

**Instituto Tecnológico y de Estudios Superiores de Monterrey**

Documentación Final: Me&Myself

Materia:

Diseño de compiladores

Profesores:

Dr. Héctor Ceballos Ing. Elda G. Quiroga

Alumno:

Alejandro Alfredo Salgado Gaspar A00513221

Grupo: 1

### Noviembre 23, 2020. Monterrey, Nuevo León

Índice

1. Descripción del Proyecto………………………………………………………………………………….3

1.1 Propósito…………………………………………………………………………………………………..3

1.2 Objetivos……………………………………………………………………………………………………3

1.3 Alcance del proyecto………………………………………………………………………………….3

1.4 Requerimientos Funcionales………………………………………………………………………4

1.5 Requerimientos No Funcionales…………………………………………………………………5

1.6 Casos de uso generales………………………………………………………………………………5

1.7 Casos de prueba……………………………………………………………………………………….10

1.8 Proceso general seguido…………………………………………………………………………..22

1.9 Bitácora de seguimiento…………………………………………………………………………..23

1.10 Reflexión………………………………………………………………………………………………..23

1. Descripción del Lenguaje ……………………………………………………………………………….24

2.1 Nombre del lenguaje ……………………………………………………………………………….24

2.2 Descripción genérica ……………………………………………………………………………….24

2.3 Lista de errores ……………………………………………………………………………………….25

1. Descripción del Compilador…………………………………………………………………………..26

3.1 Equipo de Cómputo, Lenguaje y Librerías………………………………………………..26

3.2 Análisis de Léxico……………………………………………………………………………………..26

3.3 Análisis de Sintaxis……………………………………………………………………………………28

3.4 Generación de Código Intermedio y Análisis Semántico…………………………..32

3.5 Administración de la memoria en compilación…………………………………………41

1. Descripción de la Máquina Virtual………………………………………………………………….44

4.1 Administración de la memoria en ejecución……………………………………………..44

1. Pruebas del Funcionamiento del Lenguaje …………………………………………………….47

# **1. Descripción del Proyecto**

**1.1 Propósito**

El propósito de este proyecto fue realizar una actividad integradora que requiriera el uso aglomerado de los conocimientos técnicos en programación que se han adquirido a lo largo de la carrera, poniéndolos así en práctica en un proyecto de alta dificultad cuya realización certifica nuestros conocimientos y habilidades en el área de desarrollo de software, un proyecto así como lo es el desarrollo de un compilador; diseñado acorde al paradigma de nuestra preferencia (imperativo, orientado a objetos, funcional, etc.)

**1.2 Objetivos**

Este proyecto tuvo por objetivo el desarrollo del compilador de un lenguaje prediseñado por el profesor (o propuesto por nosotros), con todo lo que ello implica: desarrollo de analizador de léxico, desarrollo del analizador sintáctico, revisión semántica, compilación (cuadruplificación) de la entrada de texto, administración de segmentos de memoria y una maquina virtual (para la ejecución del programa o entrada de texto acorde a la cuadruplificación obtenida); un compilador de un lenguaje que incorporara como mínimo ciertos elementos, tanto de control de flujo de ejecución como de estructuras de datos. Entre los elementos mínimos requeridos por el lenguaje, se encuentran: declaración de variables locales y globales, declaración de funciones con y sin valor de retorno, evaluación de expresiones (aritméticas, lógicas y relacionales), elementos de control de flujo ( if-else, while, for), estructuras de datos (arreglos unidimensionales, y opcionalmente, arreglos n-dimensionales), llamadas a funciones, tipos de datos almacenables (char,int,float), entre otros.

**1.3 Alcance del proyecto**

Entre las funcionalidades que se acordaron el desarrollo de nuestro lenguaje de programación debía cumplir como mínimo, se encuentran:

* Declaración de variables locales (sin anidar) y globales
* Tipos de datos almacenables: char, int y float
* Declaración de funciones con 0 a N parámetros, con y sin retorno
* Declaración de arreglos unidimensionales locales y globales
* Evaluación de operaciones aritméticas, relacionales y lógicas
* Llamadas a funciones
* Estatutos de repetición: for y while
* Estatutos de control de flujo de ejecución: if-else
* Estatutos de asignación
* Funciones especiales “built-in” para el despliegue de gráficos por medio de un GUI externo
* Funciones “built-in” para la lectura (read) y escritura (write) de datos
* Error-checking para el despliegue de mensajes de error tanto de tipo sintáctico como de tipo semántico

**1.4 Requerimientos Funcionales**

* RF001: Analizar que los símbolos de la entrada de texto sean tokenizables (lexer)
* RF002: Analizar que la cadena de tokens esté en el orden descrito por las reglas sintácticas (parser)
* RF003: Validar semántica de tipos de datos (asignaciones, tipo de valore de retorno, tipo de parámetro, operaciones binarias), cantidad de parámetros en llamadas a funciones, declaración previa de variables y funciones, asignación previa a variables, etc.
* RF004: Compilar de la entrada de texto en una lista de cuádruplos
* RF005: Ejecutar un programa después de compilarlo
* RF006: Administrar el uso de memoria en compilación (asignación de direcciones de memoria)
* RF007: Administrar el uso de memoria en ejecución (acceso, modificación, y destrucción de información guardada en memoria)
* RF008: Desplegar errores semánticos y sintácticos
* RF009: Permitir ejecutar una entrada de texto a través del compilador mediante un comando en consola
* RF010: Permitir la declaración (int, char o float) de variables y arreglos (unidimensionales) locales a una función
* RF011: Permitir la declaración (int, char o float) de variables y arreglos (unidimensionales) globales a todo el programa
* RF012: Permitir la declaración de funciones con 0 a N parámetros, con o sin valor de retorno
* RF013: Permitir llamadas a funciones con o sin retorno
* RF014: Realizar la evaluación de expresiones simples (operadores aritméticos, relacionales y lógicos) y complejas (arreglos y llamadas a funciones dentro de las expresiones)
* RF015: Permitir realizar estatutos de repetición (for y while)
* RF016: Permitir realizar estatutos de condición (if-else)
* RF017: Permitir realizar estatutos de asignación
* RF018: Permitir realizar llamadas a funciones “built-in” (especiales) para el despliegue de gráficos en un GUI externo
* RF019: Permitir realizar llamadas a funciones “built-in” que permitan la lectura o escritura (en consola) de datos

**1.5 Requerimientos No Funcionales**

* RNF001: Que la llamada a ejecución del compilador mediante consola (uso de comandos) sea amigable
* RNF002:Que el lenguaje diseñado, indistintamente del paradigma elegido, posea elementos con los cuales usuario pueda familiarizarse como si estuviera usando otro lenguaje perteneciente al mismo paradigma (imperativo, en este caso), haciéndolo así más fácil de utilizar y reconocer
* RN003: Que el diseño del compilador y la maquina virtual hagan uso de una administración eficiente en el uso de su memoria virtual (por ejemplo, el limpiamiento de las direcciones temporales que ya no se utilizan cada que se termina de ejecutar una función) y la comunicación entre las estructuras de datos elegidas para el desarrollo de ambos elementos (estructuras eficientes, mantenibles y escalables)

**1.6 Casos de uso generales**

|  |
| --- |
| **Caso de prueba**: Obtener Sumar |
| **ID**: CG001 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma iterativa (mediante un while) que solicita al usuario un numero N. Mas tarde, el programa solicita al usuario N cantidad de números. Finalmente, la suma de todos los números provistos es desplegada en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_obtenerSuma.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var float : i, p, k, o,suma;  var int : datos, j ;  main ( )  {  write("cuantos datos deseas sumar ");  read (datos);  suma = 0;  j = 0;  while (j < datos ) do {  write ("Escribe un numero ") ;  read(p);  suma = suma + p;  j = j + 1;  }  write("La sumatoria es ");  write(suma);}   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la termina, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N en la terminal 4. Escribir el valor de N en la terminal 5. Se solicitaran al usuario N números más 6. Escribir en consola N números más |
| **Postcondiciones:**   1. Se desplegará en consola la sumatoria de los N numeros |

|  |
| --- |
| **Caso de prueba**: Obtener Promedio |
| **ID**: CG002 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma iterativa (mediante un for) que solicita al usuario un numero N. Mas tarde, el programa solicita al usuario N cantidad de números. Finalmente, el promedio de la suma de todos los números provistos es desplegada en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_obtenerPromedio.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var float : i, j, p, k, o,suma, resultado;  var int : datos ;  main ( )  {  write("cuantos datos deseas promediar ");  read (datos);  suma = 0;  for j = 1 to j > datos do {  write ("Escribe un numero ") ;  read(p);  suma = suma + p;  j = j + 1;  }  resultado = suma / datos;  write("El promedio es ");  write(resultado);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la termina, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N en la terminal 4. Escribir el valor de N en la terminal 5. Se solicitaran al usuario N números más 6. Escribir en consola N números más |
| **Postcondiciones:**   1. Se desplegará en consola el promedio de la sumatoria de los N numeros |

|  |
| --- |
| **Caso de prueba**: Desplegar gráficos |
| **ID**: CG003 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa cuya finalidad demostrar su funcionalidad gráfica, para ello desplegando una serie de formas gráficas diferentes, como círculos, líneas, puntos y arcos, en un GUI externo |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_graficos.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : i, j[10], p;  var float : valor, yepei ;  main ( )  {  penup();  size(15);  point(30.5,30.5);  pendown();  color("red");  circle(40.5);  size(5);  line(5,12,78,180);  clear();  line(5,12,78,180);  clear();  penup();  point(0,0);  pendown();  arc(50,170);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la termina, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] |
| **Postcondiciones:**   1. Se desplegará en consola todos los gráficos que se solicitan en el código |

**1.7 Casos de prueba (“test cases”)**

|  |
| --- |
| **Caso de prueba**: Factorial iterativo |
| **ID**: CP001 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma iterativa que solicita al usuario un numero N. El programa calcula el factorial de N, y este es desplegado en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_factorial\_cic.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program Factorial;  var int: i,num,result;  main(){  write("Ingresa el numero a evaluar ");  read(num);  result = num;  for i = num - 1 to i < 1 do {    result = result \* i;  i = i - 1;  }  write ("resultado ");  write(result);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la termina, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N en la terminal 4. Escribir el valor de N en la terminal |
| **Postcondiciones:**   1. Se desplegará en consola el factorial de N |

|  |
| --- |
| **Caso de prueba**: Factorial recursivo |
| **ID**: CP002 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma recursiva que solicita al usuario un numero N. El programa calcula el factorial de N, y este es desplegado en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_factorial\_rec.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program Factorial;  var int: num,result;  int module fact(int x){  if ((x==0)||(x==1)) then{  return (1);  }  else {  return(x \* fact(x -1));  }  }  main(){  write("Ingresa el numero a evaluar con factorial");  read(num);  result = fact(num);  write(fact(num));  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N en la terminal 4. Escribir el valor de N en la terminal |
| **Postcondiciones:**   1. Se desplegará en consola el factorial de N |

|  |
| --- |
| **Caso de prueba**: Fibonacci iterativo |
| **ID**: CP003 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma iterativa que solicita al usuario un numero N. El programa calcula el N-simo número de la serie fibonacci, y este es desplegado en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_fibonacci\_cic.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : numero, contador, x,y,z;  main(){  x = 0;  y = 1;  z = 0;  contador = 0;  write("Escriba el numero para evaluar la serie fibonacci ");  read(numero);  while (contador < numero ) do {  z = x + y;  x = y;  y = z;  contador = contador + 1;  }  write ("resultado ");  write(z);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N 4. Escribir el valor de N |
| **Postcondiciones:**   1. Se desplegará en consola el N-simo número de la serie fibonacci |

|  |
| --- |
| **Caso de prueba**: Fibonacci recursivo |
| **ID**: CP004 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa desarrollado de forma recursiva que solicita al usuario un numero N. El programa calcula el N-simo número de la serie fibonacci, y este es desplegado en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_fibonacci\_rec.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : numero, resultado;  int module fibonacci(int n){  if((n == 0) || (n==1)) then {  return(1);  }  else{  return(fibonacci(n-1) + fibonacci(n-2));  }  }  main(){  write ("Escribe el numero para evaluar la serie fibonacci ");  read(numero);  resultado = fibonacci(numero) ;  write(resultado);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitara el valor de N en la terminal 4. Escribir el valor de N en la terminal |
| **Postcondiciones:**   1. Se desplegará en consola el N-simo número de la serie fibonacci |

|  |
| --- |
| **Caso de prueba**: Find en un arreglo |
| **ID**: CP005 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa iterativo el cual inserta un dato en la casilla de un arreglo, dejando el resto de las casillas vacías. Después, el programa busca de forma cíclica la casilla en la cual se insertó el dato en el arreglo, y una vez encontrada la casilla, se guarda (no el dato, sino la casilla). Finalmente, al salir del ciclo, la casilla en donde se encontraba el numero insertado es desplegada en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_arreglos\_find.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : numero, index, sum, valores[10];  main(){  valores[7] = 5;  index = 0-1;    for numero = 0 to numero > 9 do {  if (valores[numero] != null) then {  if (valores[numero] == 5) then {  index = numero;  }  }  numero = numero + 1;  }  write(index);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] |
| **Postcondiciones:**   1. Se desplegará en consola el index o casilla en donde se encuentra el numero insertado |

|  |
| --- |
| **Caso de prueba**: Sort en un arreglo |
| **ID**: CP006 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa iterativo en el cual se llena un arreglo con números cualesquiera y en desorden, y mediante un ciclo, el contenido del arreglo se ordena de menor a mayor, desplegando el contenido final y ordenado del arreglo en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_arreglos\_sort.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : i, j, index, sum, valores[7], temp;  main(){  valores[0] = 5;  valores[1] = 12;  valores[2] = 1;  valores[3] = 7;  valores[4] = 145;  valores[5] = 57;  valores[6] = 78;  index = 0-1;  for i = 0 to i > 6 do {  for j = 0 to j > 6 do {  if (valores[i] < valores[j]) then {  temp = valores[i];  valores[i] = valores[j];  valores[j] = temp ;  }  j = j + 1 ;  }  i = i + 1 ;  }  write(i);  write(j);  write(valores[0]);  write(valores[1]);  write(valores[2]);  write(valores[3]);  write(valores[4]);  write(valores[5]);  write(valores[6]);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] |
| **Postcondiciones:**   1. Se desplegará en consola el contenido del arreglo ordenado de menor a mayor |

|  |
| --- |
| **Caso de prueba**: Operaciones con arreglos |
| **ID**: CP007 |
| **Actores**: Usuario, Sistema |
| **Descripción**: Se trata de un programa iterativo en el cual se llenan dos arreglos con números cualesquiera. Después, el usuario proporciona dos números, los cuales hacen referencia a casillas en el arreglo correspondiente (el primer numero proporcionado hará referencia a la casilla del arreglo uno, y el segundo para la casilla del arreglo dos). Proporcionados los números, se procede a realizar la suma entre ambos arreglos usando el contenido de los arreglos en las casillas proporcionadas por el usuario. El resultado es desplegado en consola |
| **Precondiciones**:   1. Contar con todos los archivos que permiten ejecutar el compilador:    1. EmptyTempMemoryInstantiator.py    2. FunctionDirectory.py    3. Lexer.py    4. Parser.py    5. Operand.py    6. MeMyselfCompiler.py    7. Cuadruples.py    8. SemanticCube.py    9. Tools.py    10. VirtualAdresses.py    11. VirtualMachine.py 2. Contar con el archivo de prueba: prueba\_operaciones\_arreglos.txt |
| **Flujo de eventos:**   1. Abrir la terminal 2. Ubicarse en terminal en el directorio donde se encuentran todos los archivos necesarios 3. Si no se cuenta con el archivo de prueba, crear uno y escribir lo siguiente:   program MeMyself;  var int : A[10], B[20], i,j,casilla1,casilla2,numero1,numero2,suma, temp, contador;  main(){  for i = 0 to i > 9 do {  A[i] = i;  i = i + 1 ;  }  for j = 0 to i > 19 do {  B[j] = j;  j = j + 1 ;  }  write( "Escribe una casilla del 1 al 10" );  read (casilla1);  numero1 = A[casilla1-1];  write( "numero obtenido ");  write (numero1);  write( "Escribe una casilla del 1 al 20" );  read (casilla2);  numero2 = B[casilla2-1];  write( "numero obtenido ");  write (numero2);  suma = numero1 + numero2;  write("suma es igual a ");  write (suma);  }   1. Guardar el archivo en el mismo directorio donde están el resto de los archivos necesarios. 2. En la terminal, escribir el comando (sin los corchetes al escribir el nombre del archivo de prueba) : python3 memyselfcompiler.py [nombre con el que se guardo el archivo de prueba] 3. Se solicitará un numero en terminal 4. Escribir el numero en terminal 5. Se solicitará otro numero en terminal 6. Escribir el otro numero en terminal |
| **Postcondiciones:**   1. Se desplegará en consola el resultado de la suma del contenido de las casillas de ambos arreglos |

**1.8 Proceso general seguido**

La metodología que se siguió para el desarrollo del proyecto pudiera considerarse (en lo personal) del tipo scrum, en donde se tenían fechas de entrega en las cuales se debía entregar una versión incremental del proyecto (es decir, la versión del proyecto de la fecha de entrega anterior más las nuevas funcionalidades esperadas para la nueva fecha de entrega). La manera en la que se realizó esto (personalmente), fue utilizando github, en donde se tenia una branch funcional, y branches de trabajo. La branch funcional, o master, estaba exenta de hacérsele pushes (modificaciones directamente), y en esta branch siempre se encontraba la versión funcional del proyecto más recientes; dicho esto, la única manera de modificar la branch principal o master, era aceptando el merge request de alguna branch de trabajo (la cual se presume la funcionalidad correspondiente a esa branch de trabajo está terminad, libre de errores, y lista para fusionarse con la branch principal). En cambio, en las branches de trabajo, es ahí donde subía (push) pequeños avances cada vez que lograba algún pequeño cambio que funcionaba correctamente; dicho esto, la metodología para hacer push en las branches de trabajo fue: commits pequeños, y funcionales. En otras palabras, en las branches de trabajo, me abstuve de hacer commits que incluyeran una gran cantidad de cambios, o commits que en general no funcionaran (commits que hacían que el código no compilara).

**1.9 Bitácora de seguimiento:**

|  |  |
| --- | --- |
| **Fecha** | **Avance** |
| **Octubre (5-9) de 2020** | * Analizador léxico * Analizador sintáctico |
| **Octubre (12-16) de 2020** | * Directorio de funciones y tabla de variables * Cubo semántico |
| **Octubre (19-23) de 2020** | * Código para la generación de cuádruplos de expresiones aritméticas y estatutos secuenciales * Código para la generación de cuádruplos de estatutos condicionales: if-else |
| **Octubre (26-30) de 2020** | * Código para la generación de cuádruplos de estatutos no secuenciales: ciclos * Código para la generación de cuádruplos de funciones |
| **Noviembre (2-6) de 2020** | * Stack de memorias virtuales para la máquina virtual * Máquina virtual: Ejecución de expresiones aritméticas y estatutos secuenciales |
| **Noviembre (9-13) de 2020** | * Código para la generación de cuádruplos de arreglos. * Máquina virtual: Ejecución de estatutos condicionales y estatutos no secuenciales |
| **Noviembre (16-20) de 2020** | * Primera versión de la documentación * Código para la generación de cuádruplos de funciones especiales * Máquina virtual: Indexamiento de arreglos en ejecución, ejecución de llamadas a funciones y ejecución a llamadas especiales |
| **Noviembre (23-27) de 2020** | * Entrega final del proyecto, 25 de noviembre a las 11:59 pm |

**1.10 Reflexión**

Este proyecto, más allá de su relevancia en cuanto a mi desarrollo profesional (pues no me dedicaré a desarrollar compiladores), sí me sirve para rectificar y certificar mi grado académico, y constatar por mi cuenta que me corresponde ser acreditador a un titulo de mi carrera, pues he podido realizar el proyecto más difícil de todos nuestros cursos a nivel técnico. Una certificación de confianza en uno mismo es lo que representa este proyecto, al darse cuenta uno mismo del nivel en el que se encuentra; bajo o alto comparado con los demás alumnos del tecnológico, pero suficiente para desarrollar este proyecto que exige un alto nivel técnico: un compilador (claro, no sin ayuda de pre-diseños, pautas y consejos de los profesores, quienes básicamente ya nos ofrecen la solución del diseño del compilador, más nos encargan a nosotros traducirla a código).

# **2. Descripción del Lenguaje**

**2.1 Nombre del lenguaje**

El nombre de mi lenguaje es Me&Myself, que alude a que el único desarrollador del compilador es el autor de este documento, esto debido a que seleccionar esta modalidad con una propuesta prediseñada permitía el ahorro de tiempo en fabricar tanto en diagramas como en documentación una nueva propuesta de desarrollo para el compilador; así utilizando esa semana de trabajo en empezar temprano el proyecto (no habiendo tenido que trabajar en una propuesta nueva por querer un proyecto de mi decisión). Además, elegí esta modalidad individual porque en mi experiencia, mis trabajos en equipo siempre resultan en disputas, en donde yo hago todo y termino peleándome con el equipo por su falta de colaboración y desdén al esperar que yo haga todo como si fuera mi obligación pasarles la materia, y quienes deseosamente se roban mi trabajo y me califican mal en las coevaluaciones (reduciendo mi calificación y ellos llevándose el crédito por mi trabajo) por referirme a ellos de forma despectiva en su cara (y así provocar una situación donde es la palabra de ellos contra la mía), o sino el caso contrario; en donde ellos o alguno del equipo hace todo el trabajo, pero a diferencia del caso anterior, yo me comporto de forma agradecida, reconociendo verbalmente a esa persona que si no fuera el o ella, el proyecto no hubiera salido adelante, además de calificándome con menor puntaje en la evaluación que susodicho compañero, y tratando de ayudar en lo posible aunque de forma mínima en comparación. Por ello y más, decidí hacer este proyecto solo.

**2.2 Descripción genérica**

Me&Myself es un lenguaje que incorpora las características básicas de un lenguaje perteneciente al paradigma imperativo procedural, el cual implica un flujo de ejecución secuencial de forma general (las instrucciones se ejecutan en el orden en que se encuentran en el archivo de entrada); características básicas entre las cuales se encuentran elementos como: declaración de variables globales y locales, declaración de funciones con y sin valor de retorno, una función principal (donde inicia la ejecución), estructuras de datos básicas (arreglos), evaluación de expresiones aritméticas simples y complejas, estatutos de asignación, estatutos condicionales, estatutos no secuenciales (ciclos), entre otros. Sin embargo, este lenguaje (Me&Myself), dado su simplicidad y lo básico que es, cuenta con no varias sino muchas restricciones comparadas con la capacidad de un lenguaje imperativo popular, como lo es C; restricciones entre las cuales se encuentran:

1. Solo se pueden declarar variables tipo char, int y float
2. Se cuenta con un limitado tamaño de segmentos de memoria temporal, local y global
3. No se puede realizar declaraciones de variables anidadas (es decir, declarar variables en cualquier parte del código, por ejemplo: afuera y dentro de un if, y adentro del if que se encuentra dentro del if anterior, y así sucesivamente)
4. No es posible declarar variables locales en la función principal (main)
5. No es posible declarar un arreglo en la firma de una función, ni enviar un arreglo como parámetro de una función

**2.3 Lista de errores**

1. Llamada a una función cuando el tipo de algún parámetro no concuerda con el tipo esperado de ese parámetro
2. Llamada a una función cuando se proporcionan más parámetros de los debidos
3. Asignación de valores que no sean char, int o float
4. Operaciones binarias incompatibles (por ejemplo, bool + char)
5. Errores de sintaxis o léxico detectados por PLY
6. Una expresión aritméticas como resultado dentro de la condición de un estatuto (if, for, while) en lugar de una expresión lógica o relacional
7. Asignación de una expresión cuyo resultado final no sea del mismo tipo que la variable a la cual se le esta asignando
8. Llamada a una función inexistente
9. Uso de una variable inexistente
10. Acceso a una casilla del arreglo no valida (inferior al limite inferior o superior al limite superior)
11. Incorporación de una llamada a una función void dentro de una expresión
12. Llamada a una función no void como estatuto, en lugar de dentro de una expresión
13. Un estatuto return dentro de una función void
14. Falta de un estatuto return dentro de una función no void
15. Asignación de una expresión que resulta en un valor diferente de char, int o float (bool, por ejemplo) a una variable
16. Declaración de variables o funciones con el mismo nombre que palabras reservadas
17. Declaración de variables previamente declaradas
18. En llamadas anidadas (una función llama a otra, y así sucesivamente), llamadas a funciones que aun no han sido declaradas
19. Tratar de acceder a casillas de un arreglo en una variable que no es un arreglo

# **3. Descripción del Compilador**

**3.1 Equipo de Cómputo, Lenguaje y Librerías**

El equipo de cómputo utilizado para la elaboración del lenguaje Me&Myself fue una Macbook Pro 2010, con sistema operativo El Capitan, cuyas especificaciones técnicas son las siguientes:

* Procesador Core 2 Duo de Intel a 2,4 o 2,66 GHz con 3 MB de caché de nivel 2 integrada y compartida
* Bus frontal a 1.066 MHz
* Memoria DDR3 a 1.066 MHz de 4 GB (dos módulos SO-DIMM de 2 GB); dos ranuras SO-DIMM que admiten hasta 8 GB
* Disco duro Serial ATA de 250 o 320 GB a 5.400 rpm; disco duro opcional de 320 o 500 GB a 5.400 rpm o memoria sólida de 128, 256 o 512 GB
* Procesador gráfico GeForce 320M de NVIDIA con 256 MB de SDRAM DDR3 compartida con la memoria principal5

El lenguaje de programación utilizado para el desarrollo del compilador de Me&Myself fue Python 3.4. Además, se utilizaron varias librerías para llevar acabo diferentes funciones del compilador, entre las cuales se encuentran:

1. PLY: esta librería es una incomporación de Lex y Yacc a Python (Python-Lex-Yacc), donde lex es la librería que permite la generación automática de un analizador de léxico, y yacc es la librería que permite la generación automática de un analizador de sintaxis
2. Sys: esta librería ya se encuentra “built-in” en python, y fue utilizada para poder hacer read de datos escritos en la consola, por ejemplo: en el comando python3 parser.py myprogram.txt, mediante la librería sys se puede recuperar el dato “myprogram.txt” de la consola, para que el archivo parser.py sepa cual archivo parsear
3. PythonTurtle: esta librería se utiliza para poder desplegar en un GUI los gráficos solicitados por los requisitos del lenguaje Me&Myself, gráficos tales como: líneas, círculos, puntos, etc.

**3.2 Análisis de Léxico**

Para la tokenización de símbolos de una entrada de texto, se categorizaron los tokens es en tres tipos diferentes: palabras reservadas, tokens regulares, y literales. A continuación se describe cada uno de ellos

1. **Literales**: son aquellos tokens que no requieren una expresión regular, pues Lex traduce el lexema (texto crudo) directamente a un token. Por ejemplo: envés de generar un token llamado LPAREN cuya expresión regular sea r”\(“, se puede declarar al símbolo “(“ como un literal, el cual generará el token “(“ a partir del mismo lexema, de manera que el lexema y el token valdrán lo mismo; sin embargo, la única restricción es que el lexema sea no más de un carácter, por lo cual las literales son útiles para declarar tokens de operaciones. Dentro las literales que se utilizaron, se encuentran las siguientes:

* Lexemas: ‘+’ , ‘\*’, ‘/’ ‘{‘, ‘}’, ‘(‘, ‘)’, ‘>’, ‘<’, ‘=’, ‘[‘, ‘]’, ‘:’, ‘;’, ‘,’

1. **Palabras reservadas**: son aquellos tokens cuya expresión regular es la misma que la de un identificador, y por lo mismo, pueden ser detectadas previo y durante la identificación de un token de tipo ID. Esto puede llevarse a cabo mediante la expresión regular que genera tokens de tipo ID, revisando si el lexema se encuentra dentro de las palabras reservadas; si es así, esto significa que el token ya no se trata de un ID, sino de una palabra reservada, y por lo cual se debe guardar el token como tal (llamar al token igual que al lexema leído), y no como un token de tipo ID. Entre las palabras reservadas que se utilizaron, se encuentran las siguientes:

* Reservadas: 'program', 'main', 'module', ‘void', 'null', 'var', 'int', 'float', 'char', 'if', 'then', 'else', 'return', 'do', 'while', for', 'to', read', 'write', 'clear', 'point', 'circle', 'penup', 'pendown', 'color', 'size', 'global', "line", "arc"

1. **Tokens regulares**: estos son los tokens que requieren de una expresión regular para poder identificarlos, y su nombramiento esta a disposición del programador. Entre los tokens regulares que se utilizaron, están:

Lexemas de operaciones que no pudieron tokenizarse como literales

* minus = r'\-'
* equals = r'\=\='
* not\_equals = r'\!\='
* and = r'\&\&'
* or = r'\|\|'

Lexemas de tipos de datos e identificadores que requieren de una expresión regular

* cte\_c = r'(\‘[a-zA-z\_0-9]\’) | (\'[a-zA-z\_0-9]\')'
* cte\_s = r' (\“[a-zA-z\_0-9\s\,\=]\*\”) | (\"[a-zA-z\_0-9\s\,\=]\*\")'
* cte\_i = r'\d+'
* cte\_f = r'\d+\.\d+'
* id = r'[a-zA-Z0-9]+'

**3.3 Análisis de Sintaxis**

Las reglas gramaticales utilizadas para el análisis sintáctico de una entrada de texto tokenizada, fueron las siguientes (sin considerar los puntos neurálgicos):

1. PROGRAMA: program id ; PROGRAMA\_OPTS PRINCIPAL

| program id ; PRINCIPAL

1. PROGRAMA\_OPTS: DEC\_V FUNCS

| DEC\_V

| FUNCS

1. PRINCIPAL: main ( ) BLOQUE
2. DEC\_V : DEC\_V var TIPO\_SIMPLE : LISTA\_VAR ;

| var TIPO\_SIMPLE : LISTA\_VAR ;

1. TIPO\_SIMPLE: int | float | char
2. LISTA\_VAR: LISTA\_VAR , VARIABLE\_FIX

| VARIABLE\_FIX

1. FUNCS: FUNCS FUNC\_TYPES module id ( ) BLOQUE

| FUNCS FUNC\_TYPES module id ( PARAMS )

BLOQUE

| FUNCS FUNC\_TYPES module id ( PARAMS ) DEC\_V

BLOQUE

| FUNCS FUNC\_TYPES module id ( ) DEC\_V

BLOQUE

| FUNC\_TYPES module id ( ) BLOQUE

| FUNC\_TYPES module id ( PARAMS ) BLOQUE

| FUNC\_TYPES module id ( PARAMS ) DEC\_V BLOQUE

| FUNC\_TYPES module id ( ) DEC\_V BLOQUE

1. FUNC\_TYPES: TIPO\_SIMPLE | void
2. PARAMS: PARAMS , TIPO\_SIMPLE id | TIPO\_SIMPLE id
3. VARIABLE\_FIX: id [ cte\_i ] | id
4. VARIABLE: id [ EXPRESION ] | id
5. LOOP\_ESTATUTO: LOOP\_ESTATUTO ESTATUTO | ESTATUTO
6. BLOQUE : { LOOP\_ESTATUTO return EXPRESION ; }

| { LOOP\_ESTATUTO }

| { return EXPRESION ; }

| { }

1. EXPRESION: EXPRESION and EXP\_R

| EXPRESION or EXP\_R

| EXP\_R

1. EXP\_R: EXP\_A > EXP\_A

| EXP\_A < EXP\_A

| EXP\_A equals EXP\_A

| EXP\_A not\_equals EXP\_A

| EXP\_A

1. EXP\_A: EXP\_A + TERMINO

| EXP\_A minus TERMINO

| TERMINO

1. TERMINO: TERMINO \* FACTOR

| TERMINO / FACTOR

| FACTOR

1. ESTATUTO: ASIGNACION ;

| CONDICION

| WHILE

| FOR

| / LLAMADA ;

| LECTURA ;

| ESCRITURA ;

| LLAMADA\_BI;

1. FACTOR: LLAMADA

| CTE

| VARIABLE

| ( EXPRESION )

1. CTE: cte\_i | cte\_f | cte\_c
2. LLAMADA: id ( LLAMADA\_OPTS ) | id ( )
3. LLAMADA\_OPTS: LLAMADA\_OPTS , EXPRESION

| EXPRESION

1. ASIGNACION: VARIABLE = EXPRESION
2. CONDICION: if ( EXPRESION ) then BLOQUE

| if ( EXPRESION ) then BLOQUE else BLOQUE

1. WHILE: while ( EXPRESION ) do BLOQUE
2. FOR: for ASIGNACION to EXPRESION do BLOQUE
3. LECTURA: read ( LECTURA\_OPTS )
4. LECTURA\_OPTS: LECTURA\_OPTS , id | id
5. ESCRITURA: write ( ESCRITURA\_OPTS )
6. ESCRITURA\_OPTS: ESCRITURA\_OPTS , cte\_s

| ESCRITURA\_OPTS , EXPRESION

| cte\_s

| EXPRESION

1. LLAMADA\_BI: POINT | CIRCLE | PENUP

| PENDOWN | COLOR | SIZE | CLEAR

1. CLEAR: clear ( )
2. POINT: point ( EXPRESION , EXPRESION )
3. CIRCLE: circle ( EXPRESION )
4. PENUP: penup ( )
5. PENDOWN: pendown ( )
6. COLOR: color ( cte\_s )
7. SIZE: size ( EXPRESION )
8. LINE : line '(' EXPRESION ',' EXPRESION ',' EXPRESION ',' EXPRESION ')'
9. ARC : arc '(' EXPRESION ',' EXPRESION ')'

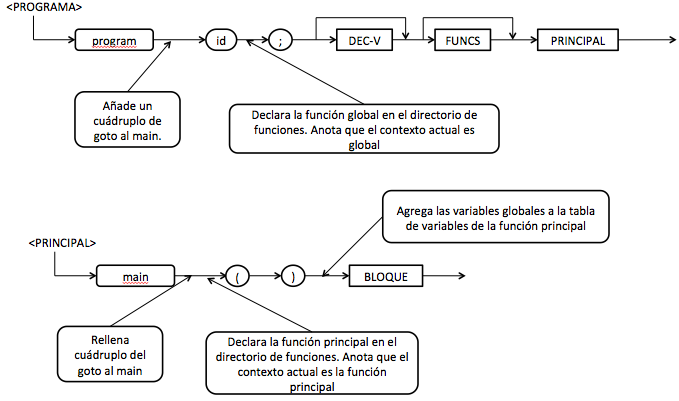
**3.4 Generación de Código Intermedio y Análisis Semántico**

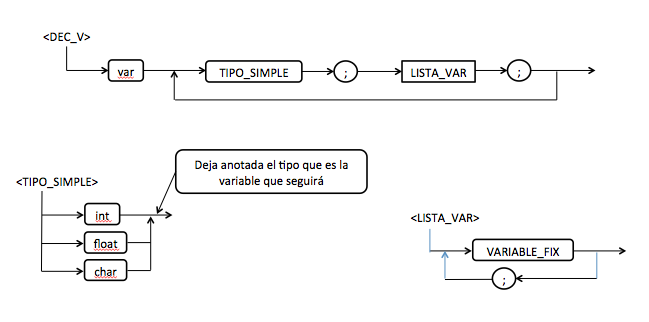
Código de Operación y Direcciones Virtuales

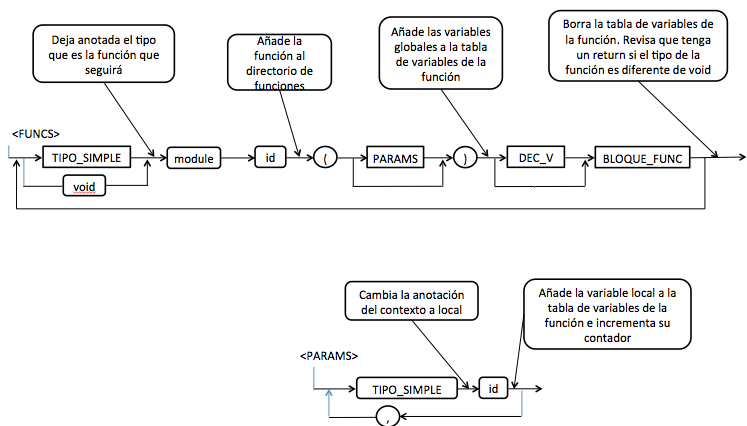
En general, se utilizaron diferentes segmentos de memoria (cada uno con el propósito de alojar un tipo de dato diferente) agrupados en una sola estructura, denominada memoria virtual. A continuación se enlista cuáles fueron los limites en las direcciones virtuales de cada uno de los segmentos de memoria de la estructura de memoria virtual:

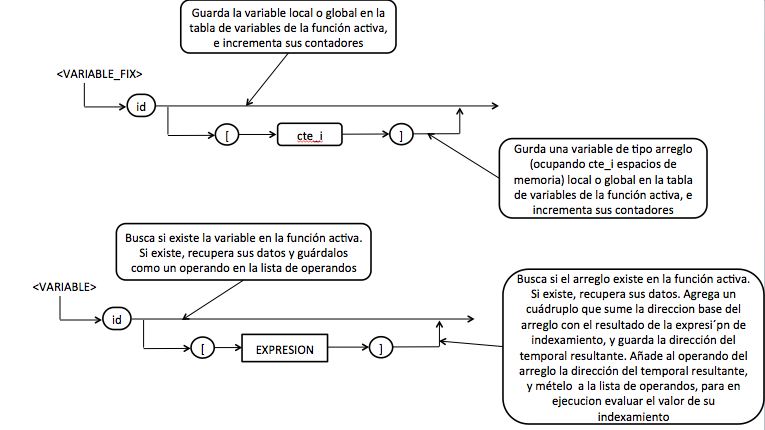
|  |  |  |  |
| --- | --- | --- | --- |
| Segmento de memoria | Tipo | Direcciones Virtuales | |
| Global | Int | 15000– 15099 | |
| Float | 15100 – 15199 | |
| Char | 15200 – 15299 | |
| Local | Int | 15300 – 15399 | |
| Float | 15400 – 15499 | |
| Char | 15500 - 15599 | |
| Temporal | Int | 15600 – 15699 | |
| Float | 15700 – 15799 | |
| Char | 15800 – 15899 | |
| Bool | 15900 – 15998 | |
| Constante | Int | 16000 – 16099 | |
| Float | 16100 – 16199 | |
| Char | 16200 – 16299 | |
| String | 16300 - 16399 |

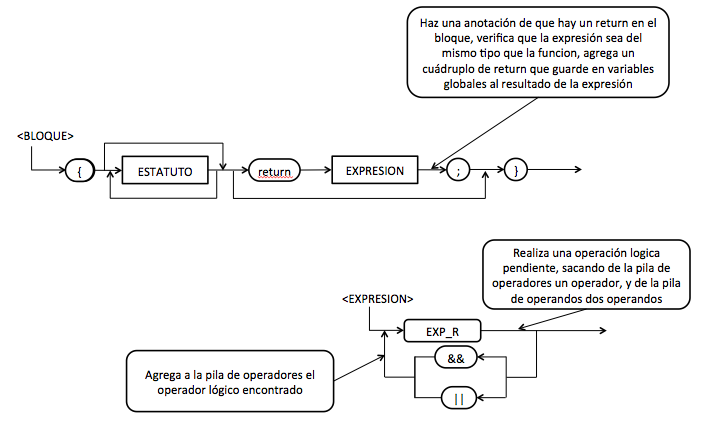
Diagramas de Sintaxis con Acciones

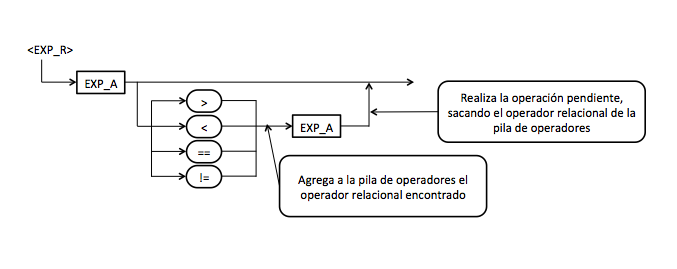


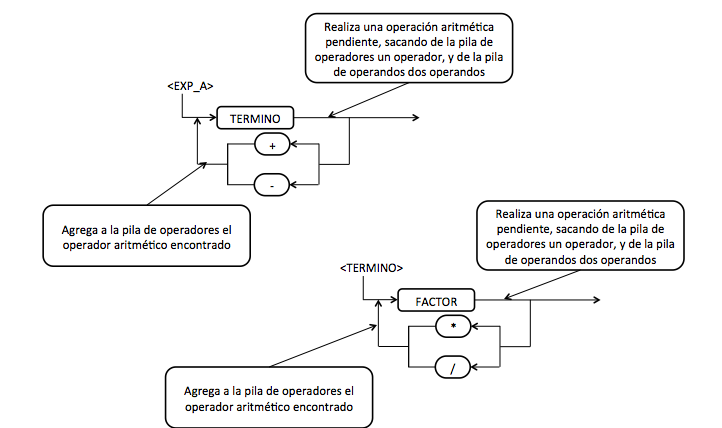


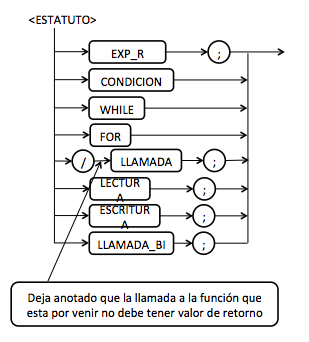


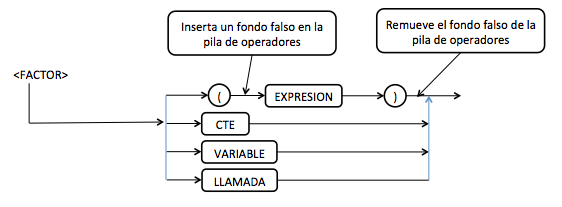


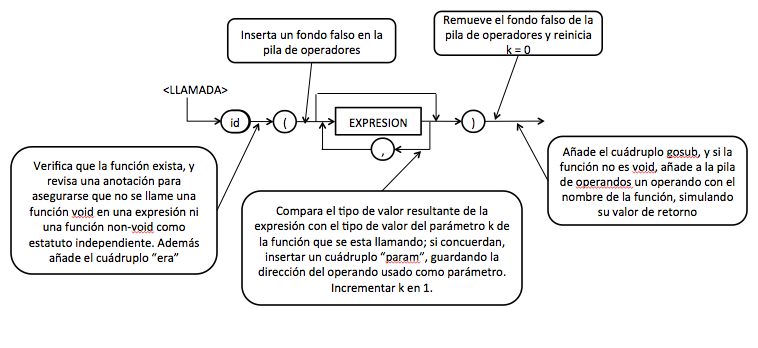


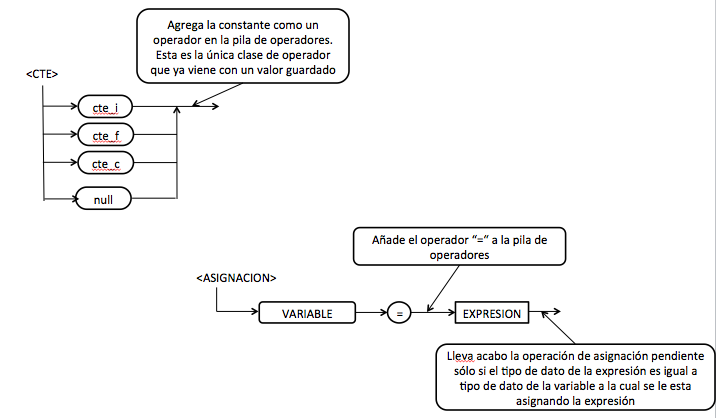


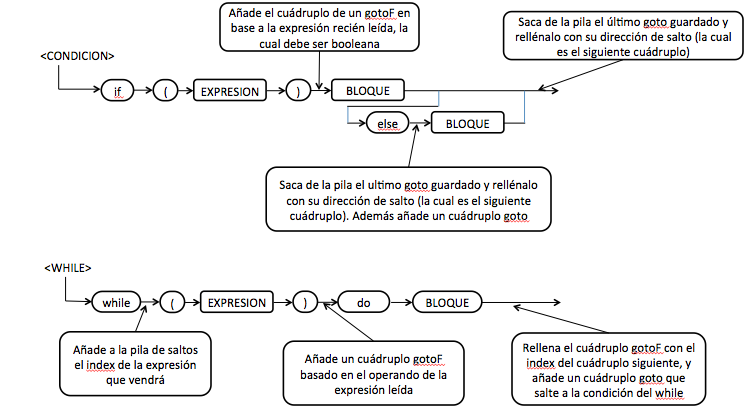


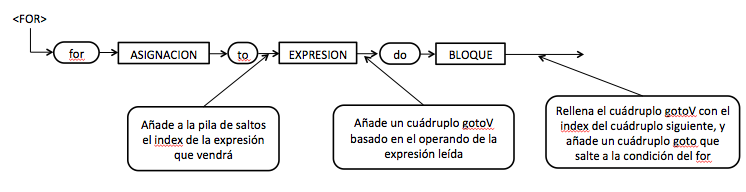


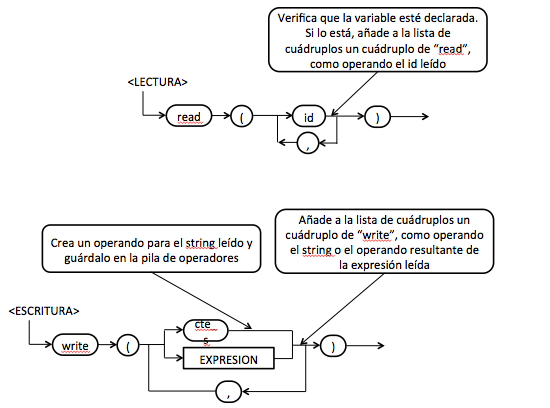


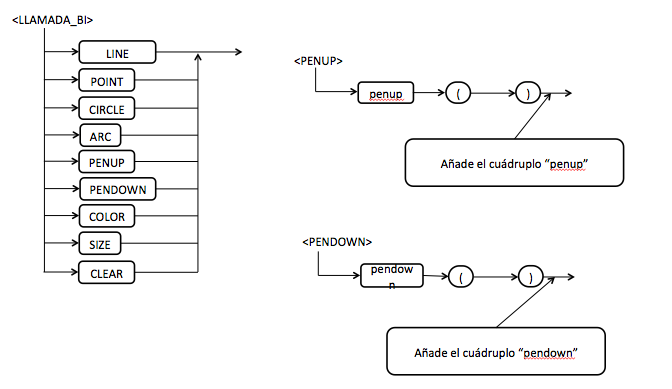


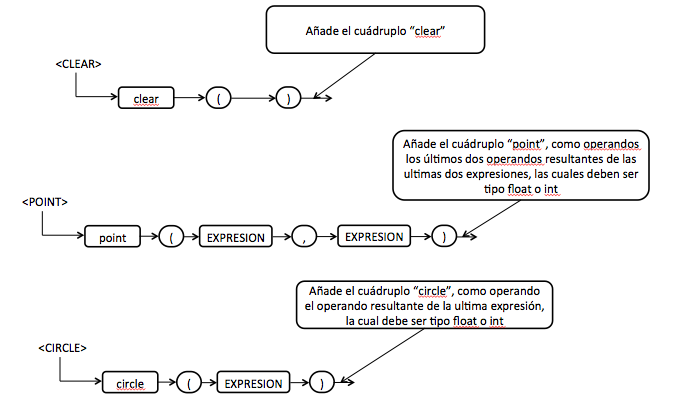


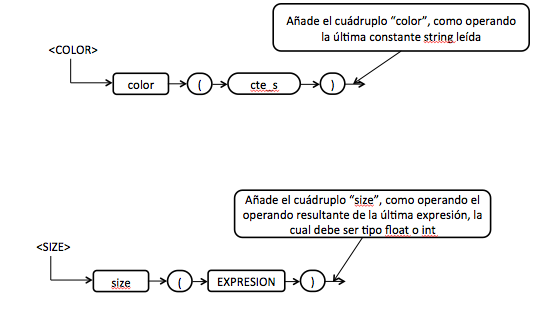












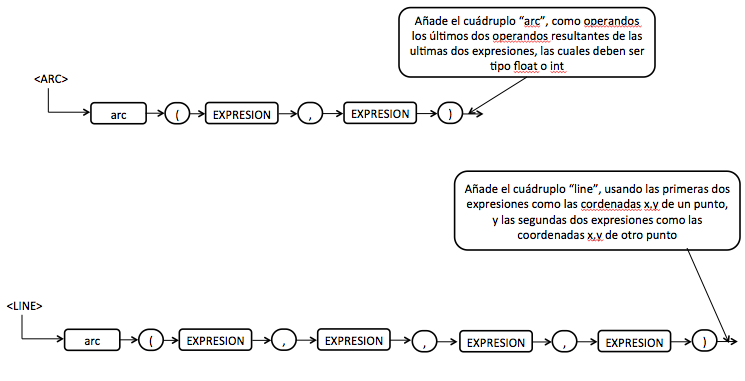
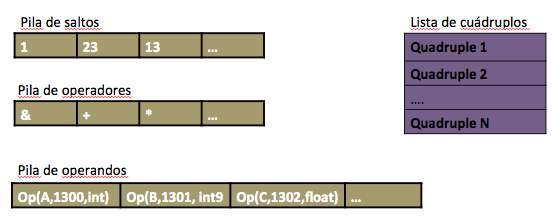
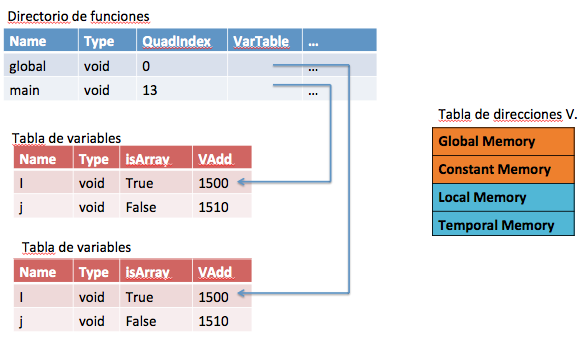


Tabla de consideraciones semánticas

La siguiente tabla describe el resultado esperado de realizar operaciones binarias (aquellas que requieren el uso de dos operandos) con diferentes tipos de operandos, evaluando así el resultado posible de cada una de las combinaciones posibles de operandos y operadores.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OperadorL | OperadorR | + | - | \* | / | = | ¡= | == | < | > | | | | && |
| Int | Int | Int | Int | Int | Int | Int | Bool | Bool | Bool | Bool | Error | Error |
| float | Float | Float | Float | Float | Error | Bool | Bool | Bool | Bool | Error | Error |
| char | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| bool | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| void | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| float | Int | Float | Float | Float | Float | Float | Bool | Bool | Bool | Bool | Error | Error |
| float | Float | Float | Float | Float | Float | Bool | Bool | Bool | Bool | Error | Error |
| char | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| bool | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| void | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| char | Int | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| float | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| char | Error | Error | Error | Error | Char | Bool | Bool | Error | Error | Error | Error |
| bool | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| void | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| bool | Int | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| float | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| char | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| bool | Error | Error | Error | Error | Bool | Bool | Bool | Error | Error | Bool | Bool |
| void | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| void | Int | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| float | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| char | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| bool | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| void | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error | Error |
| Null | Int | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| float | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| char | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| Bool | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| void | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |
| null | Error | Error | Error | Error | Error | Bool | Bool | Error | Error | Error | Error |

**3.5 Administración de la memoria en compilación**



# El diagrama anterior muestra cada una de las estructuras utilizadas para administrar el uso de memoria durante compilación. A continuación se enlista cada una de ellas, junto con su debido funcionamiento:

# Pila de saltos: Esta estructura es un stack que almacena números int, los cuales siempre representan una cosa: un índice de salto (el índice del cuádruplo al cual algún otro cuádruplo en uso debe saltar). Los estatutos que generan stackeamiento de saltos son: los estatutos condiciones (if-else, while, for) por su naturaleza no secuencial, lo que hace que requieran cuádruplos de saltos para solucionarlos (goto,gotoF,gotoV).

# Pila de operadores: Esta estructura es un stack que almacena valores char predefinidos (solamente aquellos operadores ya registrados en el lenguaje del compilador, es decir, tokenizables en el lexer), y que representa el orden en que las operaciones stackeadas debe realizarse. La manera en la que estos operadores son extraídos no respeta el orden de llegada en la entrada de texto, sino que el diseño anidado del compilador (su sintaxis) el orden se convierte en una jerarquía de operadores donde las operaciones aritméticas son mayor prioridad que las relacionales, y las relacionales que las lógicas. Las acciones que generan stackeamiento de operadores son siempre encontrarse con algún operador registrado.

# Pila de operandos: Este es un stack que contiene todos los operandos que se van encontrando a lo largo de la entrada de texto, un stackeamiento el cual sí respeta el orden de llegada de los operadores, y cuya presencia en el stack representa alguna variable, casilla de un arreglo o constante pendiente de realizar alguna operación con él (usualmente binaria, requiriendo así la extracción de dos operandos a la vez). Es de mencionar que en este stack se guardan objetos, los cuales contienen toda la información relevante de una variable o constante hasta el momento, como lo es: su nombre y su valor (si es que tienen alguno de estos), su tipo de dato y su memoria asignada. Las acciones que generan stackeamiento de operadores es siempre el encuentro con alguna variable o constante, no en su declaración, sino dentro de una expresión.

# Lista de cuádruplos: Esta es una lista de tuplas (cada una de cuatro elementos, también llamados cuadruplificación) que contiene una traducción del contenido de una entrada de texto útil para la parte de ejecución, haciendo así posible que todas las operaciones descritas en una entrada de texto sean ejecutables por otro componente externo (maquina virtual) gracias al diseño de esta lista, la cual en su contenido dicta qué operación, con qué operandos, y en que direcciones debe realizarse o ejecutarse cada una de las sentencias dictadas en estos cuádruplos. La información que se encuentra en cada uno de los cuádruplo almacenados es la siguiente: primer elemento y el más relevante de todos, la acción a realizar; el resto de los tres elementos pudieran o no contener operandos (si es que la acción los requiere) los cuales contienen la información necesaria para llevar acabo correctamente la acción encontrada.

# Tabla de direcciones virtuales: Se trata de un diccionario, en donde cada elemento del diccionario contiene segmentos de memoria, siendo cada segmento de memoria un objeto de tipo VirtualAddress (el cual recibe como entradas sus direcciones virtuales inicial y final). Los tipos de segmento de memoria que incluye esta tabla de direcciones virtuales son: direcciones globales, direcciones de constantes, direcciones para locales y direcciones para temporales. Además, la utilidad de ser utilizar un objeto como segmento de memoria es que se le puede delegar el trabajo de administrar su propia memoria asignada (saber cual es la siguiente dirección de memoria disponible que se puede asignar a una nueva variable, además de encontrar, modificar y borrar los datos de una dirección virtual en específico). Por último, es de mencionar que el trabajo en compilación de la tabla de memorias virtuales no es el mismo que el de la memoria virtual en ejecución, siendo en ejecución su trabajo más importante dejar asignada, reservada, o anotada una dirección virtual fija (con excepción de los arreglos) a cada variable que se use en ejecución, lo cual servirá para que en ejecución la maquina virtual sepa ya sea de donde obtener los datos, o a donde guardarlos en la instancia de memoria temporal adecuada.

# Directorio de funciones: Se trata de un objeto, el cual aloja como principal atributo un diccionario indexado con el nombre de los métodos ya declarados (haciendo así imposible indexar un método cuyo nombre sea el mismo que uno ya guardado, esto debido a la naturaleza de la estructura de diccionario), en donde cada elemento indexado del diccionario contiene otro diccionario, esto con el propósito de almacenar varios valores por cada elemento que haya en el diccionario principal, valores que representan información relevante y útil de una función a la hora de guardarla, información tale como: tipo de retorno, tipo de datos de los parámetros que espera, contadores de cantidad de direcciones de memoria temporales que necesitará, ubicación de su index de inicio en a lista de cuádruplos, y por supuesto, una tabla de variables (objeto). Este objeto cuenta con una serie de métodos útiles para modificar y accesar (e incluso borrar) a sus elementos, además de hacer print en consola de su contenido

# Tabla de variables: Esta estructura es similar a la del directorio de funciones, en el sentido de que se trata también de un objeto cuyo principal atributo es un diccionario que indexa sus elementos con el nombre de las variables declaradas (haciendo así imposible declarar variables con el mismo nombre), en donde cada elemento indexado del diccionario contiene otro diccionario, usado solamente para almacenar todos los datos relevantes de una variable, tales como: tipo de dato, scope de la variable, banderas boleana para saber si se trata de un arreglo y/o un parámetro, entre otros datos, siendo el más importante su dirección de memoria virtual asignada. Además , este objeto cuenta con los métodos que se necesitan para modificar y accesar a los elementos de su diccionario o tabla de variables (para así poder recuperar, modificar o borrar los datos de una variable), incluyendo métodos de escritura en consola de los contenidos de su diccionario de variables.

# **Descripción de la máquina virtual**

# **4.1 Administración de la memoria en ejecución**

# 

# A continuación se detallan cada uno de los componentes involucrados en la comunicación con la maquina virtual para su funcionamiento correcto:

# Temporal memory stack: se trata de una pila que aloja estructuras del tipo “memoria virtual temporal”, las cuales incluyen direcciones temporales y direcciones loales. Dado que en compilación se hace que cada función inicie su asignación de direcciones desde el comienzo, esto significa que no existe una asignación secuencial de direcciones entre llamadas; por ejemplo: la primer variable del método fact empieza en la dirección 001, y la primera variable de la dirección fact2 empieza en 001, siendo el problema evidente que en caso de llamadas anidadas, las variables de la segunda llamada sobrescribirán las direcciones de la primera llamada. Es por eso que un stack de memoria temporal es necesario, donde se crea un bloque de memoria temporal cada vez que se mande a llama a una función, haciendo que los estatutos de la función anidada ya no sobrescriben los datos guardados por una función previa, pues estos se encontrarán en un bloque de memoria diferente dentro del stack (en el bloque anterior). Una vez terminada la ejecución de la memoria en torno, es posible borrar del stack el bloque de memoria temporal actual

# Global Memory: Se trata de un bloque de memoria donde se almacena memoria global, lo cual implica que no es necesario crear una instancia de este bloque por llamada a función, sino que los valores aquí almacenados están siempre disponibles en cualquier parte de la ejecución de la máquina virtual. Esto es útil también para el almacenamiento de los valores de retorno de una función, pues de guardarlo en el bloque de memoria temporal en turno, este dato se perdería al finalizar la función. Así mismo, esta estructura es útil para poder realizar llamadas recursivas (llamadas anidadas a funciones con el mismo nombre), pues esto consistiría en que la llamada de mayor profundidad actualice el valor global de retorno al terminar su ejecución, y así consecutivamente, hasta que no queden más llamadas pendientes, y el valor global de retorno de la función quede con un único valor. Nota: para llamadas N llamadas recursivas de retorno (ejemplo: return fact(2) + fact(3) ) fue necesario incorporar una lista de valores en la dirección global donde se guarda el valor de retorno de la función, esto porque en un estatuto con varias llamadas a la misma función, era la llamada final la que sobrescribía al último el valor de retorno, convirtiendo, por ejemplo, un “return fact(2) + fact(3)) “ en equivocadamente un “return fact(3) + fact(3)”, pues como cada llamada modifica el valor de retorno, era la ultima llamada la que tenia la batuta siempre de cuanto iba a valer el valor de retorno; por ello se incorporó una lista de valores en el caso de valores de retorno: para guardar el resultado temporal de cada llamada, haciendo que la ultima llamada a la función no gobierne el resultado final del velor de retorno.

# Constants Memory: Es en este bloque de memoria donde se guardan las variables constantes (con su valor incluido); así mismo, no es necesario borrar este bloque por cada instanciación de una función, sino que está siempre disponible para la memoria virtual en cualquier parte de su ejecución.

# Previous temp memory: Este es un bloque temporal de memoria, que es una copia exactamente igual al bloque de memoria temporal anterior al actual. El objetivo de guardar este bloque temporal anterior es para resolver un problema: los cambios de contexto al llamar a una función. Al llamar a una función, se crea un nuevo bloque de memoria temporal vacío, y en ese proceso, se inicializan sus parámetros; sin embargo, ¿que pasa si la expresión que inicializa sus parámetros incluye variables declaradas en la bloque de memoria temporal anterior? Habría un error de ejecución pues esas variables no se encontrarían disponibles en el nuevo contexto actual. Es por ellos que, para no perder esos valores, se guarda una copia del bloque de memoria temporal anterior; para buscar en ella valores, si es necesario, al realizar un cambio de contexto que en los parámetros de la función llamada requiera valores del contexto anterior.

# Quadruples list: Esta es la lista de cuádruplos que se genera en compilación, la cual representa una traducción de la entrada de texto en operaciones realizables, la cual a su vez marca cuales y en que orden deben ejecutarse estas operaciones , así como la dirección de memoria de todas las variables que se utilizan en el camino.

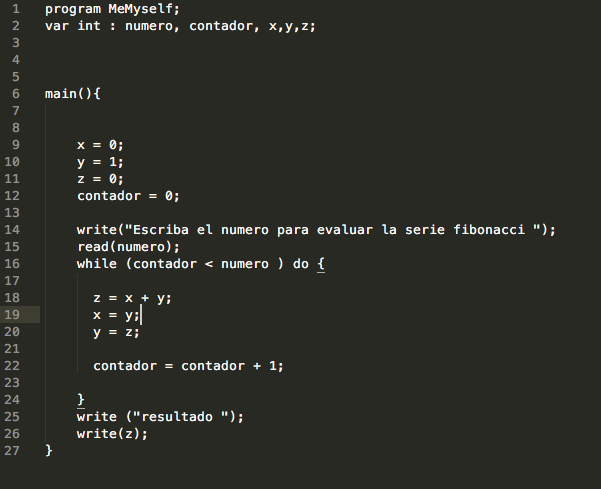
# Function Directory: Este componente, aunque no tan relevante para el funcionamiento crucial de la maquina virtual, si contiene información necesaria para la solución de algunos cuádruplos; tres para ser exactos: era, paremeter, y arrayindex. Para el cuádruplo era, este componente me permite recuperar el index del cuádruplo donde comienza la función que se está llamando; para el cuádruplo parameter, me permite recuperar del directorio la dirección verdadera del parámetro, en la cual hay que inicializar su valor dentro del nuevo contexto de memoria; y para arrayindex, me permite recuperar cuales es la dirección base, y el limite superior e inferior del arreglo (para realizar el chequeo semántico de “indexs out of bounds”, entre otras acciones más importantes)

# Virtual machine: Este es el corazón de este sistema diagramado, y es quien incorpora y utiliza el resto de los componentes mencionados para llevar a cabo correctamente la ejecución del programa; funciones entre las cuales se encuentran: administración de la memoria (creación, acceso, y destrucción de memoria temporal) real en ejecución, flujo secuencial y no secuencial de ejecución de los cuádruplos, revisiones semánticas adicionales (por ejemplo, que el indexamiento de un arreglo esté entre los limites del arreglo), etc. Es este componente el que sabe que hacer exactamente al encontrar un cuádruplo de cada tipo.

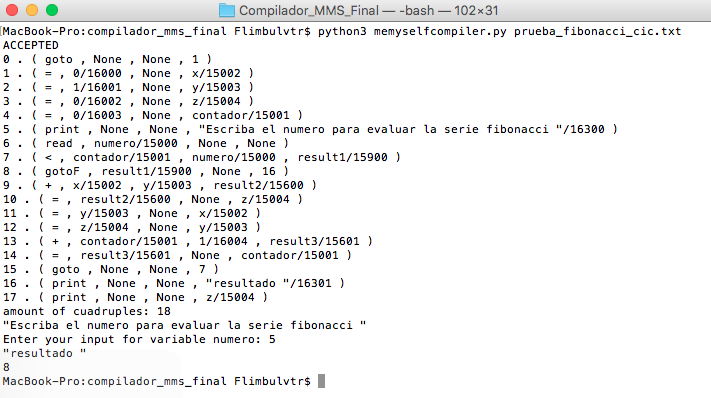
# **5. Pruebas de Funcionamiento**

# 1.-Fibonacci iterativo

# a. Entrada

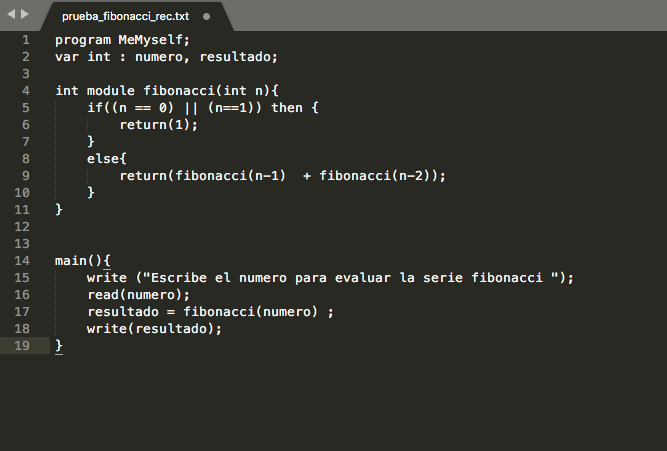
****

b. Salida

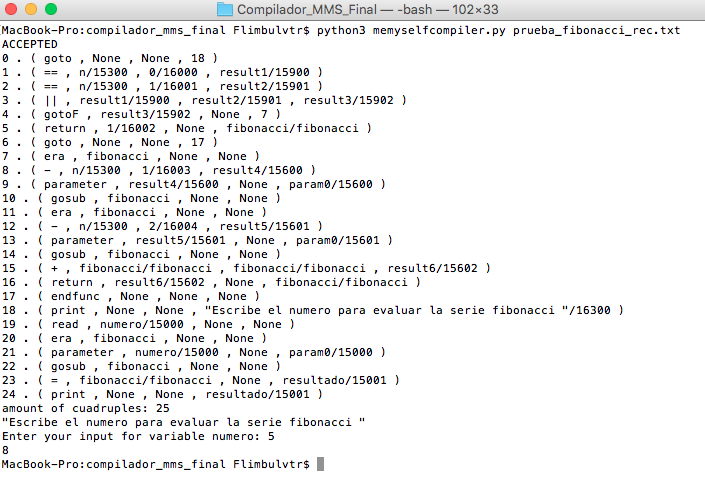


2.-Fibonacci recursivo

a.Entrada

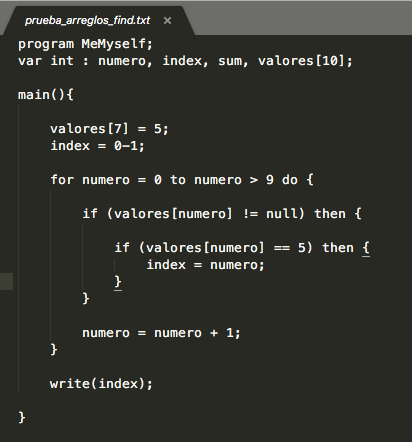


b.Salida

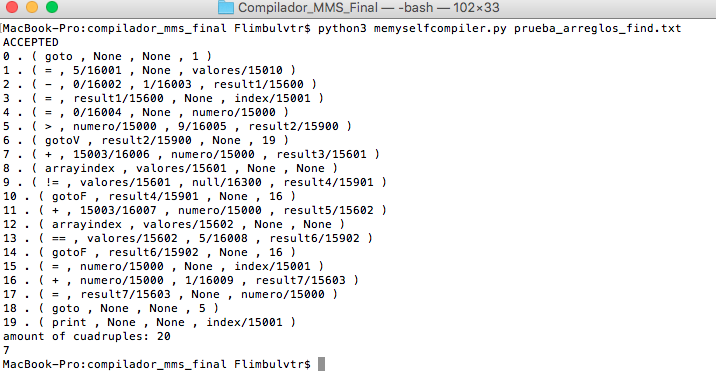


3.-Find en un arreglo

a.Entrada

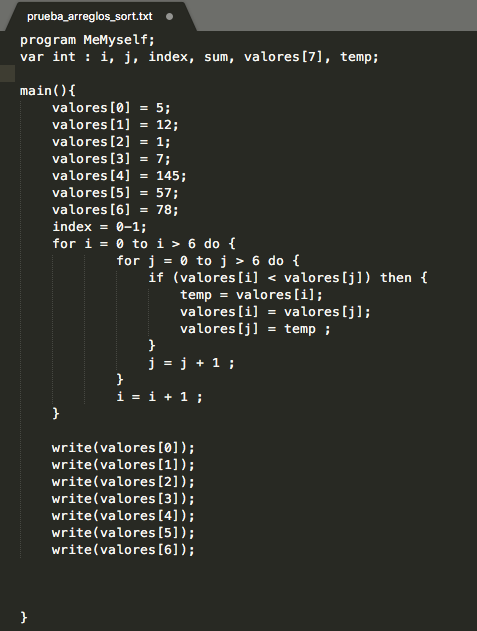


b.Salida



4.Sort de arreglo

a.Entrada



b.Salida

