INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Documentación

OBSIDIAN

César Armando Galván Valles
A00814038

Luis Mario Díaz Rincón
A00343755

Tabla de Contenidos

Descripción del Proyecto

Propósito y Visión del Lenguaje

El lenguaje de programación *Obsidian,* que se implementará como proyecto de esta clase, tiene como propósito el ser utilizado por jóvenes para que puedan aprender sobre la lógica de la programación al arrastrar bloques de código de una manera gráfica sobre un plano.

Objetivo del Lenguaje

Nuestro objetivo es implementar una solución efectiva y simple que ayude a que los jóvenes aprendan de una manera diferente, permitiendo un nuevo método efectivo de enseñanza para que los jóvenes aprendan el uso de ciclos, condiciones, estatutos de asignación, expresiones matemáticas, tipos de datos y funciones.

Alcance del Proyecto

El alcance del lenguaje *Obsidian* va de la mano con nuestro objetivo de hacer un lenguaje simple en el cual sea fácil aprender las bases de programación, en este permitimos al usuario declarar variables, hacer operaciones matemáticas con ellas, definir ciclos ¨while¨, utilizar operadores lógicos en condicionales, definir funciones normales y recursivas. También ofrecerle al usuario la capacidad de utilizar funciones de I/O como Read y Write.

Análisis de Requerimientos

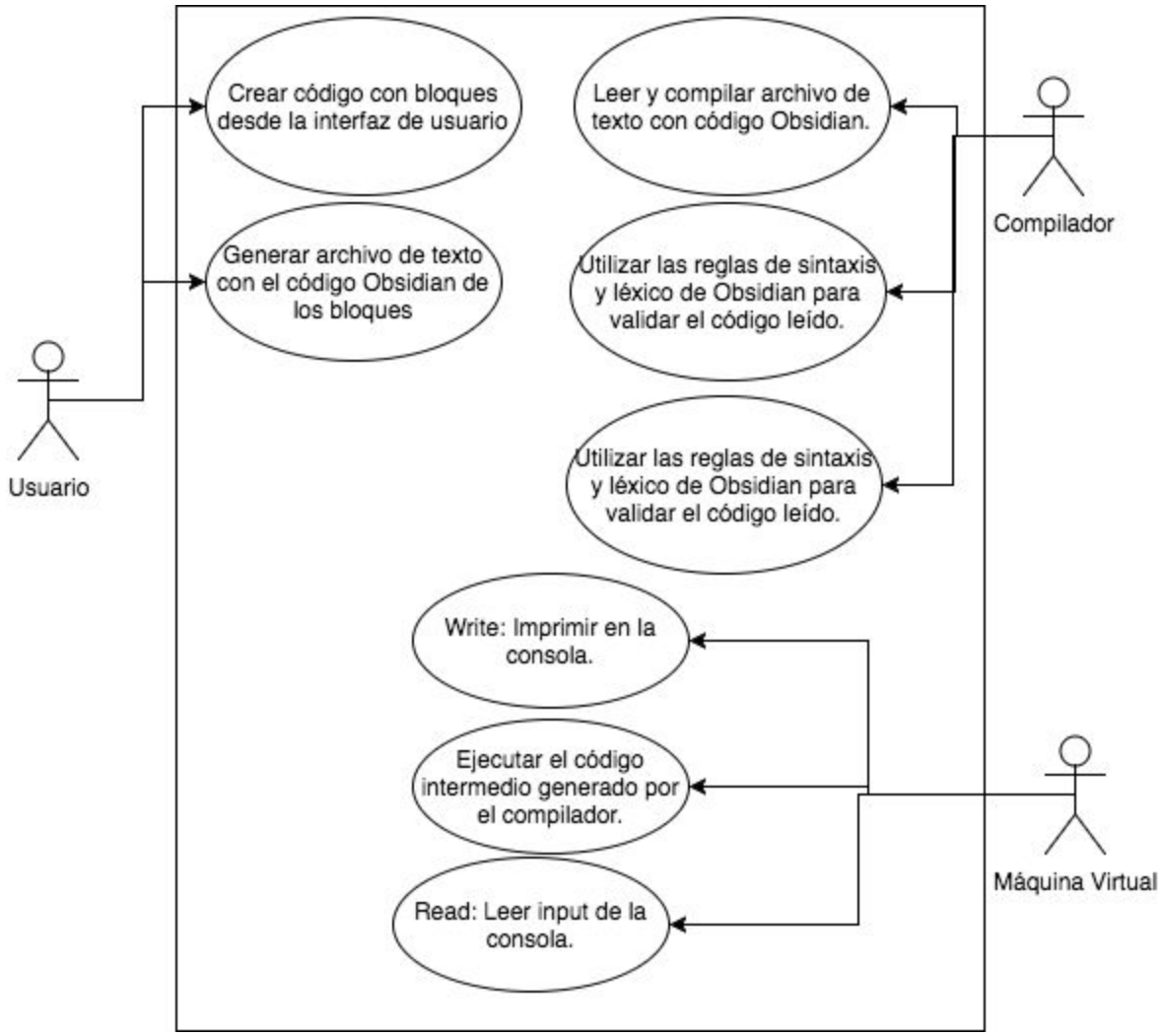
Requerimientos funcionales

- El Compilador debe de leer y compilar código *Obsidian* válido.
- El Compilador debe generar código intermedio para que sea ejecutado por la máquina virtual.
- En caso de error, el compilador debe parar su ejecución e indicar al usuario el tipo de error y en que linea se generó el mismo.
- La Máquina Virtual debe ejecutar el código intermedio generado por el compilador.
- La Interfaz de Usuario debe permitir generar código *Obsidian* válido a partir de bloques, siempre y cuando el usuario sigue las reglas del lenguaje.

Requerimientos no funcionales

- La Interfaz de Usuario debe ser de fácil uso para personas con poca experiencia programando.
- La ejecución del código debe ser rápida.

Casos de Uso Generales



Descripción de Casos de Prueba

- Prueba de generación de cuádruplos de sentencias if/else
- Pruebas de léxico
- Pruebas de sintaxis
- Prueba de generación de cuádruplos de sentencias while
- Prueba de generación de cuádruplos de declaración de variables
- Prueba de generación de cuádruplos de declaración de funciones
- Prueba de generación de cuádruplos de llamadas de funciones
- Prueba de generación de cuádruplos de declaración de arreglos
- Prueba de generación de cuádruplos de acceso a arreglos

- Prueba de ejecución de sentencias if, else, while en la máquina virtual
- Prueba de ejecución de arreglos en la máquina virtual
- Fibonacci Recursivo
- Fibonacci Ciclico
- Factorial Recursivo
- Factorial Ciclico
- Bubble Sort
- Multiplicación de Matrices
- Prueba de Print()
- Prueba de Read()

Logs del Desarrollo del Proyecto

Aprendí muchísimo durante el desarrollo de Obsidian. Ha sido creo, uno de los proyectos más interesantes durante la carrera y me ayudó a comprender lo que pasa detrás cuando corro código para el trabajo o para cualquier otra clase. El proceso de desarrollo fue interesante, decidimos utilizar pair programming como nuestro método de desarrollo pues el compilador es complicado y creíamos que encontraríamos mejores soluciones si estas eran diseñadas en conjunto, mismo caso con los problemas que se nos iban presentando. Ya en la última etapa, cuando empezó a entrar la presión del tiempo empezamos a dividir cargas para asegurar la entrega a tiempo del proyecto.

_____

Luis Mario Díaz
A00343755

Mi experiencia en este proyecto fue muy gratificante ya que ha sido el proyecto más retador que he tenido en la carrera, aprendí mucho en clase y mucho más al desarrollar el proyecto en sí, algo que no había obtenido de no haber tenido que implementar el conocimiento adquirido en clase. Durante el proceso fuimos descubriendo mejores maneras de hacer las cosas lo cual es un gran avance dentro de nuestra formación como profesionistas de calidad y creo que al final del día es lo que más me llevo del proyecto.

_____

Cesar Armando Galvan
A00814038

Descripción del Lenguaje

Nombre

Obsidian

Descripción de Características Principales del Lenguaje

Los lenguajes más utilizados en el mundo tienen una sintaxis basada en C que puede llegar a ser algo compleja para principiantes. En Obsidian decidimos basarnos en el lenguaje C, sin embargo intentamos hacerlo mucho más simple ya que estamos enfocados a personas principiantes programando. Básicamente el objetivo de Obsidian es introducir a novatos programando a la sintaxis de programación C, para que una vez que estén familiarizados con este tipo de sintaxis, ya puedan dar el paso a algo más complejo como lo es C o Java.

Obsidian tiene las funciones más importantes para que un principiante pueda empezar a programar sin verse agobiado por la cantidad de opciones que ofrecen lenguajes más avanzados. Declaraciones de variables, ciclos while, creación y llamadas a funciones, llamadas a funciones, declaración y uso de arreglos, y funciones fundamentales de I/O como write y print.

Descripción de Errores

Errores de compilación:

- Error de Sintaxis
- Error: Uso de variable o función no declarada
- Error: Llamar a una función con el número incorrecto de parámetros
- Error: Inicializar un arreglo de tamaño incompatible
- Error: Utilizar operador con operando no compatible
- Error de memoria llena

Errores en la máquina virtual:

- Direccion de memoria sin valor
- Operación no permitida

Descripción del Compilador

Lenguaje, Equipo y Utilidades utilizados en Desarrollo

Durante el desarrollo del lenguaje utilizamos dos Macbook Pro de 13 pulgadas, una con la versión 10.12 del sistema operativo y otra con la versión 10.11.

El lenguaje que se utilizó fue Python 2.7 y para la interfaz de usuario utilizamos HTML, Javascript y CSS, esto en conjunto con la librería Blockly.

Descripción del Análisis Léxico

Expresiones Regulares

```
t_ignore = ' \t'                t_PLUS = r'\+'
t_MINUS = r'-'                  t_MULTIPLICATION = r'\*'
t_DIVISION = r'/'               t_MOD = r'%'
t_EQUALS = r'='                 t_EQUALEQUALS = r'=='
t_DIFFERENT = r'!='             t_GREATER = r'>'
t_LESS = r'<'                   t_GREATEROREQUAL = r'>='
t_LESSOREQUAL = r'<='           t_AND = r'&&'
t_OR = r'\|\|'                  t_LPAR = r'\('
t_RPAR = r'\)'                  t_LBRACKET = r'{'
t_RBRACKET = r'}'               t_LSQRTBRACKET = r'\['
t_RSQRTBRACKET = r'\]'          t_COMMA = r','
t_SEMICOLON = r';'              t_CTEDOUBLE = r'-?[0-9]+\.[0-9]+'
t_CTEINT = r'-?[0-9]+'          t_CTEBOOL = r'false|true'
t_ID = r'[a-z][a-zA-Z0-9]*'     t_newline = r'\n+'
```

Enumeración de Tokens del Lenguaje y Código Asociado

Palabras Reservadas:

```
'int' : 'INT'
'double' : 'DOUBLE',
'bool' : 'BOOL',
'func' : 'FUNC',
'void' : 'VOID',
'if' : 'IF',
'else' : 'ELSE',
'main' : 'MAIN',
'while' : 'WHILE',
'read' : 'READ',
'write' : 'WRITE',
'return' : 'RETURN'
```

Tokens

```
‘PLUS’  :  ` + `                    ‘MINUS’  :  ` - `
```

```
‘DIVISION’  :  ` / `                    ‘MULTIPLICATION’  :  ` * `
‘MOD’  :  ` % `                         ‘EQUALS’  :  ` = `
‘EQUALSEQUALS’  :  ` == `               ‘DIFFERENT’  :  ` != `
‘GREATER’  : ` > `                      ‘LESS’  :  ` < `
‘GREATEROREQUAL’  :  ` >= `             ‘LESSOREQUAL’  :  ` <= `
‘AND’  :  ` && `                        ‘OR’  :  ` || `
‘LPAR’  :  ` ( `                        ‘RPAR’  : ` ) `
‘LBRACKET’  :  `  {  `                  ‘RBRACKET’  :  ` } `
‘LSQRTBRACKET’  :  ` ] `                ‘RSQRTBRACKET’  :  ` [ `
‘COMMA’  : ` , `                        ‘SEMICOLON’  : ` ; `
```

Descripción del Análisis de Sintaxis

Gramática usada para representar estructuras sintácticas

```
PROGRAM  ::= VARS* FUNC* MAIN

VARS ::= VAR_TYPE 'id' ('[' 'cte' ']')? (('=' VAR_CTE) | ) (',' 'id' ('[' 'cte' ']')? (('='
VAR_CTE) | ))* ';'

VAR_TYPE ::= ('bool' | ' int' | 'double')

VAR_CTE ::= ('cte_int' | 'cte_double' | ' cte_bool' | 'id' ('[' 'cte' ']')? | FUNC_CALL)

FUNC ::= 'func' FUNC_TYPE id '(' ( VAR_TYPE 'id' (',' VAR_TYPE 'id')*)? ')' FUNC_BLOCK

FUNC_TYPE ::= ('void' | 'bool' | ' int' | 'double')

FUNC_BLOCK ::= '{' VARS* STATEMENT* '}'

STATEMENT ::= ( READ | WRITE | CYCLE | CONDITION | ASSIGNATION | FUNC_CALL | RETURN)

READ ::= 'read' '(' 'id' ('[' EXP ']')? ')' ';'

WRITE ::= 'write' '(' EXP ')' ';'

CYCLE ::= 'while' '(' EXPRESSION ')' BLOCK

CONDITION ::= 'if' '(' EXPRESSION ')' BLOCK ("else" BLOCK)?

ASSIGNATION ::= 'id' ('[' EXP ']')* '=' EXPRESSION ';'

FUNC_CALL ::= 'id' '(' (EXP (',' EXP)*)? ')' ';'

RETURN ::= 'return' EXP ';'
```

```
BLOCK ::= '{' STATEMENT* '}'

EXPRESSION ::= CONC ( (AO CONC)+)?

CONC ::= EXP (COMP EXP)?

EXP ::= TERM (PL TERM)*

TERM ::= FACTOR (DM FACTOR)*

FACTOR ::= ( '(' EXPRESSION ')' ) | VAR_CTE

AO ::= '&&' | '||'

COMP ::= ('<' | '>' | '<=' | '>=' | '==' | '!=')

PL ::= '+' | '-'

DM ::= '*' | '/'

MAIN ::= 'main' MAIN_BLOCK

MAIN_BLOCK ::= '{' VARS* STATEMENT* '}'
```

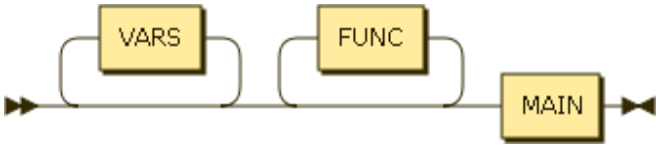
Descripción de Generación de Código intermedio y Análisis Semántico

Código de operación y direcciones virtuales asociadas a elementos del código

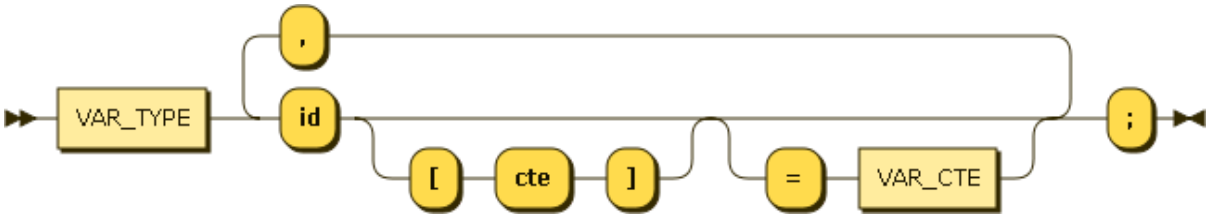
	Globales	Variable	Temporales	Constantes
int	100,000 - 100,499	101,500 - 102,499	104,500 - 105,999	109,000 - 109,999
double	100,500 - 100,999	102,500 - 103,499	106,000 - 107,499	110,000 - 110,999
bool	101,000 - 101,499	103,500 - 104,499	107,500 - 108,499	111,000 - 111,999

Diagramas de Sintaxis

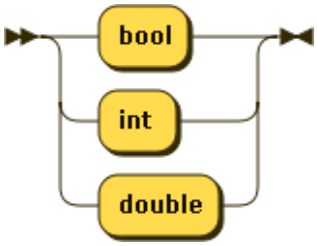
<PROGRAM>



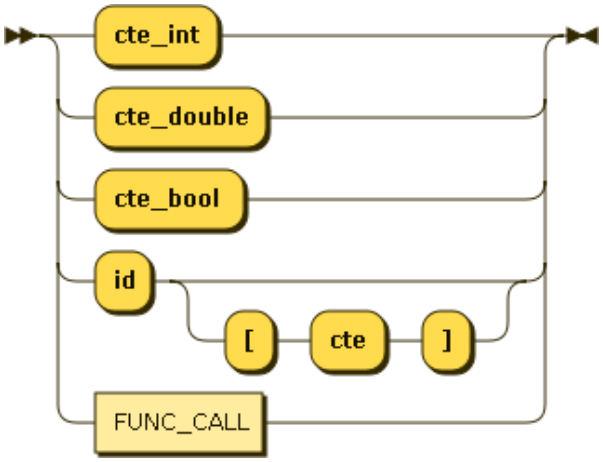
<VARS>



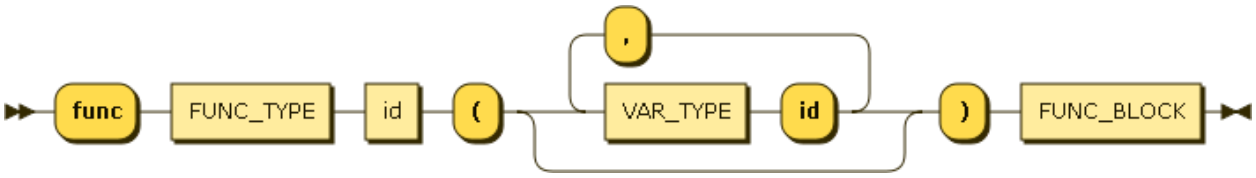
<VAR_TYPE>



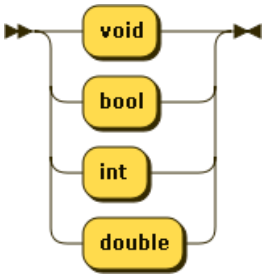
<VAR_CTE>



<FUNC>

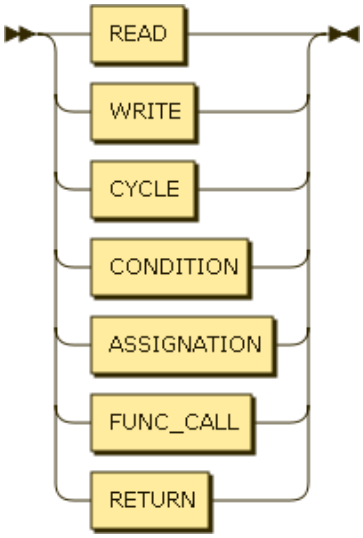


<FUNC_TYPE>

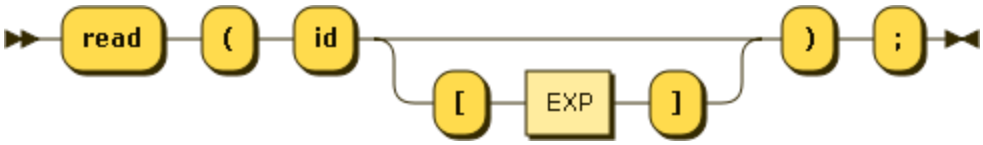


<FUNC_BLOCK>

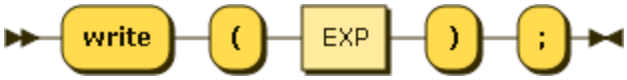
<STATEMENT>



<READ>

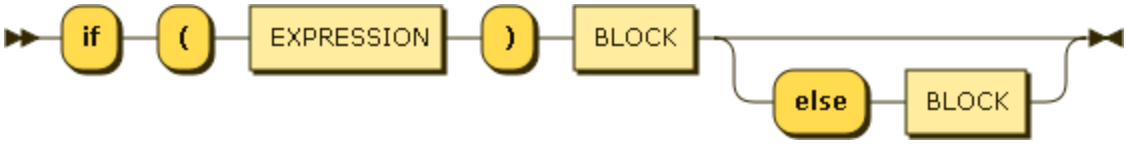


<WRITE>

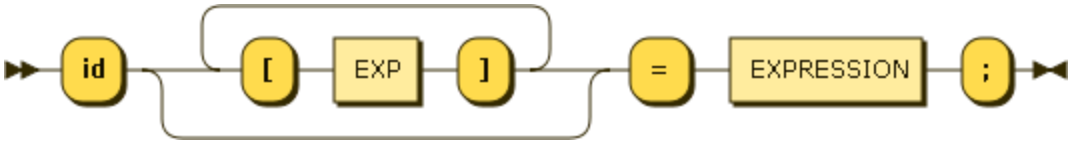


<CYCLE>

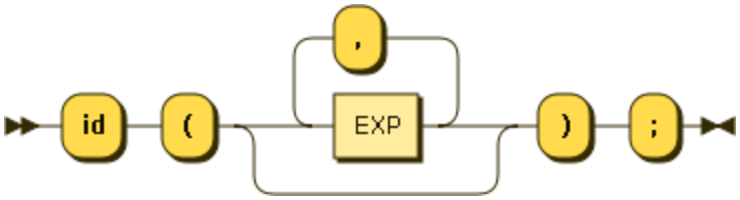
<CONDITION>



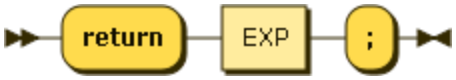
<ASSIGNATION>



<FUNC_CALL>

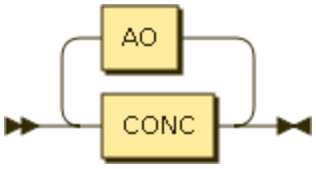


<RETURN>

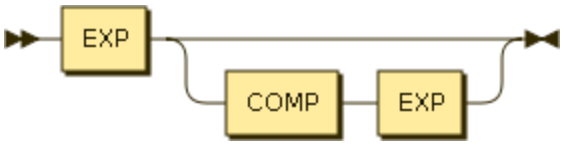


<BLOCK>

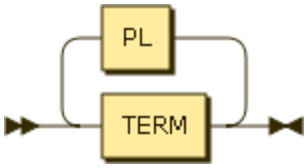
<EXPRESSION>



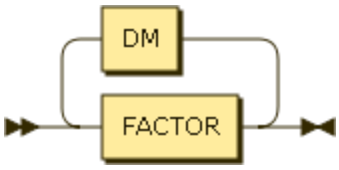
<CONC>



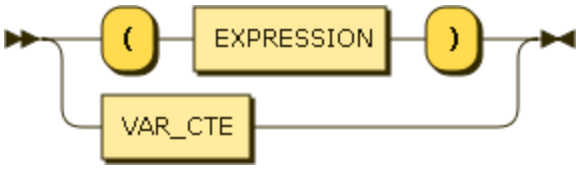
<EXP>



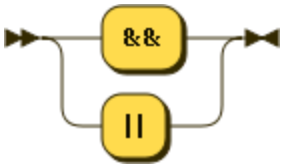
<TERM>



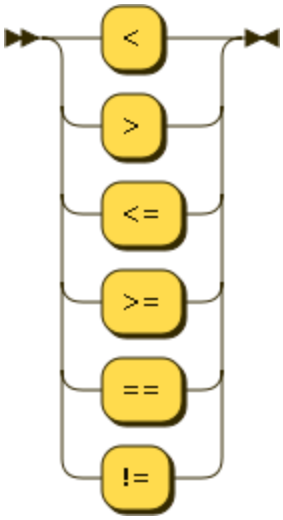
<FACTOR>



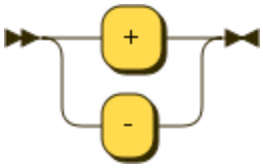
<AO>



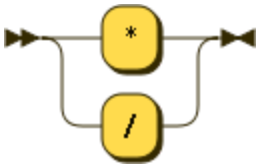
<COMP>



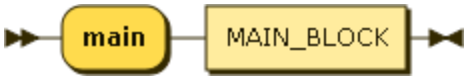
<PL>



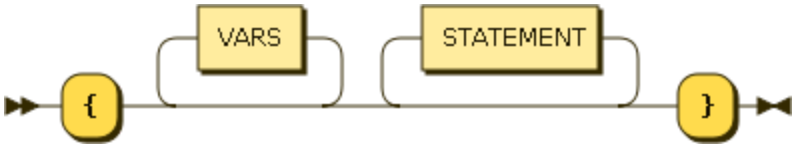
<DM>



<MAIN>



<MAIN_BLOCK>



Breve descripción de las acciones semánticas y de código

1.- Genera el cuádruplo Goto main.
2.- Se inicializa el scope ‘constants’.
3.- Se guarda el id de la o las variables globales.
4.- Si es declaración múltiple de variables, se guarda en un arreglo el tipo de las variables.
5.- Si es un arreglo se revisa que la declaración del arreglo sea con ints.
6.- Se registra el espacio del arreglo.
7.- Se guarda en la tabla de variables con su scope correspondiente la dirección y el tipo de la variable.
8.- Se obtiene la dirección de memoria, memAvail al scope de la variable
9.- Se añade a memoria el espacio de la variable a partir de la dirección
6.- Genera el cuádruplo de asignación de la variable.
8.- Se genera la dimensión de la función dentro del scope global.
9.- Se guarda el tipo y la dimensión de la función en su scope.
10.- Se guarda la dirección de la función, haciendo primero avail a la memoria.
11.- Se añade al directorio de procedimientos la función con su func_type y sus argumentos.
12.- Se añade a memoria el espacio de la función a partir de la dirección.
13.- Guarda en el directorio de procedimientos de la función los argumentos de la misma.
14.- Genera el cuádruplo ‘Endproc’
15.- Reinicia los contadores de memoria a su base.
16.- Añade a la pila de operandos la dirección de memoria del operando y la pila de tipos.

17.- Genera el cuádruplo ‘Read’
18.- Genera el cuádruplo ‘Write’
19.- Genera el cuádruplo ‘Return’ con el scope anterior.
20.- Revisa que el tipo de la expresión de como resultado bool.
21.- Genera un cuádruplo GotoF.
22.- Añade a la pila de saltos el cont de cuádruplos.
23.- Genera cuádruplo Goto con return a lo último en la pila de saltos.
24.- Llena el GotoF con el cont actual de cuádruplos.
25.- General cuádruplo de Asignación con el peek de la pila de operadores, hacer pop a la pila de operadores.
26.- Checar que los parámetros mandados en una llamada a función tengan congruencia con los argumentos de esa función.
27. Validar que la función que se está llamando existe en el directorio de procedimientos.
28.- Generar el cuádruplo Era.
29. Generar el cuádruplo Gosub.
30. Generar el cuádruplo Ver (arreglo).
31. Generar el cuádruplo de expresiones.
32.- Crear fondo falso.
33. - Eliminar fondo falso.

Tabla de consideraciones semánticas

Int = 1, bool = 2, double = 3

	+	-	*	/	%	=	==	!=	>	<	>=	<=	&&	||
int vs int	1	1	1	1	1	1	2	2	2	2	2	2		
int vs double	3	3	3	3	3	3	2	2	2	2	2	2		
int vs bool														
float vs float	3	3	3	3	3	3	2	2	2	2	2	2		
float vs bool														
bool vs bool							2	2					2	2

Descripción del proceso de Administración de Memoria en Compilación

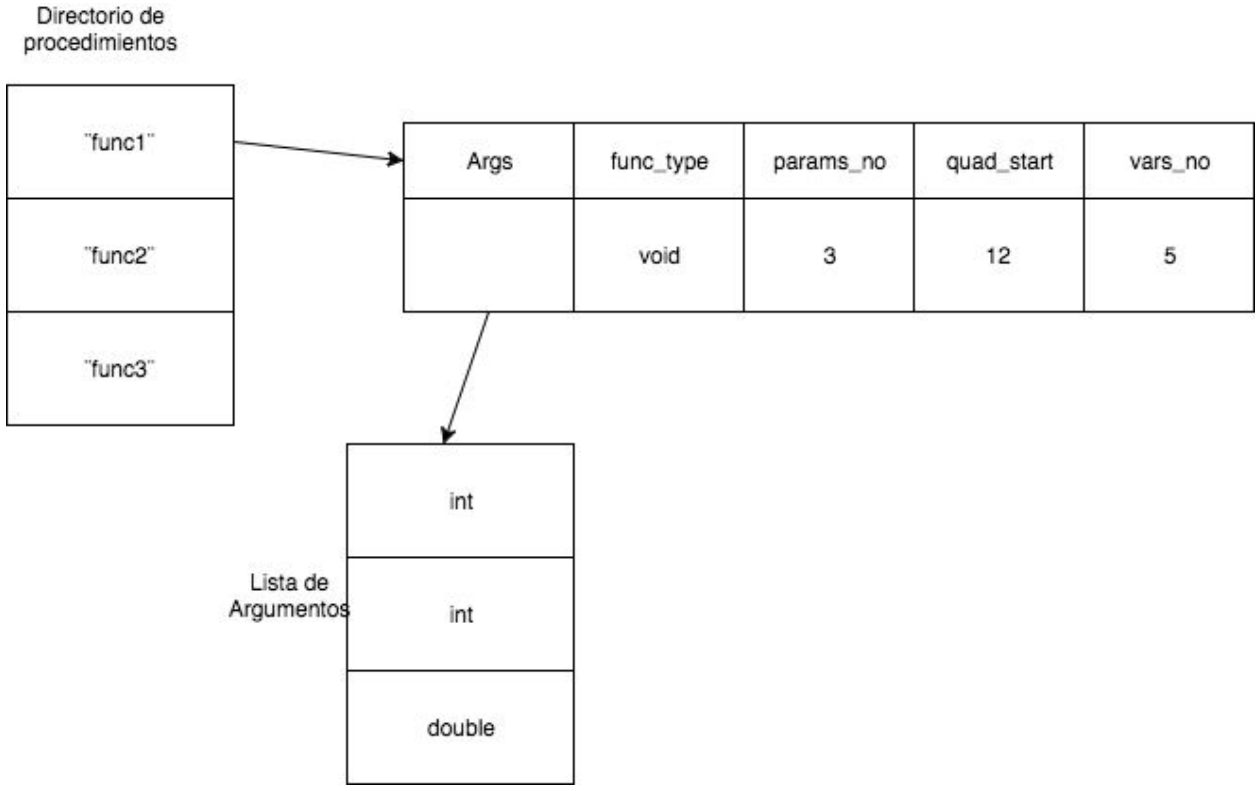
Descripción del proceso de Administración de Memoria en Compilación

En compilación se van registrando las variables, constantes y temporales dentro de la tabla de variables en su scope correspondiente, con sus respectivas direcciones de memoria ya asignadas dependiendo de su tipo y su scope.

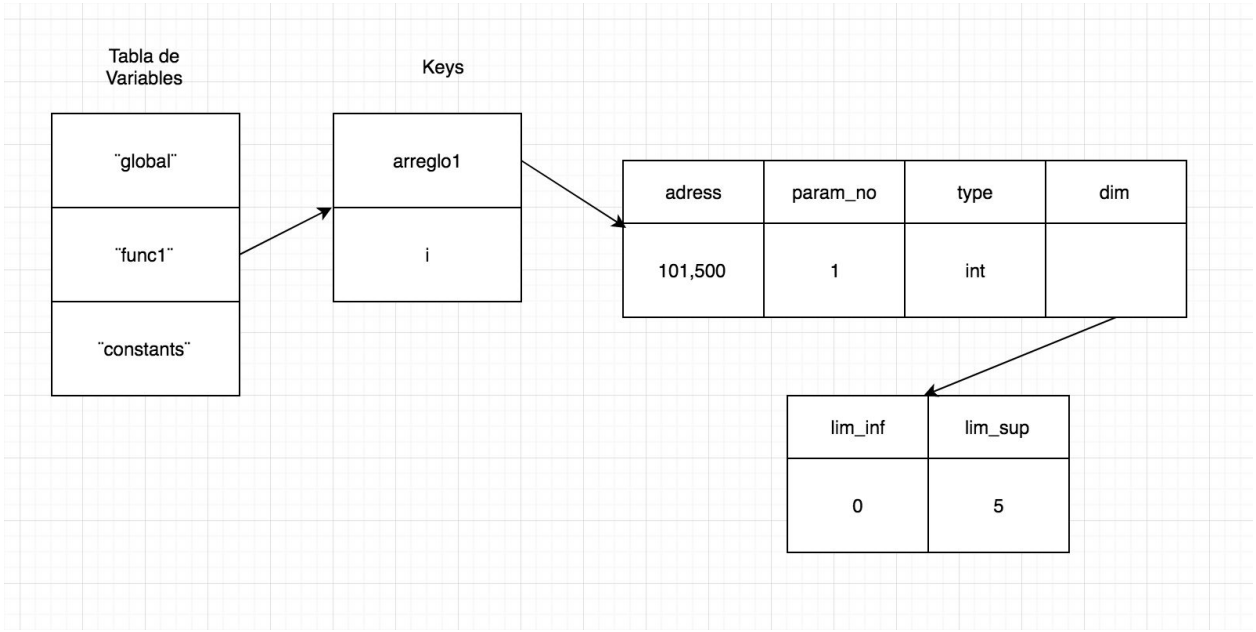
Posteriormente a esto son agregadas a memoria en su espacio correspondiente, para poder ser utilizadas en ejecución y no tener una referencia a vacío.

Especificación gráfica de las estructuras de datos utilizadas

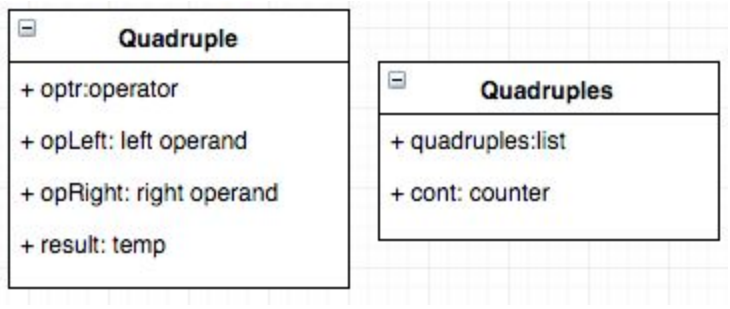
Directorio de Procedimientos



En la figura anterior se describe de manera gráfica el directorio de procedimientos, este tipo de estructura es un directorio en el cual su index es el id de la función. Y cada elemento del directorio, en este caso función tiene una lista de argumentos, el tipo de función, número de parámetros, el quad en donde inicia el procedimiento, y el número de variables en el procedimiento.



La figura anterior es la representación gráfica de la tabla de variables. Esta es un directorio con diferentes scopes, ya sea global, constantes o funciones que fueron declaradas. En cada scope se definen llaves que contienen otro diccionario que contiene la dirección de la variable, el número de parámetro que es solo si esta dentro del scope de una función, el tipo de variable, y la dim, que contiene otro directorio con el lim_inf y lim_sup en caso de ser arreglo.



Se utilizó también la clase Quadruple, la cual representa una cada cuádruplo que se genere, estos después son guardados en la clase Quadruples, que es una lista en donde se agregan los cuádruplos generados y además tiene funciones específicas, como fillQuad para llenar los cuádruplos de manera más sencilla.

Descripción de la Máquina Virtual

Lenguaje, Equipo y Utilidades utilizados en el Desarrollo

Mismos que en el compilador.

Descripción del proceso de Administración de Memoria en Ejecución

Descripción del proceso de Administración de Memoria en Ejecución

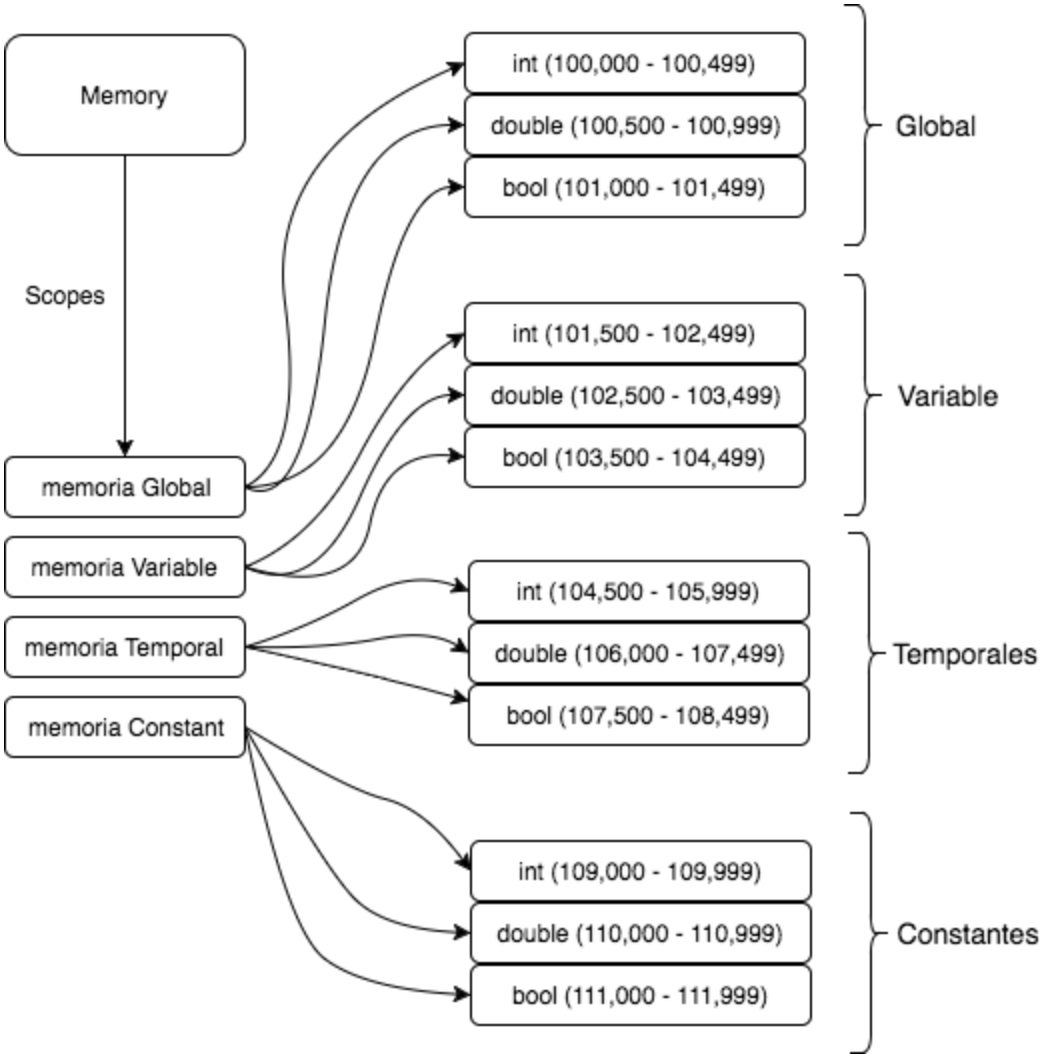
En ejecución se manejan 4 bloques de memoria: Global, Variable, Temporal y Constantes.

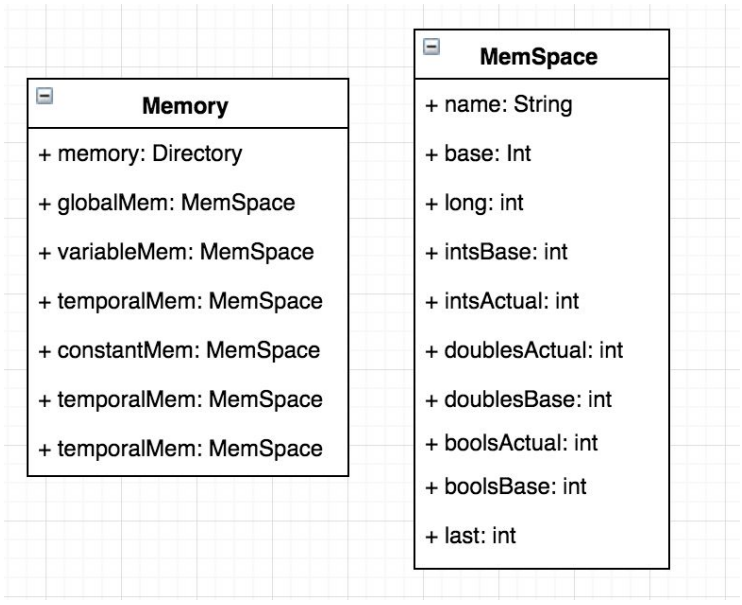
Las globales nunca se resetean y pueden ser accesadas desde cualquier parte de la ejecución, sea la memoria que sea la actual.

En el caso de las memorias Variable y Temporal se trata de un stack de memorias, esto para poder manejar llamadas a funciones y recursión, manteniendo así la memoria anterior activa para poder volver a ser utilizada al acabar la función actual

La memoria constante son números o valores específicos que se encuentran dentro del código y que se registraron para poder ser utilizadas dentro de todo el codigo y no repetir constantes, se registran con una dirección específica de constantes y se les asigna el valor que se encuentra en el código, esas direcciones nunca son modificadas y pueden ser accesadas desde cualquier parte de la ejecución.

Gráfica de Estructuras de Datos Utilizados


Memory
Scopes
memoria Global
memoria Variable
memoria Temporal
memoria Constant
int (100,000 - 100,499)
double (100,500 - 100,999)
bool (101,000 - 101,499)
int (101,500 - 102,499)
double (102,500 - 103,499)
bool (103,500 - 104,499)
int (104,500 - 105,999)
double (106,000 - 107,499)
bool (107,500 - 108,499)
int (109,000 - 109,999)
double (110,000 - 110,999)
bool (111,000 - 111,999)
Global
Variable
Temporales
Constantes


Memory
+ memory: Directory
+ globalMem: MemSpace
+ variableMem: MemSpace
+ temporalMem: MemSpace
+ constantMem: MemSpace
+ temporalMem: MemSpace
+ temporalMem: MemSpace
MemSpace
+ name: String
+ base: Int
+ long: int
+ intsBase: int
+ intsActual: int
+ doublesActual: int
+ doublesBase: int
+ boolsActual: int
+ boolsBase: int
+ last: int

La figura anterior son las clases que se utilizaron en la memoria, **Memory** es la clase principal de la memoria, este es un directorio con 4 diferentes keys que representan los scopes, que son: Global, Variable, Temporal y Constantes. Esta clase Memory declara un **MemSpace** para cada scope. **MemSpace** es una clase hecha específicamente para manejar las direcciones de memoria en cada scope que se usa en Memory.

Asociación entre Direcciones Virtuales y Reales

Al momento de registrar una variable o una función se le asigna una variable en memoria que es guardada en la tabla de variables y se añade a la memoria en su scope correspondiente con valor nulo que será asignado en ejecución, como se mostró anteriormente, los espacios en memoria son 4: globales, constantes, variables y temporales, estas son las responsables de generar los espacios disponibles en memoria.
Al terminar el scope actual en la generación de cuadruplos, las memorias de variables y temporales son reiniciadas, para dar espacio suficiente a las demás funciones de poder tener su espacio totalmente disponible, al finalizar la generación de cuadruplos, se leen las dir

Descripción de Funciones

Descripción de funciones importantes

Función	Descripción
REGLAS	
def p_program(p)	Regla inicial - Genera Goto main
def p_vars(p)	Declaración de variables, utiliza var_type para identificar el tipo de variable y more_vars para declarar más variables en una sola línea.
def p_arr(p)	Declaración de arreglos y llama a register_space que registra el espacio en memoria del arreglo.
def p_func(p)	Declaración de funciones, utiliza func_type para identificar el tipo de la función, registra la función en el directorio de procedimientos y llama a func_block que genera los cuádruplos de cada statement.
def p_arguments(p)	Definición de los argumentos de una función.
def p_statement(p)	Identifica los tipos de statements que se declaran y dependiendo a esto, se generan los cuádruplos de acorde al tipo de statement.

def p_write(p)	Define un statement write y manda llamar a gen_write_quad
def p_read(p)	Define un statement read y manda llamar a gen_read_quad
def p_return_stmt(p)	Define un statement de return y manda llamar a gen_return_quad
def p_cycle(p)	Define un ciclo while, checa que la expresión del while sea compatible usando check_type
def p_condition(p)	Define una condición if y checa que la expresión del if sea bool utilizando check_type
def p_assignation(p)	Define la asignación de un valor o función a una variable.
def p_func_call(p)	Define la llamada a una función, llama a is_valid_func para validar que la función existe, y genera el cuádruplo correspondiente.
def p_params(p)	Define los parámetros de una función
def p_expression(p)	Define la expresión y manda a llamar gen_exp_quad mandando como parámetro el tipo de expresión que se va a crear.
def p_conc(p)	Define una expresión de concatenación And o Or
def p_exp(p)	Define una expresión de comparación >, <, etc
def p_term(p)	Define una expresión + y -
def p_factor(p)	Define una expresión *, /, mod
def p_to_pilaOp(p)	Registra las variables
def p_ao(p)	
def p_comp(p)	
def p_pl(p)	
def p_dm(p)	
SEMANTIC AND SYNTAX VALIDATION FUNCTIONS	
def check_arr_param(p)	Checa que las declaraciones de arreglos sean solamente con ints

def to_var_table(p)	Guarda el nombre de las variables en la tabla de variables.
def to_proc_dir(p)	Registra la función en el directorio de procedimientos y las variables locales de la función las guarda en el scope de la función.
def to_args(varid, vartype, line, p)	**Parametros:** varid, vartype, line, p Genera un blueprint de los argumentos de un función y registra los parámetros como variables dentro de la función
def actual_quad_no(scope)	Obtiene el cuádruplo de la función para utilizarlo después en un gosub
def is_valid_func(p)	Revisa que la función que se está llamando es válida y exista.
def getConstType(p)	Obtiene el tipo de constante que es, que puede ser int, float o bool.
def exitsInVarTable(var)	Revisar que existe la variable en la tabla de variables.
def tryRegisterVar(var)	Devuelve la dirección virtual y el tipo de variable, sirve para asegurarse que las constantes se registren como constantes y asegurar que la variable está registrada.
def register_space(p)	Genera el espacio en memoria de la dimensión total del arreglo y separa el espacio en memoria del total del arreglo y registra en la tabla de variables el total del espacio.
QUAD GENERATING FUNCTIONS	
def gen_goto_main()	Genera el cuádruplo Goto y lo añade a la lista de cuádruplos.
def fill_main_quad()	Llena el cuádruplo main con el salto, obteniendo el salto de la pila de saltos.
def push_false_bottom()	Añade la pila de operadores un fondo falso
def to_pilaOp(var, val, line, p)	Recibe la variable que contiene la dirección de la variable y el tipo de la variable. Pushea a la pila de operandos la dirección, y pushea el tipo. Añade a memoria la dirección junto con su valor.

def check_type(p)	Checa que el resultado condición sea siempre bool, y si no te da un error type mismatch
def gen_est_quad(line, qtype)	Switch que recibe como parametro el tipo de statement, y a partir de ahi llama a la función de generación de cuadruplo es.
def gen_declaration_assign_quad(line)	Genera el cuádruplo de declaración de variables
def gen_read_quad(line)	General el cuádruplo de un statement read()
def gen_write_quad(line)	Genera el cuádruplo de un statement write()
def var_assign(p)	Genera el cuádruplo de asignación de valor a variables después de haber declarado una variable.
def gen_assignation_quad(line)	Genera el cuádruplo de asignación cuando se está declarando.
def gen_gotof_quad(res)	Genera un cuádruplo GotoF
def gen_goto()	Genera un cuádruplo Goto
def fill_end_condition()	Cuando se termin
def cycle_start()	Cuando inicia un ciclo se pushea la pila de salto el num de cuádruplo.
def cycle_end()	Cuando termina el while, guarda el end, return, y genera un cuádruplo Goto usando el return.
def gen_exp_quad(line, qtype)	**Parametros**: Se pasa la linea por si hay error. Tipo de cuádruplo que se va a generar, hay 4 tipos de expresiones, Factor: Multiplicación y División, Term: Suma y Resta, Comp: Comparación y Conc: And y Or
def check_args(p)	Checa los argumentos que se usaron cuando se llama a una función, concuerden con los parámetros de esa función.
def gen_go_sub()	Genera el cuádruplo Gosub cuando se llama a una función
def gen_era(p)	Genera el cuádruplo ERA
def gen_return_quad(scope, p)	Genera el cuádruplo de return

def gen_endproc_quad(p)	Genera el cuádruplo Endproc
def gen_end_quad()	General el cuádruplo END

Pruebas del Funcionamiento del Lenguaje

Pruebas de Funcionamiento

Fibonacci Recursivo

Factorial Iterativo

Llenado de arreglo iterativo