Texto

El contenido generado por IA puede ser incorrecto.

**Instituto Tecnológico y de Estudios**

**Superiores de Monterrey**

**Desarrollo de aplicaciones avanzadas de ciencias computacionales**

**(Gpo 505)**

Prof. Elda Quiroga

**Documentación final: Compilador**

Diego Velázquez Saldaña

A01177877

30/05/2025

Monterrey, N.L.

1. **Introducción**

El presente documento describe el diseño, implementación y funcionamiento del compilador Webby, un lenguaje imperativo de propósito educativo basado en la sintaxis y estructura del lenguaje BabyDuck. Este proyecto fue desarrollado con el objetivo de integrar los conocimientos adquiridos en las áreas de análisis léxico, sintáctico y semántico, así como generación de código intermedio y ejecución mediante una máquina virtual.

El compilador Webby traduce programas escritos en su propio lenguaje fuente a un conjunto de instrucciones intermedias conocidas como cuádruplos, los cuales son posteriormente ejecutados por una máquina virtual diseñada específicamente para este entorno. Este enfoque permite separar la compilación de la ejecución y simular de forma controlada el comportamiento del programa.

Entre las características principales del lenguaje Webby se encuentran:

* Tipos de datos primitivos: int y float (bool y string como temporales/constantes)
* Declaración y llamado de funciones con tipos y retornos
* Expresiones aritméticas y relacionales
* Estructuras de control como condicionales (if, else) y ciclos (while, for)
* Manejo de variables locales y globales para el múltiple
* Soporte para funciones con parámetros y retorno de valores
* Instrucciones de salida (print)

La herramienta central utilizada para el análisis léxico y sintáctico fue ANTLR, lo que permitió definir una gramática formal (WebbyLexer.g4 y WebbyParser.g4) y generar automáticamente los analizadores correspondientes. La semántica y la generación de código intermedio se implementaron mediante un visitor personalizado en Java, que recorre el árbol sintáctico y produce cuádruplos, además de llevar a cabo todas las validaciones semánticas necesarias.

Por último, el código intermedio se interpreta mediante una máquina virtual, también desarrollada en Java, que simula una arquitectura de memoria segmentada y permite ejecutar paso a paso los programas escritos en Webby, respetando el comportamiento definido por sus cuádruplos.

Esta documentación cubre todos los módulos del compilador, incluyendo el análisis léxico, análisis sintáctico, semántica, generación de cuádruplos, manejo de memoria, máquina virtual y casos de prueba utilizados para validar su funcionamiento.

1. **Análisis Léxico**

El análisis léxico tiene como objetivo transformar el código fuente en una secuencia de **tokens**, que representan las unidades léxicas del lenguaje. Para esta tarea se utilizó la herramienta **ANTLR**, mediante el archivo WebbyLexer.g4, que contiene las expresiones regulares y patrones necesarios para reconocer los distintos tipos de tokens definidos por el lenguaje Webby.

A continuación, se presenta la lista de tokens reconocidos por el compilador, agrupados por categoría. Cada tabla puede incluir el **nombre del token**, su **expresión regular** o patrón, y un **ejemplo si se considera necesario**:

* 1. **Palabras Reservadas**

|  |  |
| --- | --- |
| **Token** | **Expresión Regular** |
| PROGRAM | program |
| MAIN | main |
| END | end |
| VAR | var |
| PRINT | print |
| IF | if |
| ELSE | else |
| WHILE | while |
| DO | do |
| VOID | void |
| INT | int |
| FLOAT | float |
| RETURN | return |
| FOR | for |

* 1. **Identificadores y Literales**

|  |  |  |
| --- | --- | --- |
| **Token** | **Expresión Regular** | **Ejemplo** |
| ID | [a-z\_][a-zA-Z0-9\_]\* | x |
| CTE\_INT | [0-9]+ | 42 |
| CTE\_FLOAT | [0-9]+\.[0-9]+ | 42.42 |
| CTE\_STRING | “(.\*)?” | “Hello World” |

* 1. **Operadores y Símbolos**

|  |  |
| --- | --- |
| **Token** | **Expresión Regular** |
| ADD | + |
| SUB | - |
| MUL | \* |
| DIV | / |
| ASSIGN | = |
| NOTEQUAL | != |
| LESS | < |
| GREATER | > |
| SEMICOLON | ; |
| COLON | : |
| COMMA | , |
| LPAREN | ( |
| RPAREN | ) |
| LBRACE | { |
| RBRACE | } |
| LBRACK | [ |
| RBRACK | ] |

1. **Análisis Sintáctico**

El análisis sintáctico tiene como propósito verificar que la secuencia de tokens producida por el análisis léxico siga las reglas gramaticales del lenguaje Webby. Para ello se utilizó **ANTLR**, mediante el archivo WebbyParser.g4, que define la gramática formal del lenguaje usando una notación similar a BNF (Backus-Naur Form).

* 1. **Herramienta utilizada**
* **ANTLR v4** para la generación automática del parser a partir de la gramática WebbyParser.g4.
  1. **Estructura de la gramática**

La gramática define las reglas para reconocer:

* Declaraciones de variables y funciones
* Llamadas a funciones y manejo de parámetros
* Expresiones aritméticas, relacionales y lógicas
* Instrucciones de control (if, else, while, for)
* Instrucciones de entrada/salida como print
  1. **Reglas gramaticales**

A continuación, se presentan las reglas gramaticales descritas en el archivo:

|  |  |
| --- | --- |
| **Regla** | **Definición** |
| programa | PROGRAM ID SEMICOLON (vars)? (funcs\_list)? MAIN body END |
| vars | VAR (var\_decl)+ |
| var\_decl | id\_list COLON type SEMICOLON |
| id\_list | ID (COMMA ID)\* |
| type | INT | FLOAT |
| params | ID COLON type (COMMA ID COLON type)\* |
| funcs\_list | funcs+ |
| funcs | (type | VOID) ID LPAREN (params)? RPAREN LBRACK (vars)? body RBRACK SEMICOLON |
| body | LBRACE (statement)\* RBRACE |
| statement | assign\_stmt | condition | cycle | f\_call\_stmt | print | return | for |
| return | RETURN (expression)? SEMICOLON |
| assign\_stmt | assign SEMICOLON |
| assign | ID ASSIGN expresion |
| print | PRINT LPAREN print\_args RPAREN SEMICOLON |
| print\_args | print\_arg (COMMA print\_arg)\* |
| print\_arg | expression | CTE\_STRING |
| condition | IF LPAREN expresion RPAREN body ELSE body SEMICOLON |
| cycle | WHILE LPAREN expresion RPAREN DO body SEMICOLON |
| for | FOR LPAREN assign SEMICOLON expresion SEMICOLON assign RPAREN body SEMICOLON |
| expresion | exp ( (LESS | GREATER | NOTEQUAL) exp )\* |
| exp | termino ( (ADD | SUB) termino )\* |
| termino | factor ( (MUL | DIV) factor )\* |
| factor | (ADD | SUB)? (ID | cte | f\_call | LPAREN expresion RPAREN) |
| cte | CTE\_INT | CTE\_FLOAT |
| f\_call\_stmt | f\_call SEMICOLON |
| f\_call | ID LPAREN args? RPAREN |
| args | expresion (COMMA expresion)\* |

* 1. **Pruebas Realizadas**

Se diseñaron cinco archivos .web con distintos escenarios para verificar que el análisis sintáctico del lenguaje funcione correctamente. Las pruebas se ejecutaron usando el archivo Test.java que recorre automáticamente los archivos .web del directorio de pruebas y reporta errores de sintaxis mediante un listener de ANTLR personalizado.

**test01.web – Programa mínimo**

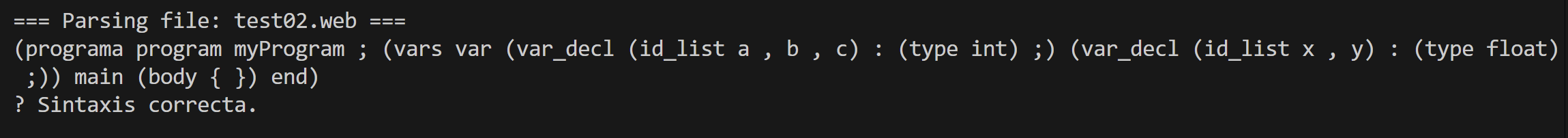
Este archivo representa el caso más simple: un programa con solo la estructura main vacía. Sirve para comprobar que la gramática acepta correctamente un programa sin declaraciones ni instrucciones. **Sintaxis aceptada sin errores.**

**A computer screen with white text

AI-generated content may be incorrect.**

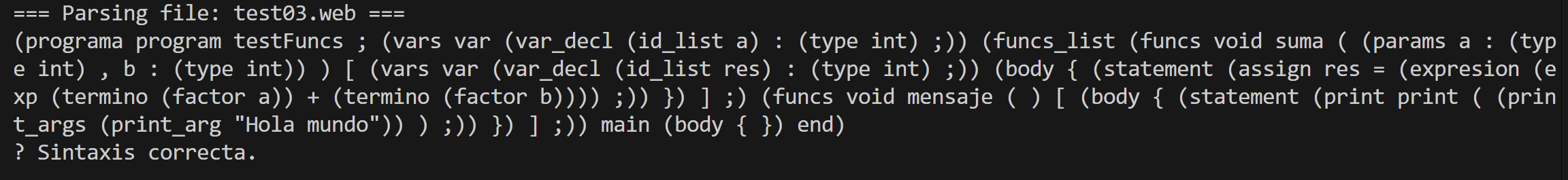
**test02.web – Declaraciones de variables globales**

Incluye varias declaraciones de variables globales de tipo int y float. Permite validar que la sección vars y la sintaxis de listas de identificadores y tipos estén correctamente definidos. **Sintaxis aceptada sin errores.**

****

**test03.web – Funciones con parámetros y cuerpo**

Contiene dos funciones, una con parámetros y una sin ellos. Dentro de las funciones hay declaraciones locales, expresiones de asignación y un print. Esta prueba valida la sintaxis de funciones completas con cuerpo. **Sintaxis aceptada sin errores.**

****

**test04.web – Error por token no reconocido**

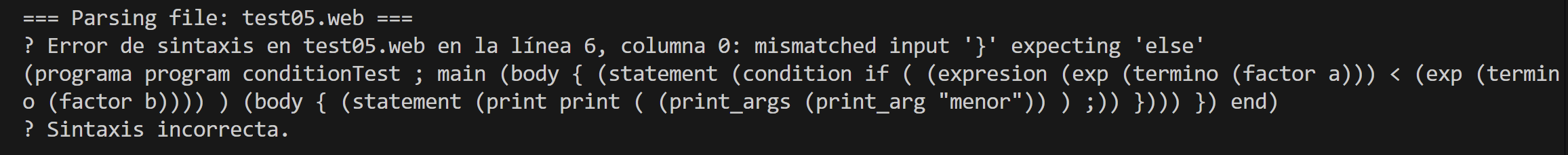
Introduce un carácter inválido (~) dentro de una expresión. Este caso prueba que el Lexer identifique correctamente tokens ilegales y que el Parser reporte adecuadamente el error de sintaxis. **Error detectado por token no reconocido.**

**A black background with white text

AI-generated content may be incorrect.**

**test05.web – Condicional incompleto**

Contiene una estructura if válida en forma, pero sin incluir la rama else, que es obligatoria según la gramática definida. Esta prueba asegura que el parser exija todas las partes requeridas de una estructura condicional. **Error de sintaxis por ausencia de else.**



1. **Descripción de Análisis de Semántica**

La fase semántica se implementa mediante un visitor (SemanticVisitor) que recorre el árbol sintáctico generado por ANTLR. Su principal función es realizar chequeos de tipo, verificación de declaraciones, gestión de memoria, y generación de cuádruplos para la traducción intermedia.

* 1. **Verificación de Variables y Tipos**
* Cada vez que se visita una variable (por ejemplo en asignaciones o uso en expresiones), se verifica que esté declarada en la tabla de funciones (dirFunc).
* Se obtiene el tipo declarado de la variable y el tipo de la expresión o valor que se intenta asignar o usar.
* Se usa el cubo semántico para validar que las operaciones entre tipos son válidas (ejemplo: asignar un FLOAT a un INT es inválido).
* Si hay una incompatibilidad, se lanza una excepción indicando el error semántico.
  1. **Evaluación de Expresiones**
* Las expresiones aritméticas y lógicas se evalúan en el visitor recursivamente, siguiendo la estructura de la gramática:
  + visitExpresion gestiona expresiones con comparadores (<, >, !=, etc.).
  + visitExp maneja suma y resta.
  + visitTermino maneja multiplicación y división.
  + visitFactor maneja valores terminales, llamadas a funciones, constantes y variables, incluyendo signos unarios.
* Cada operación:
  + Saca operandos y tipos de pilas auxiliares (stackContext).
  + Valida tipos con el cubo semántico.
  + Genera un temporal (variable temporal) que almacena el resultado.
  + Asigna dirección de memoria a ese temporal con la clase Memoria.
  + Crea un cuádruplo que representa la operación.
  + Empuja el resultado de nuevo a la pila para posibles operaciones encadenadas.
  1. **Asignaciones**
* En visitAssign, se valida que la variable destino exista y que el tipo de la expresión sea compatible.
* Se genera el cuádruplo de asignación =.
* Los operandos y tipos se manejan con las pilas auxiliares para mantener consistencia y facilitar la generación de código intermedio.
  1. **Llamadas a Funciones**
* En visitF\_call se verifica la existencia de la función.
* Se genera el cuádruplo ERA para reservar espacio.
* Se procesan y verifican los argumentos pasados contra los parámetros esperados (cantidad y tipo).
* Se generan cuádruplos PARAM para pasar argumentos.
* Se genera el cuádruplo GOSUB para saltar a la función.
* Si la función retorna valor distinto de VOID, se asigna a un temporal y se empuja a la pila para que pueda usarse en expresiones.
  1. **Manejo de Argumentos**
* En visitArgs, se recorren las expresiones que forman los argumentos.
* Cada argumento se evalúa y se extrae su dirección y tipo para ser almacenados en colas temporales (argumentQueue y argumentTypeQueue).
* Estas colas se usan luego para generar los cuádruplos de PARAM con verificación de tipos.
  1. **Pila Contextual para Operandos y Tipos**
* Se usa una estructura de datos llamada stackContext para almacenar operandos (nombres o direcciones) y sus tipos, facilitando el seguimiento y validación durante la generación de cuádruplos.
* También mantiene una pila de operadores para manejar correctamente la precedencia y orden de operaciones.
  1. **Cuádruplos**
* Los cuádruplos generados representan la traducción intermedia del código, con estructura [operador, operando1, operando2, resultado].
* Esto facilita posteriormente la ejecución en una Máquina Virtual o la optimización.

1. **Estructuras de Soporte para el Análisis Semántico**

Durante el análisis semántico, se utilizan múltiples estructuras de datos que permiten llevar un control preciso sobre variables, constantes, funciones, tipos de datos y memoria. Estas estructuras están diseñadas para validar correctamente el uso del lenguaje, generar cuádruplos coherentes y gestionar direcciones de memoria de forma sistemática.

* 1. **DirFunc: Directorio de Funciones**

El **Directorio de Funciones (DirFunc)** es la estructura central del análisis semántico. Internamente utiliza un HashMap<String, FunctionInfo>, donde la clave es el nombre de la función (incluyendo "global" para el ámbito global), y el valor es un objeto FunctionInfo que encapsula toda la información semántica de esa función:

* Cada FunctionInfo contiene:
  + Una instancia de VarTable local (otra estructura hash) para las variables de esa función.
  + Una lista (ArrayList<VarInfo>) de parámetros con su nombre, tipo y dirección.
  + El tipo de retorno (VarType).
  + El número de cuádruplo donde inicia la función.
  + La dirección de retorno (si aplica).
* DirFunc también mantiene un campo currentFunction que indica en qué ámbito se está trabajando (por ejemplo, "global" o una función específica).
* Permite validar:
  + Existencia de funciones al ser llamadas.
  + Tipos y número de parámetros.
  + Tipos de retorno.
  + Detección de redefiniciones.
  1. **VarTable: Tabla de Variables**

La **Tabla de Variables (VarTable)** es una estructura basada en un HashMap<String, VarInfo>, donde:

* La clave es el nombre de la variable.
* El valor es un objeto VarInfo que contiene el tipo (VarType) y la dirección de memoria asignada.

Esta tabla se instancia de forma individual para:

* El ámbito global.
* Cada función (a través de su FunctionInfo).

Permite:

* Verificar si una variable ya ha sido declarada.
* Consultar su tipo y dirección de memoria.
* Distinguir entre ámbitos local y global.
  1. **VarType: Enumeración de Tipos**

La **enumeración VarType** define los tipos válidos del lenguaje:

* INT, FLOAT, BOOL, STRING, VOID.

Se utiliza para:

* Validar tipos en declaraciones.
* Consultar el tipo esperado en expresiones.
* Operar con el Cubo Semántico.
* Controlar el tipo de retorno de funciones.
  1. **ConstTable: Tabla de Constantes**

La **Tabla de Constantes (ConstTable)** se implementa como un HashMap<Object, Integer>, donde:

* La clave es el valor constante literal (por ejemplo, 3, 3.14, "hola").
* El valor es la dirección de memoria única asignada a esa constante.

Funciones clave:

* Verifica si una constante ya fue registrada para evitar duplicación.
* Solicita una dirección de memoria al MemoryManager si es una nueva constante.
* Sirve para acceder a constantes de manera uniforme desde cuádruplos.
  1. **CuboSemantico: Validación de Tipos**

El **Cubo Semántico (CuboSemantico)** se implementa como una estructura de tipo HashMap<VarType, HashMap<VarType, HashMap<String, VarType>>>, es decir, un mapa tridimensional que permite consultar el tipo resultante de aplicar un operador binario entre dos operandos de tipos específicos.

Uso:

* Si la combinación es válida, devuelve el tipo resultante.
* Si no es válida, devuelve null, lo que activa un error semántico.

Soporta operadores:

* Aritméticos (+, -, \*, /)
* Relacionales (<, >, !=, ==)
* Booleanos (&&, ||, !)
* De asignación (=)

Aquí se muestra a continuación que se asignó para cada tipo de operación:

1. **Operaciones aritméticas ( + , - , \* , / )**

|  |  |  |
| --- | --- | --- |
| **Izquierda \ Derecha** | **INT** | **FLOAT** |
| **INT** | INT | FLOAT |
| **FLOAT** | FLOAT | FLOAT |

1. **Operaciones relacionales ( < , > , != )**

|  |  |  |
| --- | --- | --- |
| **Izquierda \ Derecha** | **INT** | **FLOAT** |
| **INT** | BOOL | BOOL |
| **FLOAT** | BOOL | BOOL |

1. **Asignación ( = )**

|  |  |  |
| --- | --- | --- |
| **Izquierda \ Derecha** | **INT** | **FLOAT** |
| **INT** | INT | ERROR |
| **FLOAT** | ERROR | FLOAT |

* 1. **MemoryManager: Gestor de Memoria**

El **MemoryManager** es una clase que administra la asignación de direcciones de memoria para:

* Variables globales.
* Variables locales.
* Temporales.
* Constantes.

Estructura:

* Internamente, mantiene contadores independientes por segmento y tipo de dato (ej. globalIntCounter, tempFloatCounter, etc.).
* Se definieron rangos base específicos para cada tipo de memoria:
  + **Global**: INT desde 1000 y FLOAT desde 3000
  + **Local**: INT desde 5000 y FLOAT desde 7000
  + **Temporal**: INT desde 9000, FLOAT desde 11000 y BOOL desde 13000
  + **Constantes**: INT desde 15000, FLOAT desde 17000, STRING desde 19000

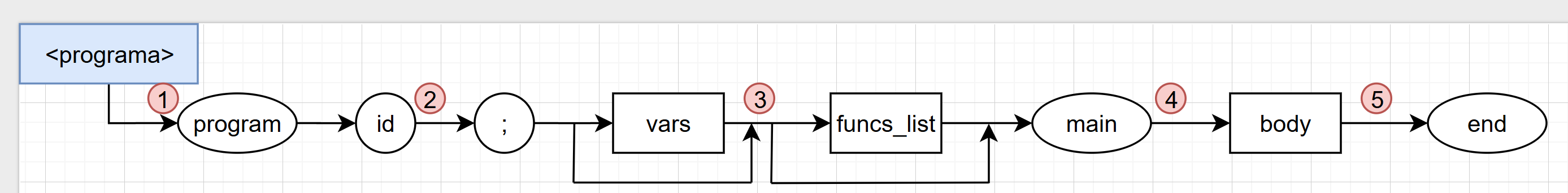
Cuando se solicita una nueva dirección:

* El MemoryManager incrementa el contador correspondiente y devuelve la siguiente dirección válida.

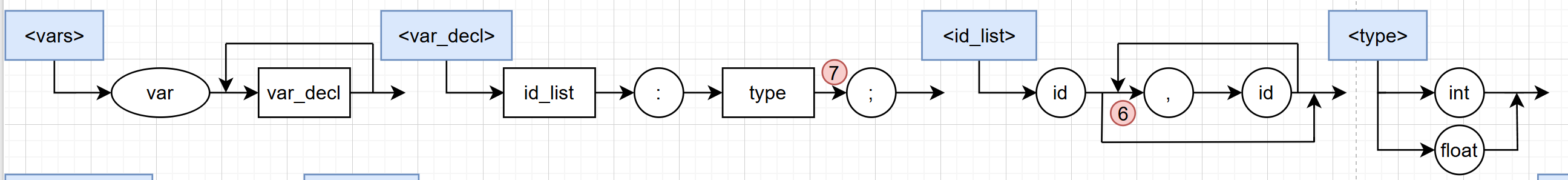
Permite una asignación limpia y sistemática que evita colisiones y facilita el trabajo de la Máquina Virtual.

1. **Diagramas y Puntos Neurálgicos del Análisis Semántico**

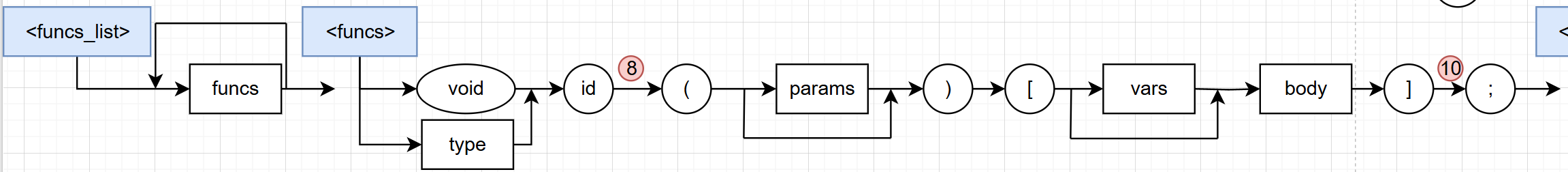
A continuación, se presentan los diagramas más relevantes para el funcionamiento del análisis semántico, en caso de querer verse a profundidad el link de los diagramas por si solo se puede encontrar en el Apéndice.

****

1. Creación de List<Quadruples> quadruples, MemoryManager memory, ConstTable constTable, SemanticStackContext stackContext, int tempCounter
2. Creación de DirFunc dirFunc
   1. dirFunc.globalName = id
   2. dirFunc. currentFunction = id
   3. creación de HashMap Functions en dirFunc e insertar id con nueva FunctionInfo
3. Agregar cuádruplo GOTO para saltar al body del main y guardar en gotoMainIndex el índice del salto
4. Definir DirFunc.setCurrentFunction(globalName) y modificar cuádruplo en gotoMainIndex a saltar en quadruples.size()
5. Agregar cúadruplo ENDPROG



1. Si dirFunc. variableExistsInCurrentScope(id) lanzar error de variable doblemente declarada.
2. varType = type, iterar por cada id de id\_list y definir address
   1. Si dirFunc.globalIsCurrent()
      1. entonces address = memoria.assignGlobalAddress( varType)
      2. sino address = memoria.assignLocalAddress(varType)
   2. dirFunc.addVariable(id, varType, address)



A diagram of a flowchart

AI-generated content may be incorrect.

1. funcName = id
   1. Si !dirFunc.addFunction(funcName)
      1. entonces lanzar error de función doblemente declarada,
      2. sino dirFunc.setCurrentFunction(funcName), dirFunc.setFunctionStartQuad(funcName, quadruples.size())
   2. Si la función no es de tipo VOID
      1. Entonces retAddr = memoria.assignTempAddress(returnType), dirFunc.setReturnAddress(funcName, retAddr) y dirFunc.addVariable("\_\_return", returnType, retAddr)
      2. Además, inicializar hasUnconditionalReturn como falso para asegurarnos que exista un return sin condicionales de por medio
2. Si dirFunc.variableExistsInCurrentScope(id)
   1. Entonces lanzar error de variable doblemente declarada
   2. Sino address = memoria.assignLocalAddress(varType) y dirFunc.addParameter(id, varType, address)
3. Resetear contadores para asignar dirección de memoria con memoria.resetLocalAndTemp() por si viene otra función, insertar cuádruplo de ENDFUNC

A diagram of a computer code

AI-generated content may be incorrect.

1. Si !dirFunc.variableExists(id) lanzar error de uso de variable no declarada
2. Obtener varType = dirFunc.getVariableType(id), obtener exprType = stackContext.popType() y exprValue = stackContext.popOperand(), si cuboSemantico.getResultType("=", varType, exprType) es null entonces lanzar error de asignación, sino generar cuádruplo de asignación

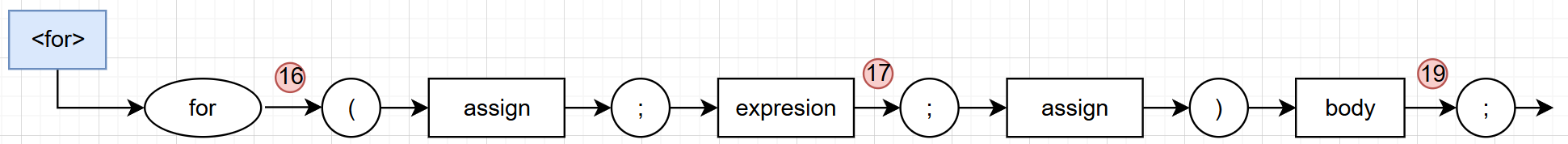
A diagram of a diagram

AI-generated content may be incorrect.

1. Obtener variable/temporal conditionResult = visit(ctx.expresion()), generar cuádruplo GOTOF de conditionResult y guardar índice de cuádruplo en gotofIndex
2. Crear GOTO para hacer salto si es que se entro al body verdadero del if y guardar el índice del cuádruplo en gotoEndIndex, modificar cuádruplo en gotofIndex a saltar en quadruples.size()
3. Modificar cuádruplo en gotoEndIndex a saltar en quadruples.size()

A graph with numbers and a number

AI-generated content may be incorrect.



1. Guardar en loopStart = quadruples.size()
2. Generar cuádruplo GOTOF y guardar índice de cuádruplo en gotofIndex
3. Generar cuádruplo GOTO e indicar salto en loopStart y modificar cúadruplo GOTOF con quadruples.size()
4. Igual que 18 pero antes visitar el segundo assign del for para agregar los cuádruplos correspondientes

A diagram of a computer

AI-generated content may be incorrect.

A diagram of a function

AI-generated content may be incorrect.

1. Si !dirFunc.hasFunction(funcName)
   1. Entonces lanzar error de llamado de función no existente
   2. Sino generar cuádruplo ERA con funcName
2. Se vacían argumentQueue y argumentTypeQueue
3. Se hacen argAddr = stackContext.popOperand() y argType = stackContext.popType() y se agregan argumentQueue.add(argAddr) y argumentTypeQueue.add(argType)
4. Si argumentQueue.size() != expectedParams.size()
   1. Entonces se lanza error de número de argumentos en la función no son los esperados
   2. Sino se itera por toda la queue y se compara el argType con el expectedType para verificar que el llamado tiene el mismo tipo de variable que el parámetro correspondiente de la función y se agrega cuádruplo PARAM
5. Se obtiene startQuad = dirFunc.getFunctionStartQuad(funcName) y se genera cuádruplo GOSUB con el startQuad
   1. Si dirFunc.getFunctionReturnType(funcName) no es VOID
      1. Entonces se genera un temporal, con su dirección, se añade la variable y finalmente se agrega un cúadruplo de asignación para asignar el resultado de la función al temporal generado

A diagram of a graph

AI-generated content may be incorrect.

1. Si constTable.hasConstant(str) agregar CTE\_STRING a la tabla de constantes, asignandole una dirección de memoria, Generar cuádruplo PRINT con el string
2. Retornar el resultado de la operación
3. Si viene otro print\_arg
   1. Entonces generar cúadrplo con PRINT
   2. Sino generar cuádruplo con PRINTLN

A diagram of a flowchart

AI-generated content may be incorrect.

1. Si !constTable.hasConstant(value)
   1. Entonces agregar consTable la constante, asignándole antes su respectiva dirección de memoria

A diagram of a computer

AI-generated content may be incorrect.

1. Si !dirFunc.variableExists(id)
   1. Entonces lanzar error de variable no existente, sino varType = dirFunc.getVariableType(id)
2. stackContext.pushOperand(var, varType) donde se hacen los respectivos push al stackOperand y al typeOperand

A diagram of a computer

AI-generated content may be incorrect.

1. Si stackContext.topOperator() es \* o / entonces
   1. rightOperand = stackContext.popOperand(), rightType = stackContext.popType(), leftOperand = stackContext.popOperand(), leftType = stackContext.popType(), op = stackContext.popOperator()
   2. se hace validación de resultType = cuboSemantico.getResultType(op, leftType, rightType) y si es null regresa error por tipos incompatibles
   3. tempName = generateTemp() y se genera tempAddress = memoria.assignTempAddress(resultType) para hacer dirFunc.addVariable(tempName, resultType, tempAddress)
   4. Genera cuádruplo con operador, leftOperand, rightOperand y tempName
   5. stackContext.pushOperand(tempName, resultType)
2. stackContext.pushOperator(op)
3. Si stackContext.topOperator() es + o – hacer número #29
4. Si stackContext.topOperator() es <, > o != hacer número #29
5. **Pruebas de Semántica**
   1. **Validaciones iniciales de semántica**

Se llevaron a cabo tres pruebas específicas para validar la capacidad del compilador de detectar errores semánticos, pese a que el análisis sintáctico fue exitoso en todos los casos. Los archivos .web fueron construidos sin errores de gramática, pero con problemas relacionados con las reglas del contexto, como declaraciones duplicadas o usos indebidos de identificadores.

**test01.web – Función doblemente declarada**

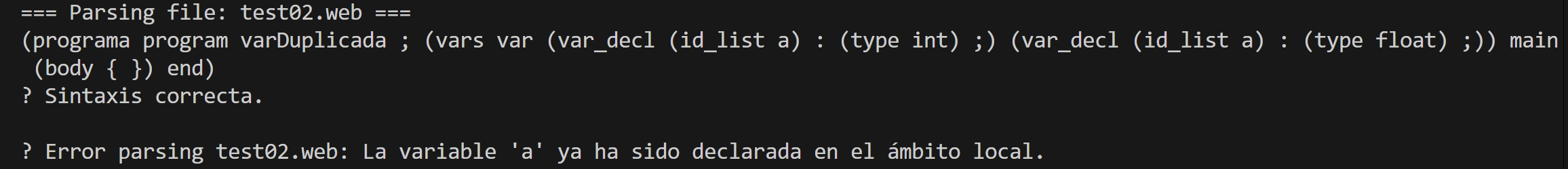
Este archivo define dos funciones con el mismo identificador hola. Ambas funciones tienen firmas idénticas y se encuentran en el mismo ámbito global. **Error semántico: Función 'hola' ya fue declarada.**

A black screen with white text

AI-generated content may be incorrect.

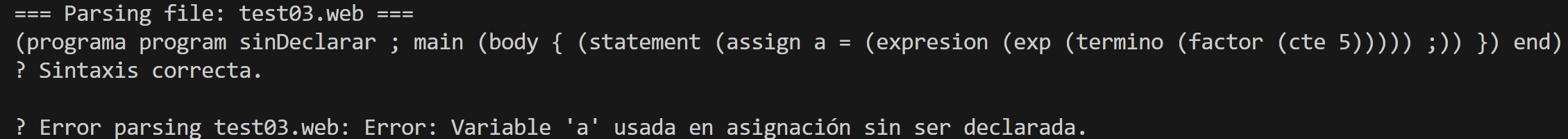
**test02.web – Variable doblemente declarada**

Se intenta declarar la variable a dos veces como variable global, primero como int y luego como float. **Error semántico: La variable 'a' ya ha sido declarada en el ámbito local.**



**test03.web – Uso de variable no declarada**

Dentro del bloque main, se realiza una asignación a la variable a sin haberla declarado previamente en ninguna parte del programa. **Error semántico: Variable 'a' usada en asignación sin ser declarada.**



Estas pruebas confirman que el análisis semántico del compilador identifica correctamente violaciones al contexto del lenguaje, incluso si el programa es sintácticamente válido.

* 1. **Generación de cuádruplos**

Se llevaron a cabo tres archivos de prueba para verificar la correcta generación de cuádruplos en distintos contextos del lenguaje. A continuación, se describen brevemente:

**test01.web – Asignaciones complejas**

Este archivo contiene varias asignaciones que mezclan operaciones aritméticas simples y compuestas. Se genera correctamente una secuencia de cuádruplos que reflejan el orden y jerarquía de operaciones.

A black screen with white text

AI-generated content may be incorrect.

**test02.web – Print con operaciones**

Este ejemplo incluye una instrucción print que imprime tanto texto como el resultado de una expresión aritmética. Se verifica que el cuádruplo PRINT se pueda usar tanto para cadenas como para resultados de operaciones.

A black screen with white text

AI-generated content may be incorrect.

**test03.web – Condicional con comparación**

Este archivo prueba la evaluación de una condición con operadores relacionales y dos bloques if / else. Se generan correctamente los cuádruplos para ambas ramas, evaluando primero la condición y después cada cuerpo.

A black screen with white text

AI-generated content may be incorrect.

* 1. **Generación de cuádruplos con memoria**

Se diseñaron dos archivos adicionales para comprobar el correcto funcionamiento de la generación de cuádruplos en estructuras de control condicional y cíclica. A continuación, se presentan los detalles:

**test04.web – Condicional simple (if / else)**

Este archivo contiene una instrucción if con una comparación entre una variable y un valor constante. Se verifica que se genera un cuádruplo GOTOF para omitir el cuerpo del if si la condición no se cumple, que se incluye un GOTO adicional para saltar la rama del else y que los saltos son actualizados correctamente para reflejar el flujo del programa.

A computer screen with white text

AI-generated content may be incorrect.

**test05.web – Ciclo while simple**

Este archivo evalúa el comportamiento de un ciclo while con una condición sencilla y una actualización interna de la variable. Se comprueba que el cuádruplo GOTOF se utiliza para salir del ciclo si la condición es falsa, que se genera un GOTO que regresa al inicio del ciclo para reevaluar la condición y que se produce una secuencia repetitiva de cuádruplos que representa adecuadamente la iteración hasta que la condición se vuelve falsa.

A black screen with white text

AI-generated content may be incorrect.

**test06.web – Ciclo for simple**

Este archivo evalúa el comportamiento de un ciclo for con una asignación inicial, una condición booleana de control y una actualización del contador. Se comprueba que, se genera correctamente la asignación inicial antes del ciclo, el cuádruplo GOTOF evalúa la condición y permite salir del ciclo si es falsa, los cuádruplos dentro del cuerpo del ciclo se ejecutan correctamente mientras la condición sea verdadera, se genera un cuádruplo para la actualización del contador al final de cada iteración, se produce un cuádruplo GOTO que regresa al inicio del ciclo para reevaluar la condición y se actualiza correctamente con la posición de salida del ciclo, asegurando que la ejecución continúe normalmente cuando se termina la iteración.

A computer screen with white text

AI-generated content may be incorrect.

1. **Algoritmo Principal de Máquina Virtual**

La Máquina Virtual (VM) de Webby es el componente encargado de ejecutar los programas escritos en el lenguaje, a partir de los cuádruplos generados por el compilador. Su funcionamiento está inspirado en una arquitectura basada en pila de llamadas y segmentación de memoria, simulando un entorno de ejecución que permite manejar expresiones, condicionales, ciclos, llamadas a funciones y retorno de valores.

* 1. **Entradas de la Máquina Virtual**

Al iniciar su ejecución, la VM recibe tres elementos clave:

* **Lista de cuádruplos** (List<Quadruple>): Representa el código intermedio generado por el compilador. Cada cuádruplo contiene un operador, uno o dos operandos (direcciones de memoria), y una dirección destino donde guardar el resultado.
* **Directorio de funciones** (DirFunc): Contiene información sobre cada función del programa, incluyendo su tabla de variables, parámetros, tipo de retorno y dirección de inicio dentro del código intermedio.
* **Tabla de constantes** (ConstTable): Mapea los valores constantes utilizados en el programa a sus respectivas direcciones de memoria.

Estos elementos permiten a la máquina virtual reconstruir el estado completo del programa y simular su ejecución paso a paso.

* 1. **Proceso de Ejecución**

La ejecución comienza en la instrucción 0 y continúa de forma secuencial, salvo que alguna instrucción modifique el flujo (como GOTO, GOTOF o GOSUB). La VM mantiene un puntero de instrucción (instructionPointer) que señala el cuádruplo actual a ejecutar.

El algoritmo general es:

1. Cargar constantes y variables globales en memoria.
2. Iterar mientras haya cuádruplos por ejecutar.
3. Identificar el operador del cuádruplo actual y delegar a la operación correspondiente (aritmética, asignación, condicional, etc.).
4. Ejecutar la operación usando los valores en memoria, actualizando los resultados donde corresponda.
5. Avanzar al siguiente cuádruplo, o saltar según el flujo de control.
   1. **Soporte para Funciones**

Cuando se llama una función:

* Se prepara un nuevo marco de memoria (MemoryFrame) para las variables locales y temporales de la función.
* Se cargan los parámetros con instrucciones PARAM.
* La instrucción GOSUB salta al inicio de la función y guarda la dirección de retorno en una pila (callStack).
* RETURN y ENDFUNC manejan el retorno de valores y limpieza de memoria.

De este modo, la VM soporta llamadas anidadas, recursividad y funciones con valor de retorno.

* 1. **Operadores Soportados**

A continuación, se listan los operadores ya implementados:

|  |  |  |
| --- | --- | --- |
| **Código** | **Operación** | **Descripción** |
| 1-4 | +, -, \*, / | Operaciones aritméticas entre enteros o flotantes |
| 5 | = | Asignación de valores |
| 6-8 | !=, <, > | Comparaciones relacionales |
| 9–10 | PRINT  PRINTLN | Salida a consola |
| 11–12 | GOTO  GOTOF | Control de flujo |
| 13–15 | ERA  PARAM  GOSUB | Preparación y ejecución de funciones |
| 16-17 | RETURN ENDFUNC | Retorno de funciones y fin de contexto (pop de memoria local/temporal actual) |
| 18 | |  | | --- | |  |  |  | | --- | | ENDPROG | | Fin del programa |

1. **Memoria en la Máquina Virtual**

La memoria de la máquina virtual se encarga de almacenar y acceder a todos los valores requeridos durante la ejecución del programa: variables globales, locales, temporales y constantes. Para lograr una correcta segmentación y control de contexto, la memoria se divide en distintos segmentos con rangos de direcciones predefinidos. A continuación se explica en detalle su estructura y funcionamiento.

* 1. **Segmentos de Memoria**

La memoria está dividida en distintos segmentos según el tipo de variable y su contexto de uso:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Tipo** | **Rango de Direcciones** | **Estructura interna** | | Global int | 1000 - 2999 | ArrayList<Integer> | | Global float | 3000 - 4999 | ArrayList<Float> | | Local int | 5000 - 6999 | MemoryFrame.localInt | | Local float | 7000 - 8999 | MemoryFrame.localFloat | | Temporal int | 9000 - 10999 | MemoryFrame.tempInt | | Temporal float | 11000 - 12999 | MemoryFrame.tempFloat | | Temporal bool | 13000 - 14999 | MemoryFrame.tempBool | | Const int | 15000 - 16999 | ArrayList<Integer> | | Const float | 17000 - 18999 | ArrayList<Float> | | Const string | 19000 - 20999 | ArrayList<String> | |  |  |

Estas direcciones son manipuladas directamente por los cuádruplos durante la ejecución.

* 1. **Marco de Memoria (MemoryFrame)**

Cada llamada a función activa un nuevo **frame de memoria** que contiene:

* Variables locales (enteros y flotantes)
* Variables temporales (enteros, flotantes, booleanos)

Este frame es gestionado mediante una pila (Deque<MemoryFrame>) para respetar el orden de llamadas (LIFO). Al iniciar una función (cuádruplo ERA), se crea un **pendingFrame** donde se almacenan los parámetros. Este frame es activado con commitPendingFrame al ejecutar GOSUB.

Cuando la función termina (cuádruplos RETURN o ENDFUNC), el frame es eliminado con popFrame.

* 1. **Constantes**

Las constantes se cargan al inicio de la ejecución usando loadConstants(), que llena los arreglos correspondientes (constInt, constFloat, constString) basándose en las direcciones y valores definidos en la ConstTable.

* 1. **Acceso y modificación**

La clase VirtualMemory provee dos métodos clave:

* Object getValue(int address)  
  Devuelve el valor en una dirección, dependiendo del rango.
* void setValue(int address, Object value)  
  Asigna un valor a una dirección, con validación del tipo y segmento.
  1. **Consideraciones de diseño**
* Las estructuras de memoria crecen dinámicamente según se necesiten nuevas direcciones.
* No se permite modificar las constantes: setValue lanza una excepción si se intenta.
* Se puede obtener el tamaño de la pila de frames en tiempo real con getMemoryFramesSize().

1. **Ejemplo de ejecución de Máquina Virtual**
   1. **Programa para ejecutar**

Para tener una mejor comprensión de la Máquina Virtual a continuación se muestra un ejemplo de un programa a ejecutar.

**program testCuadrado;**

**var x: int;**

**int cuadrado(n: int) [ { return n \* n; }];**

**main {**

**x = cuadrado(2);**

**print(x);**

**}**

**End**

* 1. **Cuádruplos generados**

En este programa se generan los siguientes cuádruplos

**Cuádruplos generados:**

**(GOTO, , , #4)**

**(\*, n, n, t0)**

**(RETURN, , , t0)**

**(ENDFUNC, , , )**

**(ERA, , , cuadrado)**

**(PARAM, 2, , param0)**

**(GOSUB, , , #1)**

**(=, cuadrado, , t1)**

**(=, t1, , x)**

**(PRINTLN, , , x)**

**(ENDPROG, , , )**

* 1. **Paso a paso de ejecución**

Aquí el seguimiento de la ejecución, considerando los cuádruplos y la memoria que se maneja

**Inicialización**

* Se cargan constantes y variables globales en memoria:
  + loadConstants(constTable)
  + loadGlobalVariables(...)
* Se inicia el instructionPointer = 0.

**Cuádruplo 0: (GOTO, , , #4)**

* Acción: Salta la ejecución al cuádruplo 4, donde inicia el main.
* Justificación: Esto evita ejecutar el cuerpo de funciones al inicio.
* Efecto: instructionPointer = 4.

**Cuádruplo 4: (ERA, , , cuadrado)**

* Acción: Se prepara la llamada a la función cuadrado.
* Se crea un pendingFrame vacío con las variables locales y temporales de cuadrado.
* Se guarda currentFunctionName = "cuadrado".
* paramCounter = 0.

**Cuádruplo 5: (PARAM, 2, , param0)**

* Acción: Se pasa el argumento 2 a la función cuadrado.
* 2 ya debe estar en memoria de constantes (constInt[2] → 17002).
* Se copia el valor a la dirección param0 de la función cuadrado.
* Se incrementa paramCounter = 1.

**Cuádruplo 6: (GOSUB, , , #1)**

* Acción: Salta a la dirección de inicio de cuadrado, que es cuádruplo 1.
* Se guarda el instructionPointer = 7 en la callStack.
* El pendingFrame se convierte en el nuevo frame activo (commitPendingFrame).

**Cuádruplo 1: (\*, n, n, t0)**

* Acción: Se multiplica el parámetro n por sí mismo (2 \* 2).
* n está en memoria local.
* Resultado se guarda en un temporal (t0), que está en la memoria temporal actual.
* Ejemplo: t0 = 4.

**Cuádruplo 2: (RETURN, , , t0)**

* Acción: Se retorna el valor calculado en t0 a la dirección de retorno de la función.
* El valor se guarda en el espacio reservado para cuadrado en el directorio de funciones.
* Se hace popFrame() → se destruye el contexto local de cuadrado.
* Se recupera el instructionPointer = 7 (retorno).

**Cuádruplo 7: (=, cuadrado, , t1)**

* Acción: Asigna el valor retornado por cuadrado a un temporal t1.
* Como leftAddr == 0, se interpreta cuadrado como nombre de función.
* Se accede a dirFunc.getReturnAddress("cuadrado") para obtener el valor retornado.
* Luego se asigna a t1 (temporal en main).

**Cuádruplo 8: (=, t1, , x)**

* Acción: Asigna el valor de t1 (4) a la variable global x.
* x está en el segmento global de enteros.
* Se copia el valor desde la memoria temporal a memoria global.

**Cuádruplo 9: (PRINTLN, , , x)**

* Acción: Imprime el valor de x, que es 4.
* Salida en consola: 4.

**Cuádruplo 10: (ENDPROG, , , )**

* Acción: Finaliza la ejecución del programa.
* return del método execute().

1. **Archivos de prueba**

A continuación se presentan distintos archivos de prueba desarrollados en el lenguaje Webby. Cada uno se diseñó para verificar el correcto funcionamiento de la ejecución de programas tanto desde el main como desde funciones, incluyendo el uso de estructuras de control como ciclos, condicionales y recursión. Las pruebas cubren los algoritmos clásicos de factorial y Fibonacci, en diferentes variantes, demostrando las capacidades semánticas y de ejecución del compilador.

**10.1. fact.web**

**Contexto:**  
Implementación del cálculo del factorial directamente desde el main, usando un ciclo for.

**Este archivo comprueba:**

* Declaración y uso de variables globales.
* Ejecución de código dentro de main.
* Uso del ciclo for.
* Asignaciones y expresiones aritméticas.
* Correcta impresión de resultados concatenando texto y valores enteros.

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**10.2. fact\_func.web (función void con ciclo while)**

**Contexto:**  
El cálculo del factorial se realiza dentro de una función void que utiliza un ciclo while.

**Este archivo comprueba:**

* Definición y llamada a funciones tipo void.
* Paso de parámetros por valor.
* Ciclo while dentro de una función.
* Impresión desde dentro de funciones.
* Correcta manipulación de variables locales dentro de la función.

A computer screen shot of a code

AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.

**10.3. fact\_rec.web (función void recursiva con múltiples parámetros)**

**Contexto:**  
Versión recursiva de factorial en una función void, utilizando parámetros auxiliares para acumular el resultado.

**Este archivo comprueba:**

* Definición y ejecución de funciones recursivas void.
* Paso de múltiples parámetros a funciones.
* Condicionales con if y else.
* Evaluación correcta de expresiones recursivas.
* Mantenimiento del flujo de control sin return.

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**10.4. fact\_rec2.web (función int recursiva con return)**

**Contexto:**  
Implementación recursiva del factorial con retorno de valor entero.

**Este archivo comprueba:**

* Definición de funciones que retornan valores (int).
* Ejecución de llamadas recursivas con return.
* Asignación de valores retornados a variables.
* Uso de expresiones return anidadas.
* Impresión del resultado desde main tras recibir el valor de retorno.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**10.5. fib.web**

**Contexto:**  
Implementación iterativa de la serie de Fibonacci dentro del main, usando un ciclo while.

**Este archivo comprueba:**

* Declaración de múltiples variables y su actualización.
* Uso de while para control iterativo.
* Impresión de resultados en cada iteración.
* Operaciones aritméticas y reasignación de valores.

A screen shot of a computer program

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

**10.6. fib\_func.web (función void con ciclo)**

**Contexto:**  
Versión iterativa de Fibonacci dentro de una función void.

**Este archivo comprueba:**

* Función void con múltiples variables locales.
* Control iterativo con while dentro de una función.
* Llamadas a funciones desde main.
* Impresiones desde funciones con varias variables.

A screenshot of a computer program

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

**10.7. fib\_rec.web (función void recursiva con múltiples parámetros)**

**Contexto:**  
Versión recursiva de Fibonacci como función void, usando parámetros para mantener el estado de la secuencia.

**Este archivo comprueba:**

* Recursión en funciones void.
* Paso de varios parámetros incluyendo contadores y acumuladores.
* Condicionales y recursividad controlada.
* Impresión de valores por posición de la serie.

A computer screen shot of a program code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**10.8. fib\_rec2.web (función int recursiva con return)**

**Contexto:**  
Versión clásica recursiva de Fibonacci con retorno de entero y suma de llamadas recursivas.

**Este archivo comprueba:**

* Función int con múltiples llamadas recursivas.
* Evaluación de expresiones que suman resultados de return.
* Condiciones de corte base (n < 2).
* Impresión directa del resultado retornado desde main.

A computer screen with white text

AI-generated content may be incorrect.

**A screen shot of a computer program

AI-generated content may be incorrect.**

1. **Apéndice**

**Liga de GitHub:** <https://github.com/diegovelsal/webby_compilador>

**Liga de Diagramas:** <https://drive.google.com/file/d/17ziCW2X6KB1pywQJncUa47JeXYyViKch/view?usp=sharing>

**¿Cómo ejecutar las pruebas?** Se pueden ver los comandos necesarios desde el README.md del repositorio, solo asegúrese de clonar el repositorio y estar en el directorio general del repositorio.