, -1, -1, -1, -1, -1, -1,  3,  3,  3,  3,  3]   # bool
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # string
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # void
```

Considerando el primer operador de tipo ‘string’

```
# string
#  +   -   *   /   >   <  >=  <=  <>  ==  |   &   =
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # int
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # float
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # char
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # bool
[-1, -1, -1, -1, -1, -1, -1, -1,  3,  3, -1, -1,  4]   # string
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]   # void
```

Donde el valor -1 significa error semántico y todas las operaciones que involucren a alguna variable tipo void retornarán error.

3.10.1 Especificaciones de las estructuras de datos utilizadas en el proceso de compilación.

Para poder llevar a cabo la tarea de compilación, nos vimos en la tarea de crear diferentes clases que se encargaran de administrar las diferentes estructuras utilizadas durante este proceso, dentro de estas clases estan:

FunctionDirectory.py

Encargada de mantener el directorio de funciones, con todos sus métodos correspondientes. La principal estructura de esta clase está definida de la siguiente manera:

Manejamos un diccionario de funciones, la cual el nombre de la función la utilizamos como key, y el valor es un número incremental que comienza desde 0. Decidimos utilizar un diccionario porque internamente python utiliza un algoritmo de hashing, el cual hace que los datos sean de rápido acceso.

El valor almacenado en este diccionario representa el renglón de la función en el directorio de funciones (que no es más que un array), el cual está definido de la siguiente manera:

[nombreFuncion, tipoRetorno, diccionarioVars, directorioVars, cuadStart, [paramTypes], [regActivacion], cantidadTemporales]

Donde directorio Vars se define como:
[[nombreVar, tipo, direccion virtual]]

Ej.

```
['sumatoria',  'void',
        {       '_t0': 2,
                '_t1': 3,
                '_t2': 4,
                '_t3': 5,
                '_t4': 6,
                'i': 0,
                'n': 1
        },
        [       ['i', 'int', 13000, [], [1]],
                ['n', 'int', 13001, [], [1]],
                ['_t0', 'bool', 16000, [], [1]],
                ['_t1', 'int', 13002, [], [1]],
                ['_t2', 'int', 13003, [], [1]],
                ['_t3', 'int', 13004, [], [1]],
                ['_t4', 'int', 13005, [], [1]]
        ],
        61,
        ['int', 'int'],
        [6, 0, 0, 1, 0], 5]
]
```

QuadrupleManager.py

La estructura principal que maneja esta clase está definida de la siguiente manera:

[[idOperacion, dirVir1, dirVir2, resDirVir]]

Para esta estructura decidimos utilizar un array, pues estos datos no los estaremos consultando de manera recurrente, y en caso de que queramos acceder a uno en particular, ya conoceremos el subíndice de dicho cuádruplo, es decir, nunca vamos a ‘buscar’ entre todos los cuádruplos uno en específico.

Ej.

```
[  0  13016  7002  13017 ]
```

Donde 0 representa a la operación +.

MemoryManager.py

La memoria está segmentada en 3 tipos de scopes diferentes: ‘global’, ‘constante’ y ‘local’, sin embargo esta segmentación es solo lógica, y no física.

Para la estructura de la memoria decidimos utilizar un diccionario, pues necesitábamos que la memoria estuviera almacenada en una estructura asociativa, donde pudiéramos acceder a una entrada en específico sin iterar por todas las entradas, esto se logra con el diccionario mediante un hash implementado por Python. La estructura de la memoria está definida de la siguiente manera:

{dirVir : valor}

Ej.

```
{
        7002: 1000,
        7003: 10,
        7004: -24
}
```

Ademas se utilizan las siguientes estructuras como estructuras auxiliares para poder construir las anteriormente mencionadas:

```
# Sintaxis y Semantica
DimsList    = []
ParamsList  = []
ArrayIDList    = []
ParamTypeList  = []
FCStack        = Stack()
TypeStack      = Stack()
FunctionStack  = Stack()
PilasDimensionadas = Stack()
```

```
# Generacion de Cuadruplos
POper   = Stack()
PilaO   = Stack()
PSaltos = Stack()
```

3.10.2 Proceso de administración de Memoria durante la compilación

Como se mencionó anteriormente, se implementó una clase llamada *MemoryManager,* la cual tiene los siguientes métodos:
resetMemory()
Resetea la memoria completamente, elimina tanto los contadores como el diccionario. Utilizada si se quiere leer nuevos archivos de código fuente.

showMemory()
Muestra la memoria en pantalla.

resetLocalMemory()
Borra todas las variables que pertenecen al scope local, tanto de memoria como sus contadores.

addEntry(scope, tipo, value)
Se agrega una nueva entrada a la memoria, según su scope y su tipo de dato es sencillo calcular su dirVir asignada con la siguiente fórmula:

```
IndiceContador = len(DataTypes) * IndexScope + IndexType
dirVir = Counters[IndiceContador]
Counters[IndiceContador]++
```

getEntryType (virDir)
Recibe una dirección virtual y retorna el tipo de dato al cual pertenece.

getEntryValue(virDir)
Recibe una dirección virtual y retorna el valor almacenado en ella.

setEntryValue(virDir, value)
Recibe una direccion virtual y asigna el valor dado como argumento.

checkMemoryOverflow()
Verifica que no este sucediendo un error de overflow, se corre cada que alguien le pide una nueva dirección virtual a la memoria.

4 Descripción de la máquina virtual

4.1 Equipo de computo

Como la máquina virtual fue implementada en *Python*, nos permitió utilizar diferentes sistemas operativos durante el desarrollo de este. No se utilizó particularmente una computadora potente, por lo que no es problema utilizarlo en PCs con especificaciones promedio.

4.2 Lenguaje

El único lenguaje utilizado en el desarrollo de la máquina virtual es *Python*, especificamente la version 3.5.

4.3 Utilerías especiales

Fuera del lenguaje propio de python, y sus librerías de matemáticas, copiado e impresiones, no hicimos uso de ninguna utilería externa para el desarrollo de la máquina virtual.

4.4 Descripción detallada del proceso de administración de memoria en ejecución.

4.4.1 Especificación gráfica de cada estructura de datos usada

Las estructuras utilizadas durante el proceso de interpretación son las mismas que se utilizaron que durante el proceso de compilación (una clase administradora de memoria, una clase administradora de cuadruplos, pilas y filas para guardar datos temporales).

4.4.2 Asociación hecha entre las direcciones virtuales (compilación) y reales (ejecución)

Aunque se reutiliza la clase administradora de memoria, fue necesario hacer conversiones entre las direcciones que venian en los cuádruplos y las direcciones que realmente usaba en ejecución, en la clase que implementamos VirtualMachine, tenemos un método específicamente para eso:

```
     def translateVirtualToAbsolute(self, virDir):
          absDir = virDir;
          if self.asigningParams == True:
               actual = self.localTypeCounters.pop()
          if virDir >= self.MemoryManager.getLocalStart():
               actualCounters= self.localTypeCounters.peek()
               virDirType= self.MemoryManager.getEntryTypeId(virDir)
               offSet = virDir -
self.MemoryManager.getInitialIndexType(virDirType)
               absDir = actualCounters[virDirType] + offSet
          if self.asigningParams == True:
               self.localTypeCounters.push(actual)
          return absDir
```

En la cual podemos observar recibe una dirección virtual como parámetro, y retorna la dirección ‘absoluta’ o ‘real’, que es la que se utilizara en la ejecución. Basicamente lo que hace es asignar a la direccion, un índice el cual, sumado con un offSet (de variables utilizadas previamente), nos da como resultado la direccion absoluta.

Cuando se manda llamar una función se guardan los topes de los índices de cada tipo de dato del scope local, para que una vez que esa funcion finalize, se pueda liberar la memoria desde la última direccion que esta utilizó, hasta la dirección que guardamos previamente, este proceso se realiza para cada tipo de dato; esto nos da una pequeña

simulación de como cuando se tiene cargado mas de una función en memoria, como la última funcion está sobre quien la invocó, pero aun asi no se emplaman.

5 Pruebas de funcionamiento del lenguaje

Multiplicación de matrices

```
program matrix;

var string endl = "\n";
var int valor;
var int indice;
var int A[3, 3];
var int B[3, 3];
var int C[3, 3];

function void initVectors(){
        var int i = 0;
        var int j = 0;

        while(i < 3){
         j = 0;

         while(j < 3){
               A[i, j] = random(1, 8);
               B[i, j] = random(55, 100);
               j = j + 1;
         }
         i = i + 1;
        }
        return;
}

function void showVectorA(){
        var int i = 0;
        var int j = 0;
        var int n;
        while(i < 3){
         j = 0;

         while(j < 3){
               n = A[i, j];
               print("A[", i , ", ", j ,"] = ", n, endl);
               j = j + 1;
         }
         i = i + 1;
        }
        return;
}

function void showVectorB(){
        var int i = 0;
        var int j = 0;
        var int n;
```

Cuadruplos

```
#   Name          Quadruple
0   =     [ 12   11000   None   5000 ]
1   era   [ 22   None    None   2 ]
2   goSub [ 19   None    None   176 ]
3   =     [ 12   7000    None   13000 ]
4   =     [ 12   7000    None   13001 ]
5   <     [ 5    13000   7001   16000 ]
6   gotoF [ 17   16000   None   32 ]
7   =     [ 12   7000    None   13001 ]
8   <     [ 5    13001   7001   16001 ]
9   gotoF [ 17   16001   None   29 ]
10  verify [ 23  13000   0      3 ]
11  *     [ 2    13000   7001   13002 ]
12  verify [ 23  13001   0      3 ]
13  *     [ 2    13001   7002   13003 ]
14  +     [ 0    13003   13002  13004 ]
15  +     [ 0    13004   7003   13005 ]
16  random [ 15  7002    7004   13006 ]
17  =     [ 12   13006   None   *13005 ]
18  verify [ 23  13000   0      3 ]
19  *     [ 2    13000   7001   13007 ]
20  verify [ 23  13001   0      3 ]
21  *     [ 2    13001   7002   13008 ]
22  +     [ 0    13008   13007  13009 ]
23  +     [ 0    13009   7005   13010 ]
24  random [ 15  7006    7007   13011 ]
25  =     [ 12   13011   None   *13010 ]
26  +     [ 0    13001   7002   13012 ]
27  =     [ 12   13012   None   13001 ]
28  goto  [ 18   None    None   8 ]
29  +     [ 0    13000   7002   13013 ]
30  =     [ 12   13013   None   13000 ]
31  goto  [ 18   None    None   5 ]
32  return [ 21  None    None   None ]
33  =     [ 12   7000    None   13000 ]
34  =     [ 12   7000    None   13001 ]
35  <     [ 5    13000   7001   16000 ]
36  gotoF [ 17   16000   None   60 ]
37  =     [ 12   7000    None   13001 ]
38  <     [ 5    13001   7001   16001 ]
39  gotoF [ 17   16001   None   57 ]
40  verify [ 23  13000   0      3 ]
41  *     [ 2    13000   7001   13003 ]
42  verify [ 23  13001   0      3 ]
43  *     [ 2    13001   7002   13004 ]
44  +     [ 0    13004   13003  13005 ]
45  +     [ 0    13005   7003   13006 ]
```

```
        while(i < 3){
         j = 0;

         while(j < 3){
              n = B[i, j];
              print("B[", i , ", ", j ,"] = ", n, endl);
              j = j + 1;
         }
         i = i + 1;
        }
        return;
}

function void showVectorC(){
        var int i = 0;
        var int j = 0;
        var int n;

        while(i < 3){
         j = 0;
         while(j < 3){
              n = C[i, j];
              print("C[", i , ", ", j ,"] = ", n, endl);
              j = j + 1;
         }
         i = i + 1;
        }
        return;
}

function void multiplyVectors(){
        var int i = 0;
        var int j = 0;
        var int k = 0;

        var int nA, nB, nC, res;

        while(i < 3){
         j = 0;
         while(j < 3){
              k = 0;
              C[i, j] = 0;
              while(k < 3){
                  nA = A[i, k];
                  nB = B[k, j];
                  nC = C[i, j];
                  res = nC +  nA * nB;
                  C[i, j] = res;

                  k = k + 1;
              }
              j = j + 1;
         }
         i = i + 1;
        }
```

```
46  =      [ 12  *13006   None   13002 ]
47  print  [ 13   None    None   11001 ]
48  print  [ 13   None    None   13000 ]
49  print  [ 13   None    None   11002 ]
50  print  [ 13   None    None   13001 ]
51  print  [ 13   None    None   11003 ]
52  print  [ 13   None    None   13002 ]
53  print  [ 13   None    None    5000 ]
54  +      [  0   13001   7002   13007 ]
55  =      [ 12   13007   None   13001 ]
56  goto   [ 18   None    None      38 ]
57  +      [  0   13000   7002   13008 ]
58  =      [ 12   13008   None   13000 ]
59  goto   [ 18   None    None      35 ]
60  return [ 21   None    None    None ]
61  =      [ 12   7000    None   13000 ]
62  =      [ 12   7000    None   13001 ]
63  <      [  5   13000   7001   16000 ]
64  gotoF  [ 17   16000   None      88 ]
65  =      [ 12   7000    None   13001 ]
66  <      [  5   13001   7001   16001 ]
67  gotoF  [ 17   16001   None      85 ]
68  verify [ 23   13000      0       3 ]
69  *      [  2   13000   7001   13003 ]
70  verify [ 23   13001      0       3 ]
71  *      [  2   13001   7002   13004 ]
72  +      [  0   13004   13003  13005 ]
73  +      [  0   13005   7005   13006 ]
74  =      [ 12  *13006   None   13002 ]
75  print  [ 13   None    None   11004 ]
76  print  [ 13   None    None   13000 ]
77  print  [ 13   None    None   11002 ]
78  print  [ 13   None    None   13001 ]
79  print  [ 13   None    None   11003 ]
80  print  [ 13   None    None   13002 ]
81  print  [ 13   None    None    5000 ]
82  +      [  0   13001   7002   13007 ]
83  =      [ 12   13007   None   13001 ]
84  goto   [ 18   None    None      66 ]
85  +      [  0   13000   7002   13008 ]
86  =      [ 12   13008   None   13000 ]
87  goto   [ 18   None    None      63 ]
88  return [ 21   None    None    None ]
89  =      [ 12   7000    None   13000 ]
90  =      [ 12   7000    None   13001 ]
91  <      [  5   13000   7001   16000 ]
92  gotoF  [ 17   16000   None     116 ]
93  =      [ 12   7000    None   13001 ]
94  <      [  5   13001   7001   16001 ]
95  gotoF  [ 17   16001   None     113 ]
96  verify [ 23   13000      0       3 ]
97  *      [  2   13000   7001   13003 ]
98  verify [ 23   13001      0       3 ]
99  *      [  2   13001   7002   13004 ]
100 +      [  0   13004   13003  13005 ]
101 +      [  0   13005   7008   13006 ]
```

```
        return;
}

main {
        initVectors();
        showVectorA();
        showVectorB();
        multiplyVectors();
        showVectorC();
}
```

```
102 =      [ 12 *13006    None   13002 ]
103 print  [ 13   None    None   11005 ]
104 print  [ 13   None    None   13000 ]
105 print  [ 13   None    None   11002 ]
106 print  [ 13   None    None   13001 ]
107 print  [ 13   None    None   11003 ]
108 print  [ 13   None    None   13002 ]
109 print  [ 13   None    None    5000 ]
110 +      [  0   13001   7002   13007 ]
111 =      [ 12   13007   None   13001 ]
112 goto   [ 18   None    None      94 ]
113 +      [  0   13000   7002   13008 ]
114 =      [ 12   13008   None   13000 ]
115 goto   [ 18   None    None      91 ]
116 return [ 21   None    None   None ]
117 =      [ 12    7000   None   13000 ]
118 =      [ 12    7000   None   13001 ]
119 =      [ 12    7000   None   13002 ]
120 <      [  5   13000   7001   16000 ]
121 gotoF  [ 17   16000   None     175 ]
122 =      [ 12    7000   None   13001 ]
123 <      [  5   13001   7001   16001 ]
124 gotoF  [ 17   16001   None     172 ]
125 =      [ 12    7000   None   13002 ]
126 verify [ 23   13000      0      3 ]
127 *      [  2   13000   7001   13007 ]
128 verify [ 23   13001      0      3 ]
129 *      [  2   13001   7002   13008 ]
130 +      [  0   13008   13007  13009 ]
131 +      [  0   13009   7008   13010 ]
132 =      [ 12    7000   None *13010 ]
133 <      [  5   13002   7001   16002 ]
134 gotoF  [ 17   16002   None     169 ]
135 verify [ 23   13000      0      3 ]
136 *      [  2   13000   7001   13011 ]
137 verify [ 23   13002      0      3 ]
138 *      [  2   13002   7002   13012 ]
139 +      [  0   13012   13011  13013 ]
140 +      [  0   13013   7003   13014 ]
141 =      [ 12 *13014    None   13003 ]
142 verify [ 23   13002      0      3 ]
143 *      [  2   13002   7001   13015 ]
144 verify [ 23   13001      0      3 ]
145 *      [  2   13001   7002   13016 ]
146 +      [  0   13016   13015  13017 ]
147 +      [  0   13017   7005   13018 ]
148 =      [ 12 *13018    None   13004 ]
149 verify [ 23   13000      0      3 ]
150 *      [  2   13000   7001   13019 ]
151 verify [ 23   13001      0      3 ]
152 *      [  2   13001   7002   13020 ]
153 +      [  0   13020   13019  13021 ]
154 +      [  0   13021   7008   13022 ]
155 =      [ 12 *13022    None   13005 ]
156 *      [  2   13003   13004  13023 ]
157 +      [  0   13005   13023  13024 ]
```

```
158 =      [ 12   13024    None   13006 ]
159 verify [ 23   13000      0      3 ]
160 *      [ 2   13000    7001   13025 ]
161 verify [ 23   13001      0      3 ]
162 *      [ 2   13001    7002   13026 ]
163 +      [ 0   13026   13025   13027 ]
164 +      [ 0   13027    7008   13028 ]
165 =      [ 12   13006    None  *13028 ]
166 +      [ 0   13002    7002   13029 ]
167 =      [ 12   13029    None   13002 ]
168 goto   [ 18    None    None    133 ]
169 +      [ 0   13001    7002   13030 ]
170 =      [ 12   13030    None   13001 ]
171 goto   [ 18    None    None    123 ]
172 +      [ 0   13000    7002   13031 ]
173 =      [ 12   13031    None   13000 ]
174 goto   [ 18    None    None    120 ]
175 return [ 21    None    None    None ]
176 era    [ 22    None    None      3 ]
177 goSub  [ 19    None    None      3 ]
178 era    [ 22    None    None      4 ]
179 goSub  [ 19    None    None     33 ]
180 era    [ 22    None    None      5 ]
181 goSub  [ 19    None    None     61 ]
182 era    [ 22    None    None      7 ]
183 goSub  [ 19    None    None    117 ]
184 era    [ 22    None    None      6 ]
185 goSub  [ 19    None    None     89 ]
186 end    [ 24    None    None    None ]
```

Sort de un vector

```
program find;

var string endl = "\n";
var int vector[10];

main {
        var int i = 0;
        var int j = 0;
        var int cantidad = 10;
        var int pasada = 1;

        while(i < 10){
        vector[i] = random(-50, 50);
        i = i + 1;
        }

        print("Vector generado aleatoriamente", endl);
        i = 0;
        while(i < 10){
```

```
#   Name            Quadruple
0   =      [ 12   11000    None    5000 ]
1   era    [ 22    None    None      2 ]
2   goSub  [ 19    None    None      3 ]
3   =      [ 12    7000    None   13000 ]
4   =      [ 12    7000    None   13001 ]
5   =      [ 12    7001    None   13002 ]
6   =      [ 12    7002    None   13003 ]
7   <      [ 5   13000    7001   16000 ]
8   gotoF  [ 17   16000    None      17 ]
9   verify [ 23   13000      0      10 ]
10  *      [ 2   13000    7002   13004 ]
11  +      [ 0   13004    7003   13005 ]
12  random [ 15    7004    7005   13006 ]
13  =      [ 12   13006    None  *13005 ]
14  +      [ 0   13000    7002   13007 ]
15  =      [ 12   13007    None   13000 ]
16  goto   [ 18    None    None      7 ]
17  print  [ 13    None    None   11001 ]
```

```
        var int n;
        n = vector[i];
        print("vector[", i, "] = ", n, endl);
        i = i + 1;
        }

        var bool intercambio = True;

        while(pasada < cantidad){
        if(intercambio){
                intercambio = False;
                j = 0;
                var int resta;
                resta = cantidad - pasada;

                while(j < resta){
                var int a = vector[j];
                var int b = vector[j + 1];
                if(a > b){
                        var int aux;
                        aux = vector[j];

                        var int aux2;
                    aux2 = vector[j + 1];

                        vector[j] = aux2;
                        vector[j + 1] = aux;
                        intercambio = True;
                }
                j = j + 1;
                }
                pasada = pasada + 1;
        }
         else{
                pasada = cantidad;
        }
        }

        print(endl, "Vector ordenado", endl);
        i = 0;
        while(i < 10){
        var int n;
        n = vector[i];
        print("vector[", i, "] = ", n, endl);
        i = i + 1;
        }

}
```

```
18  print  [ 13   None   None   5000 ]
19  =      [ 12   7000   None   13000 ]
20  <      [ 5   13000   7001   16001 ]
21  gotoF  [ 17   16001   None    34 ]
22  verify [ 23   13000     0     10 ]
23  *      [ 2   13000   7002   13009 ]
24  +      [ 0   13009   7003   13010 ]
25  =      [ 12  *13010   None   13008 ]
26  print  [ 13   None   None   11002 ]
27  print  [ 13   None   None   13000 ]
28  print  [ 13   None   None   11003 ]
29  print  [ 13   None   None   13008 ]
30  print  [ 13   None   None   5000 ]
31  +      [ 0   13000   7002   13011 ]
32  =      [ 12   13011   None   13000 ]
33  goto   [ 18   None   None    20 ]
34  =      [ 12   10000   None   16002 ]
35  <      [ 5   13003   13002   16003 ]
36  gotoF  [ 17   16003   None    82 ]
37  gotoF  [ 17   16002   None    80 ]
38  =      [ 12   10001   None   16002 ]
39  =      [ 12   7000   None   13001 ]
40  -      [ 1   13002   13003   13013 ]
41  =      [ 12   13013   None   13012 ]
42  <      [ 5   13001   13012   16004 ]
43  gotoF  [ 17   16004   None    77 ]
44  verify [ 23   13001     0     10 ]
45  *      [ 2   13001   7002   13014 ]
46  +      [ 0   13014   7003   13015 ]
47  =      [ 12  *13015   None   13016 ]
48  +      [ 0   13001   7002   13017 ]
49  verify [ 23   13017     0     10 ]
50  *      [ 2   13017   7002   13018 ]
51  +      [ 0   13018   7003   13019 ]
52  =      [ 12  *13019   None   13020 ]
53  >      [ 4   13016   13020   16005 ]
54  gotoF  [ 17   16005   None    74 ]
55  verify [ 23   13001     0     10 ]
56  *      [ 2   13001   7002   13022 ]
57  +      [ 0   13022   7003   13023 ]
58  =      [ 12  *13023   None   13021 ]
59  +      [ 0   13001   7002   13025 ]
60  verify [ 23   13025     0     10 ]
61  *      [ 2   13025   7002   13026 ]
62  +      [ 0   13026   7003   13027 ]
63  =      [ 12  *13027   None   13024 ]
64  verify [ 23   13001     0     10 ]
65  *      [ 2   13001   7002   13028 ]
66  +      [ 0   13028   7003   13029 ]
67  =      [ 12   13024   None  *13029 ]
68  +      [ 0   13001   7002   13030 ]
69  verify [ 23   13030     0     10 ]
70  *      [ 2   13030   7002   13031 ]
71  +      [ 0   13031   7003   13032 ]
72  =      [ 12   13021   None  *13032 ]
73  =      [ 12   10000   None   16002 ]
```

```
74  +     [  0   13001   7002   13033 ]
75  =     [  12  13033   None   13001 ]
76  goto  [  18  None    None   42 ]
77  +     [  0   13003   7002   13034 ]
78  =     [  12  13034   None   13003 ]
79  goto  [  18  None    None   81 ]
80  =     [  12  13002   None   13003 ]
81  goto  [  18  None    None   35 ]
82  print [  13  None    None   5000 ]
83  print [  13  None    None   11004 ]
84  print [  13  None    None   5000 ]
85  =     [  12  7000    None   13000 ]
86  <     [  5   13000   7001   16006 ]
87  gotoF [  17  16006   None   100 ]
88  verify [ 23  13000   0      10 ]
89  *     [  2   13000   7002   13035 ]
90  +     [  0   13035   7003   13036 ]
91  =     [  12  *13036  None   13008 ]
92  print [  13  None    None   11002 ]
93  print [  13  None    None   13000 ]
94  print [  13  None    None   11003 ]
95  print [  13  None    None   13008 ]
96  print [  13  None    None   5000 ]
97  +     [  0   13000   7002   13037 ]
98  =     [  12  13037   None   13000 ]
99  goto  [  18  None    None   86 ]
100 end   [  24  None    None   None ]
```

Factorial iterativo

```
program factIterativo;

function int factorial (int n) {
        var int ret = 1;
        var int i = 2;

        if(n == 0 | n == 1){
          ret = 1;
        }

        while(i <= n){
          ret = ret * i;
          i = i + 1;
        }

        return ret;
}
```

```
#   Name          Quadruple
0   era     [ 22   None   None    2 ]
1   goSub   [ 19   None   None    18 ]
2   =       [ 12   7000   None   13001 ]
3   =       [ 12   7001   None   13002 ]
4   ==      [  9   13000  7002   16000 ]
5   ==      [  9   13000  7000   16001 ]
6   |       [ 10   16000  16001  16002 ]
7   gotoF   [ 17   16002  None    9 ]
8   =       [ 12   7000   None   13001 ]
9   <=      [  7   13002  13000  16003 ]
10  gotoF   [ 17   16003  None    16 ]
11  *       [  2   13001  13002  13003 ]
12  =       [ 12   13003  None   13001 ]
13  +       [  0   13002  7000   13004 ]
14  =       [ 12   13004  None   13002 ]
15  goto    [ 18   None   None    9 ]
16  =       [ 12   13001  None   1000 ]
```

```
main {
        var string endl = "\n";
        var int num, fact;

        print("Factorial de: ");
        read(num);
        fact = factorial(num);

        print(num, "! = ", fact, endl);

}
```

```
17  return [ 21   None   None   1000 ]
18  =      [ 12   11000  None   17000 ]
19  print  [ 13   None   None   11001 ]
20  read   [ 14   None   None   13000 ]
21  era    [ 22   None   None   3 ]
22  params [ 20   13000  None   13000 ]
23  goSub  [ 19   None   None   2 ]
24  =      [ 12   1000   None   13002 ]
25  =      [ 12   13002  None   13001 ]
26  print  [ 13   None   None   13000 ]
27  print  [ 13   None   None   11002 ]
28  print  [ 13   None   None   13001 ]
29  print  [ 13   None   None   17000 ]
30  end    [ 24   None   None   None ]
```

Fibonacci Recursivo Terminal

```
program fibRecursivo;

var int iter = 0;

function int tailFibonacci(int n0, int n1, int n2){
        /*
    In tail recursion, you perform your calculations first, and then you
        execute the recursive call, passing the results of your current step to
        the next recursive step. Basically, the return value of any given recursive
        step is the same as the return value of the next recursive call.
  */
        var int ret;

        iter = iter + 1;

        if (n0 <= 0) {
        ret = n2;
        }

        else {
        ret = tailFibonacci(n0 - 1, n1 + n2, n1);
```

```
#   Name            Quadruple
0   era    [ 22   None   None   2 ]
1   goSub  [ 19   None   None   18 ]
2   =      [ 12   7000   None   13001 ]
3   =      [ 12   7001   None   13002 ]
4   ==     [ 9    13000  7002   16000 ]
5   ==     [ 9    13000  7000   16001 ]
6   |      [ 10   16000  16001  16002 ]
7   gotoF  [ 17   16002  None   9 ]
8   =      [ 12   7000   None   13001 ]
9   <=     [ 7    13002  13000  16003 ]
10  gotoF  [ 17   16003  None   16 ]
11  *      [ 2    13001  13002  13003 ]
12  =      [ 12   13003  None   13001 ]
13  +      [ 0    13002  7000   13004 ]
14  =      [ 12   13004  None   13002 ]
15  goto   [ 18   None   None   9 ]
16  =      [ 12   13001  None   1000 ]
17  return [ 21   None   None   1000 ]
17  return [ 21   None   None   1000 ]
18  =      [ 12   11000  None   17000 ]
19  print  [ 13   None   None   11001 ]
20  read   [ 14   None   None   13000 ]
21  era    [ 22   None   None   3 ]
22  params [ 20   13000  None   13000 ]
23  goSub  [ 19   None   None   2 ]
```

```
        }

        return ret;
}

function int fibonacci(int n){
        return tailFibonacci(n, 1, 0);
}

main {
        var string endl = "\n";
        var int fib, num;

        print("Fibonacci de: ");
        read(num);
        fib = fibonacci(num);

        print("Fib(", num, ") = ", fib, endl);
        print("Iteraciones fib: ", iter, endl);

}
```

```
24  =      [ 12   1000   None   13002 ]
25  =      [ 12   13002   None   13001 ]
26  print  [ 13   None   None   13000 ]
27  print  [ 13   None   None   11002 ]
28  print  [ 13   None   None   13001 ]
29  print  [ 13   None   None   17000 ]
30  end    [ 24   None   None   None ]
```