

Liga de GitHub: https://github.com/diegovelsal/webby_compilador

BabyDuck – Entrega #0

1.- Diseñar las Expresiones Regulares que representan a los diferentes elementos de léxico que ahí aparecen.

| Token | Expresión Regular |
|------------|---------------------|
| ID | [a-z][a-zA-Z0-9]* |
| CTE INT | [0-9] + |
| CTE FLOAT | [0-9] +\.[0-9] + |
| CTE STRING | “(.*?”” |
| ADD | + |
| SUB | - |
| MUL | * |
| DIV | / |
| ASSIGN | = |
| NOTEQUAL | != |
| LESS | < |
| GREATER | > |
| SEMICOLON | ; |
| COLON | : |
| COMMA | , |
| LPAREN | (|
| RPAREN |) |
| LBRACE | { |
| RBRACE | } |
| LBRACK | [|
| RBRACK |] |
| PROGRAM | program |
| MAIN | main |
| END | end |
| VAR | var |
| PRINT | print |
| IF | if |
| ELSE | else |
| WHILE | while |
| DO | do |
| VOID | void |
| INT | int |
| FLOAT | float |

2.- Listar todos los Tokens que serán reconocidos por el lenguaje

Palabras Clave

- program
- main
- end
- if
- else
- while
- do
- print
- void
- type (int o float)

Variables

- id
- cte_int
- cte_float
- cte_string

Operadores aritméticos

- +
- -
- *
- /

Operadores de comparación

- =
- !=
- <
- >

Delimitadores

- (
-)
- {
- }
- [
-]
- ;
- ,
- :

3.- Diseñar las reglas gramaticales (Context Free Grammar) equivalentes a los diagramas.

| 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 | VOID ID LPAREN (params)? RPAREN LBRACK (vars)? body RBRACK SEMICOLON |
| body | LBRACE (statement)* RBRACE |
| statement | assign condition cycle f_call print |
| assign | ID ASSIGN expresion SEMICOLON |
| 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 |
| expresion | exp (LESS GREATER NOTEQUAL) exp)* |
| exp | termino ((ADD SUB) termino)* |
| termino | factor ((MUL DIV) factor)* |
| factor | (ADD SUB)? (ID cte LPAREN expresion RPAREN) |
| cte | CTE_INT CTE_FLOAT |
| f_call | ID LPAREN args? RPAREN SEMICOLON |
| args | expresion (COMMA expresion)* |

BabyDuck – Entrega #1

Como parte del desarrollo del compilador para el lenguaje BabyDuck, esta primera entrega tuvo como objetivo comprender y aplicar herramientas automáticas para la generación de analizadores léxicos (scanners) y sintácticos (parsers). Para ello, se realizó una investigación sobre distintas opciones existentes, evaluando su documentación, facilidad de uso, compatibilidad con el lenguaje de programación que elegí para aprender (que en mi caso fue Java) y su potencial para integrarse en futuras etapas del compilador.

Las principales herramientas consideradas fueron ANTLR, Lex/Yacc, Flex/Bison y JavaCC. Lex/Yacc y Flex/Bison, aunque ampliamente utilizados en entornos de bajo nivel como C/C++, presentan una curva de aprendizaje más pronunciada y una sintaxis menos intuitiva. JavaCC, por otro lado, es una alternativa viable en Java, pero tiene una comunidad más reducida y menor cantidad de recursos actualizados. En cambio, ANTLR (Another Tool for Language Recognition) resultó ser la opción más adecuada debido a su amplia documentación, soporte activo, sintaxis clara y modular mediante archivos .g4, así

como su compatibilidad directa con Java. Por estas razones, se seleccionó ANTLR 4 como la herramienta principal para esta y futuras entregas del compilador BabyDuck.

La implementación del lenguaje comenzó con la separación del léxico y la sintaxis en archivos diferentes. En el archivo BabyDuckLexer.g4 se definieron todas las reglas gramaticales. Aquí se dividieron dos archivos, el primero siendo **WebbyLexer.g4** donde se definieron los tokens tal y como se escribieron en la entrega #0 (primera tabla). Por otra parte, en el archivo **WebbyParser.g4** se definió la gramática y fue redactada tal y como se especificó en la entrega #0 (segunda tabla). La implementación respetó el uso de reglas y ANTLR se encargó de generar automáticamente las clases Java correspondientes al lexer y parser.

Para validar el funcionamiento del compilador, se desarrolló un conjunto de once casos de prueba que cubren los elementos esenciales del lenguaje. Estos casos incluyen desde un programa mínimo con solo program y main, hasta declaraciones de variables, impresiones, operaciones aritméticas, estructuras de control de flujo y funciones. También se incluyó un caso con un error intencional de sintaxis (una instrucción sin punto y coma) para comprobar la detección de errores. Estos archivos se guardaron con la extensión .web y se procesaron mediante un programa Java que instanciaba el lexer y parser, y aplicaba un listener personalizado para detectar errores sintácticos de forma clara y confiable.

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.

```
== Parsing file: test01.web ==
(programa program myProgram ; main (body { }) end)
? Sintaxis correcta.
```

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.

```
== Parsing file: test02.web ==
(programa program myProgram ; (vars var (var_decl (id_list a , b , c) : (type int) ;) (var_decl (id_list x , y) : (type float) ;)) main (body { }) end)
? Sintaxis correcta.
```

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.

```
== Parsing file: test03.web ==
(programa program testFuncs ; (vars var (var_decl (id_list a) : (type int) ;)) (funcs_list (funcs void suma ( (params a : (type int) , b : (type int)) ) [ (vars var (var_decl (id_list res) : (type int) ;)) (body { (statement (assign res = (expresion (expr (termino (factor a)) + (termino (factor b)))) ;)) } ) ] ;) (funcs void mensaje ( ) [ (body { (statement (print print ( (print_args (print_arg "Hola mundo") ) ;)) } ) ] ;) ) main (body { }) end)
? Sintaxis correcta.
```

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.

```
== Parsing file: test04.web ==
line 3:8 token recognition error at: '~'
? Error de sintaxis en test04.web en la línea 3, columna 10: mismatched input '3' expecting {'+', '-', '*', '/', '!=', '<', '>', ';' }
(programa program exprProg ; main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5)))) 3 * ( 2 - 1 ) ;)
) }) end)
? Sintaxis incorrecta.
```

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.

```
== Parsing file: test05.web ==
? Error de sintaxis en test05.web en la línea 6, columna 0: mismatched input '}' expecting 'else'
(programa program conditionTest ; main (body { (statement (condition if ( (expresion (exp (termino (factor a))) < (exp (termino (factor b)))) ) (body { (statement (print print ( (print_args (print_arg "menor")) ) ;)) })) } end)
? Sintaxis incorrecta.
```

BabyDuck – Entrega #2

Tablas de Consideraciones Semánticas

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

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

2. Operaciones relacionales (< , > , !=)

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

3. Asignación (=)

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

DirFunc y VarTable

Para gestionar las funciones del lenguaje y sus respectivos ámbitos de variables, se diseñó la clase DirFunc. Esta clase utiliza como estructura principal un `HashMap<String, VarTable>` llamado `functions`, donde cada llave representa el nombre de una función (incluyendo el ámbito global) y el valor es una instancia de `VarTable`, que contiene las variables locales a dicha función. Esta estructura permite realizar operaciones como búsquedas, inserciones y validaciones de manera eficiente, en tiempo constante promedio ($O(1)$), lo cual es ideal para un lenguaje donde las funciones pueden ser accedidas frecuentemente por nombre.

Además de la tabla de funciones, DirFunc mantiene dos atributos auxiliares: globalScopeName, que almacena el nombre del ámbito global (típicamente el nombre del programa), y currentFunction, que indica en qué función se encuentra actualmente el compilador o el analizador semántico. Esto permite que las operaciones sobre variables, como inserciones o validaciones, se realicen directamente sobre el contexto correcto sin necesidad de pasarlo explícitamente en cada operación.

Las operaciones principales de esta clase incluyen la creación de nuevas funciones (addFunction), el cambio de contexto (setCurrentFunction), la inserción de variables en la función actual (addVariable), y la validación de existencia de variables tanto en el ámbito local como global. Esta organización permite mantener una jerarquía clara y un control estricto sobre los ámbitos, además de facilitar la verificación semántica en múltiples fases del compilador.

Por otra parte, la clase VarTable se encarga de representar el conjunto de variables declaradas dentro del ámbito de una función o del ámbito global. Internamente utiliza un `HashMap<String, VarInfo>` llamado variables, donde cada llave es el nombre de una variable y su valor es una instancia de VarInfo, clase interna que encapsula tanto el tipo (VarType) como el valor actual de la variable. Esta estructura permite realizar búsquedas, inserciones y actualizaciones de forma eficiente, en tiempo constante promedio, ideal para entornos donde se espera acceder frecuentemente a las variables por nombre.

El uso de la clase auxiliar VarInfo proporciona flexibilidad y escalabilidad, ya que permite almacenar más información relacionada a la variable en el futuro (por ejemplo, dirección de memoria, tipo de acceso, etc.), sin modificar la lógica principal de la tabla. Entre las operaciones principales de VarTable se encuentran: agregar una nueva variable (addVariable), obtener su tipo (getVariableType), leer o modificar su valor (getVariableValue y setVariableValue), y verificar su existencia (hasVariable). Todas estas funciones están diseñadas para asegurar la consistencia del ámbito, evitando redefiniciones y permitiendo un control estricto sobre las declaraciones y usos válidos de variables en el lenguaje.

En conjunto con DirFunc, esta clase permite modelar de manera clara y eficiente los diferentes niveles de ámbito que existen en un lenguaje con funciones, diferenciando entre variables locales y globales, y facilitando la implementación de reglas semánticas como la detección de dobles declaraciones o el uso de variables no definidas.

Puntos neurálgicos y Validaciones

Para establecer ahora si las funciones previas al revisar el código se uso un validador semántico de ANTLR. El SemanticVisitor es una clase que extiende de WebbyParserBaseVisitor<Void> y se encarga de recorrer el árbol sintáctico generado por ANTLR para realizar las validaciones semánticas necesarias del lenguaje. Su función principal es verificar que el código fuente cumpla con las reglas semánticas definidas para variables, funciones y estructuras de control, además de registrar la información necesaria para la ejecución o traducción del programa. Para ello, utiliza una instancia de DirFunc, la cual encapsula toda la información de ámbito y declaración de variables y funciones, separando claramente el contexto global del contexto local de cada función.

Durante el recorrido del árbol, el SemanticVisitor primero registra el nombre del programa como función principal y crea una nueva instancia del directorio de funciones. Después, visita las declaraciones de variables globales, funciones y finalmente el bloque principal del programa. Una de las primeras validaciones importantes es asegurar que no haya variables duplicadas dentro del mismo ámbito (local o global). En el caso de funciones, también se verifica que no existan funciones duplicadas. Cuando se detecta una redefinición de variable o función, se lanza una excepción con un mensaje claro de error.

En cuanto a las funciones, SemanticVisitor revisa que todos los parámetros estén correctamente definidos y que no se repitan. Al registrar parámetros o variables locales, se realiza una verificación del tipo de dato utilizando el enumerador VarType, y se almacenan en la tabla de variables del ámbito correspondiente. Esto asegura que cada variable esté correctamente tipada y declarada antes de ser utilizada.

Dentro de las expresiones, como las asignaciones o factores (por ejemplo, variables usadas en cálculos), se valida que todas las variables hayan sido declaradas previamente, ya sea en el ámbito local o global. En caso contrario, se lanza un error señalando el uso indebido de una variable no declarada. Este control permite prevenir errores de ejecución y mejora la robustez del lenguaje.

Pruebas Realizadas

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.

```
== Parsing file: test01.web ==
(programa program programDuplicado ; (funcs_list (funcs void hola ( ) [ (body { (statement (print print ( (print_args (print_arg "Hola mundo")) ) ;)) }) ] ;) (funcs void hola ( ) [ (body { (statement (print print ( (print_args (print_arg "Otra versión"
)) ) ;)) }) ] ;)) main (body { }) end)
? Sintaxis correcta.

? Error parsing test01.web: Error: Función 'hola' ya fue declarada.
```

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.

```
== Parsing file: test02.web ==
(programa program varDuplicada ; (vars var (var_decl (id_list a) : (type int) ;) (var_decl (id_list a) : (type float) ;)) main
(body { }) end)
? Sintaxis correcta.

? Error parsing test02.web: 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.

```
== Parsing file: test03.web ==
(programa program sinDeclarar ; main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5))))))) }) end)
? Sintaxis correcta.

? Error parsing test03.web: Error: 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.

BabyDuck – Entrega #3

Estructuras para Traducción a Cuádruplos

Clase Quadruple

La clase Quadruple representa la estructura base utilizada para traducir instrucciones del lenguaje BabyDuck a una forma intermedia conocida como cuádruplos. Cada cuádruplo consiste en cuatro elementos: una operación (op), dos operandos (arg1, arg2) y un resultado (res). Esta representación simplifica la generación de código intermedio al permitir una forma uniforme y flexible para describir operaciones aritméticas, asignaciones,

condiciones, impresiones y más. Esta clase también sobrescribe el método `toString()` para permitir una impresión legible del cuádruplo, útil para pruebas e inspección manual.

Clase **SemanticStackContext**

Durante la generación de cuádruplos, es necesario manejar múltiples pilas de trabajo para mantener el estado semántico de manera estructurada. La clase **SemanticStackContext** encapsula las siguientes pilas:

- **operandStack**: almacena los operandos (nombres de variables o constantes) usados en las expresiones.
- **operatorStack**: guarda los operadores aritméticos o relacionales en espera de evaluación.
- **typeStack**: lleva el seguimiento del tipo de cada operando (por ejemplo, INT, FLOAT) para realizar validaciones semánticas usando las tablas de consideraciones.

El uso de estas pilas permite transformar expresiones infijas en cuádruplos, generando instrucciones intermedias paso a paso conforme se recorren los nodos del árbol de análisis sintáctico.

Cambios al **SemanticVisitor** para Generación de Cuádruplos

Además del análisis semántico previamente implementado, el **SemanticVisitor** fue extendido para traducir expresiones del lenguaje BabyDuck a cuádruplos.

Primeramente, se introdujeron variables nuevas donde se destacan:

- **quadruples**: lista que acumula todos los cuádruplos generados hasta el momento.
- **tempVarCounter**: contador para nombrar variables temporales generadas durante la evaluación de expresiones complejas.

Además, se instancia la clase **SemanticStackContext** para tener nuestras pilas listas para utilizar.

Posterior a eso se hicieron cambios en los métodos que conforman este. Este proceso ocurre dentro de los métodos relacionados con:

- Asignaciones (`visitAssign`)
- Factores y términos (`visitFactor`, `visitTermino`, `visitExp`)
- Prints (`visitPrint_arg`)

Cada vez que se detecta una operación aritmética o una asignación, se realiza lo siguiente:

- Se colocan operandos y operadores en sus respectivas pilas.

- Al finalizar la subexpresión, se "resuelve" la operación, generando un cuádruplo.
- Se asigna una variable temporal (por ejemplo, t0, t1, etc.) para guardar el resultado.
- Este resultado se vuelve a colocar en la pila de operandos para su uso en operaciones futuras.

Finalmente, en los cuadruplos también se incluyen los prints de el resultado final que se manda a imprimir (o por consiguiente el letrero).

El resultado final es una lista ordenada de cuádruplos que puede utilizarse para interpretación directa o generación de código más adelante. La estructura generada es clara, lineal y respeta las reglas de precedencia de operadores del lenguaje.

Pruebas Realizadas

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.

```
*** Parsing file: test01.web ***
(programa program asignacionCompleja ; (vars var (var_decl (id_list a , b , c , d , resultado) : (type int) ;)) main (body { (statement (assign a = (exp (termino (factor (cte 5)))) ;))
(statement (assign b = (exp (termino (factor (cte 10)))) ;))) (statement (assign c = (exp (termino (factor (cte 2)))) ;))) (statement (assign d = (exp (termino (factor (cte 3)))) ;))) (statement (assign resultado = (exp (exp (termino (factor a)) + (termino (factor b) * (factor c)) - (termino (factor (cte 1)))))))) ;)) end
? Parsed successfully.

Cuádruplos generados:
(=, 5, _, a)
(=, 10, _, b)
(=, 2, _, c)
(=, 3, _, d)
(+, b, c, t0)
(+, a, t0, t1)
(-, a, 1, t2)
(/, d, t2, t3)
(-, t1, t3, t4)
(=, t4, _, resultado)
```

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.

```
*** Parsing file: test02.web ***
(programa program printComplejo ; (vars var (var_decl {id_list x , y , z} : (type int) )); main (body { (statement (assign x = (expresion (exp (termino (factor (cte 4))))));) (statement (assign y = (expresion (exp (termino (factor (cte 8))))));) (statement (assign z = (expresion (exp (termino (factor (cte 2))))));) (statement (print print ( (print_args (print_arg "Resultado: ") , (print_arg (expresion (exp (termino (factor x) * (factor y)) + (termino (factor z)) / (factor ( (expresion (exp (termino (factor x)) - (termino (factor (cte 2))))))))))) ) ;)) } end)
? Parsed successfully.

Cuádruplos generados:
(=, 4, _, x)
(=, 8, _, y)
(=, 2, _, z)
(PRINT, _, _, "Resultado: ")
(*, x, y, t0)
(-, x, 2, t1)
(/, z, t1, t2)
(+, t0, t2, t3)
(PRINT, _, _, t3)
```

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.

```
*** Parsing file: test03.web ***
(programa program ifComplejo ; (vars var (var_decl {id_list m , n , p} : (type int) )); main (body { (statement (assign m = (expresion (exp (termino (factor (cte 6))))));) (statement (assign n = (expresion (exp (termino (factor (cte 3))))));) (statement (assign p = (expresion (exp (termino (factor (cte 10))))));) (statement (condition if ( (expresion (exp (termino (factor ( (expresion (exp (termino (factor m) * (factor n)) + (termino (factor (cte 4)))))))))) > (exp (termino (factor ( (expresion (exp (termino (factor p)) - (termino (factor n) / (factor (cte 2)))))))))) ) (body { (statement (print print ( (print_args (print_arg (expresion (exp (termino (factor m)) + (termino (factor n)) + (termino (factor p))))))) ;)) } ) else (body { (statement (print print ( (print_args (print_arg "No cumple"))))) ;)) } ) end)
? Parsed successfully.

Cuádruplos generados:
(=, 6, _, m)
(=, 3, _, n)
(=, 10, _, p)
(*, m, n, t0)
(+, t0, 4, t1)
(/, n, 2, t2)
(-, p, t2, t3)
(>, t1, t3, t4)
(+, m, n, t5)
(+, t5, p, t6)
(PRINT, _, _, t6)
(PRINT, _, _, "No cumple")
```

BabyDuck – Entrega #4

Cambios de Código y Cubo Semántico

Durante esta etapa del desarrollo del compilador se realizaron varios ajustes en el sistema de tipos, el cubo semántico y la representación de cuádruplos con el objetivo de habilitar soporte para condiciones, ciclos y evaluación más robusta de expresiones. En primer lugar, se amplió la enumeración **VarType** para incluir nuevos tipos de datos. Originalmente, únicamente estaban definidos los tipos INT y FLOAT; sin embargo, fue necesario agregar BOOL y STRING para dar soporte semántico completo a expresiones relacionales y mensajes de salida, respectivamente.

Posteriormente, se realizaron modificaciones clave en la clase **CuboSemantico**. Esta estructura, que define las reglas de compatibilidad entre tipos de datos para diversas operaciones, fue enriquecida con nuevas combinaciones para operadores relacionales (<, >, !=) y el operador de asignación (=). Se agregaron entradas que permiten validaciones correctas cuando se realizan comparaciones entre enteros y flotantes, y se especificaron los resultados como valores booleanos. Además, se definieron reglas para la asignación, permitiendo únicamente asignaciones entre tipos idénticos y marcando como inválidas aquellas que implican asignación cruzada entre INT y FLOAT. Estas reglas son fundamentales para evitar errores de tipo durante la ejecución de programas.

Por otro lado, se incorporaron cambios en la clase **Quadruple**, con el fin de facilitar la representación de instrucciones de control de flujo como los condicionales y los ciclos. Se añadieron los operadores GOTO y GOTOF al mapa de códigos de operación (**OPERATOR_CODES**), los cuales son indispensables para generar saltos incondicionales y condicionales durante la ejecución del código intermedio. A su vez, se mejoró el método `toMemoryString`, que transforma los cuádruplos en su representación final usando direcciones de memoria. Una modificación importante en este método fue la inclusión de una verificación para detectar si un operando inicia con el símbolo `#`, indicando así que se trata de un valor literal que no debe buscarse en memoria, sino pasarse tal cual. Esto resuelve posibles ambigüedades con literales numéricos que también pueden existir como constantes registradas.

En conjunto, estos cambios establecen una base sólida para la correcta verificación semántica de expresiones, así como para la generación de cuádruplos que se alinean con una máquina virtual basada en direcciones de memoria, permitiendo la implementación de estructuras de control como `if` y `while`.

MemoryManager

La clase `MemoryManager` fue diseñada y modificada para encargarse de la asignación de direcciones de memoria a todas las variables, constantes y temporales del programa, según su tipo y ámbito (global, local o temporal). Esta arquitectura es crucial para la correcta ejecución del programa intermedio generado, ya que permite mapear nombres simbólicos a direcciones de memoria concretas que pueden ser interpretadas por la máquina virtual.

Justificación de los Cambios

Anteriormente, la asignación de memoria era un proceso implícito o no estaba completamente delimitado por segmentos. Esta versión implementa una gestión segmentada y ordenada de los espacios de memoria, lo cual mejora la claridad, escalabilidad y depuración del compilador.

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

Esta separación permite que las direcciones no colisionen entre segmentos distintos y facilita su interpretación durante la ejecución del programa.

Funcionalidad Clave

- **assignGlobal y assignLocal:** Asignan direcciones de memoria a variables declaradas en el ámbito global o dentro de funciones.
- **assignTemp:** Asigna direcciones a valores intermedios generados durante la ejecución de operaciones.
- **assignConst:** Registra valores constantes, asegurando que cada uno se almacene una sola vez.
- **getAddress:** Recupera la dirección de cualquier identificador según su prioridad: temporales, locales, globales y constantes.
- **resetLocalAndTemp:** Se llama al iniciar una nueva función para reiniciar los contadores locales y temporales, permitiendo reutilización segura de direcciones.

Consideraciones Adicionales

Además de soportar los tipos INT y FLOAT, esta nueva versión ya contempla tipos booleanos (BOOL) y cadenas (STRING) para temporales y constantes respectivamente, en preparación para futuras funcionalidades como condicionales, ciclos y manejo de strings.

Cuádruplos para Condicionales y Ciclos

Una parte crucial de la generación de código intermedio es la correcta gestión del flujo de control, lo cual se logra mediante los cuádruplos GOTO y GOTOF. Estos cuádruplos permiten implementar condicionales (if/else) y ciclos (while) mediante saltos incondicionales o condicionales a otras instrucciones del programa.

Condicionales (if/else)

El manejo de un if con else en cuádruplos se divide en varias etapas:

1. **Evaluación de la condición:** Se visita la expresión condicional y se obtiene su resultado lógico (una dirección booleana en memoria).
2. **Inserción del GOTOF:** Se genera un cuádruplo GOTOF <condición> -1, que significa "si la condición es falsa, salta a la instrucción después del if". El -1 es un marcador temporal, ya que aún no se sabe la dirección exacta del salto.
3. **Generación del bloque if (body verdadero):** Se visitan las instrucciones que corresponden al bloque que se ejecuta si la condición es verdadera.
4. **Creación del GOTO para saltar el else:** Tras ejecutar el bloque if, se genera un GOTO -1 para saltar el bloque else.
5. **Actualización del GOTOF con la dirección del bloque else:** Una vez conocido el inicio del bloque else, se actualiza el GOTOF para que apunte a esa dirección.

6. Generación del bloque else: Se generan los cuádruplos para el bloque alternativo si la condición fue falsa.
7. Actualización final del GOTO: El segundo GOTO (al final del if) se actualiza para apuntar al final del bloque else.

Este esquema permite que solo uno de los bloques (if o else) se ejecute, dependiendo del valor de la condición.

Ciclos (while)

Para los ciclos, el flujo es similar, pero con un salto hacia atrás:

1. **Guardar el inicio del ciclo:** Se guarda el índice del cuádruplo actual antes de evaluar la condición. Este índice es el punto de regreso para repetir el ciclo.
2. **Evaluar la condición:** Se visita la expresión booleana que controla el ciclo.
3. **Generar el GOTOF:** Si la condición es falsa, se genera un GOTOF con salto pendiente hacia fuera del ciclo.
4. **Visitar el cuerpo del ciclo:** Se generan los cuádruplos correspondientes a las instrucciones dentro del ciclo.
5. **Generar un GOTO de regreso al inicio:** Una vez terminado el cuerpo, se genera un GOTO que apunta al índice guardado al principio, cerrando así el ciclo.
6. **Actualizar el GOTOF con la salida del ciclo:** Finalmente, se actualiza el GOTOF con la dirección del cuádruplo después del cuerpo, para romper el ciclo si la condición se evalúa como falsa.

Justificación

El uso de GOTO y GOTOF permite representar cualquier flujo de control de alto nivel usando únicamente instrucciones de salto. Esta técnica simplifica el análisis posterior en la máquina virtual y permite una implementación más directa de estructuras como if, else, while, e incluso for en futuras extensiones.

Se generan con marcadores temporales (#-1) porque al momento de crear el cuádruplo no se conoce todavía a dónde debe saltar. Solo después de procesar los bloques correspondientes se puede determinar el destino real del salto, y es entonces cuando se actualiza el cuádruplo con la dirección correcta.

Cuádruplos con Direcciones de Memoria

Una vez que los cuádruplos simbólicos están generados (usando nombres de variables o constantes), se requiere traducirlos a una forma que utilice direcciones de memoria reales. Esto es fundamental para la ejecución en una máquina virtual.

Para esto, cada cuádruplo tiene un método `toMemoryString(...)`, que hace lo siguiente:

1. Traduce el operador a un código numérico, mediante `getOperatorCode`, para facilitar su interpretación por la máquina virtual.
2. Resuelve las direcciones de memoria de los operandos y el resultado, usando `resolveOperandAddress`.
 - a. Si el valor empieza con #, es un salto relativo o constante directa y se toma como está.
 - b. Si es un número o identificador, se busca su dirección de memoria correspondiente en el `MemoryManager`, dependiendo del ámbito actual.
 - c. Si no se encuentra o es inválido, se deja como está.

Pruebas Realizadas

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 `GOTOIF` 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.

```
== Parsing file: test04.web ==
(programa program ifSimple ; (vars var (var_decl (id_list x) : (type int) ;)) main (body { (statement (assign x = (exp (termino (factor (cte 10)))))) ;}) (statement (condition if ( (expresion (exp (termino (factor x))) > (exp (termino (factor (cte 5)))))) ) (body { (statement (print print ( (print_args (print_arg (expresion (exp (termino (factor x)))))))) ) ;})) } else (body { (statement (print print ( (print_args (print_arg (expresion (exp (termino (factor (cte 0)))))))) ) ;))) } ;)) end
? Sintaxis correcta.

Cuádruplos generados:
(=, 10, _, x)
(>, x, 5, t0)
(GOTOIF, t0, , #5)
(PRINT, _, _, x)
(GOTO, _, _, #6)
(PRINT, _, _, 0)
Cuádruplos usando memoria:
(5, 8000, _, 1000)
(8, 1000, 8001, 7000)
(11, 7000, _, 5)
(9, _, _, 1000)
(10, _, _, 6)
(9, _, _, 8002)
```

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 `GOTOIF` 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.

```

== Parsing file: test05.web ==
(programa program whileSimple ; (vars var (var_decl (id_list i) : (type int) ;)) main (body { (statement (assign i = (expresion (exp (termino (factor r (cte 0))))))) ;)) (statement (cycle while ( (expresion (exp (termino (factor i))) < (exp (termino (factor (cte 3)))))) ) do (body { (statement (print
print ( (print_args (print_arg (expresion (exp (termino (factor i))))))) ) ;)) (statement (assign i = (expresion (exp (termino (factor i)) + (termino
o (factor (cte 1))))))) ;)) } ;)) }) end
? Sintaxis correcta.

Cuádruplos generados:
(=, 0, _, i)
(<, i, 3, t0)
(GOTO, t0, , #7)
(PRINT, _, _, i)
(+, i, 1, t1)
(=, t1, _, i)
(GOTO, , , #1)
Cuádruplos usando memoria:
(5, 8000, _, 1000)
(7, 1000, 8001, 7000)
(11, 7000, , 7)
(9, _, _, 1000)
(1, 1000, 8002, 5000)
(5, 5000, _, 1000)
(10, , , 1)

```

BabyDuck – Entrega #5

Virtual Machine

La clase VirtualMachine representa el componente encargado de ejecutar los cuádruplos generados durante el análisis semántico y la traducción intermedia del programa. Su principal función es simular la ejecución del programa fuente, interpretando cada cuádruplo como una instrucción de bajo nivel. Esta clase conecta directamente con la tabla de funciones (DirFunc), la tabla de constantes (ConstTable) y la clase VirtualMemory, que gestiona el estado de la memoria durante la ejecución.

El constructor de VirtualMachine recibe la lista de cuádruplos, la tabla de funciones y la tabla de constantes. A partir de esta información, inicializa la memoria virtual y la pila de llamadas. También se define un contador de parámetros y un apuntador de instrucción (instructionPointer) que controla el flujo de ejecución del programa.

Durante la ejecución, el método execute recorre la lista de cuádruplos uno por uno, interpretando el operatorCode de cada cuádruplo para realizar la acción correspondiente. Por ejemplo, las operaciones aritméticas básicas como suma, resta, multiplicación y división se manejan en el método executeArithmetic, que evalúa los operandos y almacena el resultado en la dirección correspondiente.

La máquina virtual también maneja instrucciones de control de flujo como GOTO, GOTOIF y ENDPORG. Estas instrucciones permiten saltar a diferentes partes del código o finalizar la ejecución. En el caso de GOTOIF, se evalúa una condición booleana y se realiza el salto solo si esta es falsa.

El soporte para funciones se implementa con instrucciones como ERA, PARAM, GOSUB y ENDFUNC. La instrucción ERA inicializa un nuevo marco local para la función que está a punto de llamarse. PARAM transfiere los argumentos al marco pendiente, y GOSUB realiza el salto a la función, guardando la dirección de retorno en una pila. Una vez que la función termina, ENDFUNC limpia el marco local y regresa la ejecución al punto de llamada original.

Las operaciones de comparación, como desigualdad ($!=$), menor que ($<$) y mayor que ($>$), se evalúan con ayuda del método `compareValues`, que asegura la compatibilidad de tipos entre operandos. También se maneja la impresión con la instrucción `PRINT`, que imprime en consola el valor ubicado en la dirección especificada.

En general, `VirtualMachine` permite ejecutar de manera completa y controlada un programa compilado en cuádruplos, soportando expresiones, asignaciones, condiciones, funciones y operaciones aritméticas. Esta clase es crucial para validar que la lógica del programa fuente se ejecuta correctamente según la semántica definida durante la compilación.

VirtualMemory

La clase `VirtualMemory` implementa la abstracción principal para manejar la memoria virtual en tiempo de ejecución dentro del modelo de Máquina Virtual. Se encarga de gestionar tanto los segmentos globales como los locales, temporales y constantes, además de soportar múltiples contextos de ejecución mediante una pila de marcos de memoria.

Estructura de almacenamiento

- Los valores globales y constantes se almacenan en `ArrayLists` propios dentro de la clase.
- Las variables locales y temporales se manejan mediante objetos `MemoryFrame`, que representan un contexto de ejecución (por ejemplo, una llamada a función).
- Los `MemoryFrame` se almacenan en una pila (`memoryFrames`) para soportar el paso de contextos, y se gestiona un `pendingFrame` para preparar el contexto antes de una llamada.

Funcionalidad principal

- **`getValue(int address)` / `setValue(int address, Object value)`:** Acceden o modifican valores de memoria según la dirección virtual, delegando a la estructura correspondiente (global, frame actual o constante).
- **`pushFrame(MemoryFrame)` / `popFrame()`:** Manipulan el stack de contextos de ejecución.
- **`loadConstants(ConstTable)`:** Carga los valores constantes definidos por el compilador al inicio de la ejecución.
- **`loadGlobalVariables(Map<String, VarInfo>)`:** Inicializa las estructuras globales y temporales del main de acuerdo con las direcciones proporcionadas.
- **`loadLocalTempVariables(Map<String, VarInfo>)`:** Prepara un nuevo `MemoryFrame` con tamaños adecuados para variables locales y temporales.

- **commitPendingFrame()**: Activa el frame preparado anteriormente como el nuevo contexto de ejecución.

Validaciones

El sistema lanza excepciones si se intenta acceder a un segmento no válido, si no hay un frame activo o pendiente en contexto, o si se intenta modificar un valor en un segmento constante.

Propósito

Esta clase abstrae toda la lógica de administración de memoria para que el componente ejecutor de cuádruplos (la CPU de la Máquina Virtual) pueda operar simplemente utilizando direcciones virtuales, sin preocuparse por los detalles internos de segmentación o contexto.

MemoryFrame

La clase MemoryFrame representa un marco de memoria dinámica que se utiliza para almacenar valores de variables locales y temporales durante la ejecución de funciones en la Máquina Virtual. Esencialmente, actúa como una pila de activación que contiene los valores que necesita una función activa en un momento dado.

Cada MemoryFrame gestiona cinco segmentos distintos de memoria mediante ArrayList que son básicamente las variables que son locales y las temporales

Se inicializa cada segmento con el tamaño recibido, reservando espacio con valores null para cada posición. Esto permite que la VM realice asignaciones y consultas de forma segura sin errores de índice fuera de rango.

Se establece también un valor en la dirección especificada. El método determina el segmento correcto en función del rango de direcciones y convierte el valor al tipo correspondiente. Si la dirección es inválida, lanza una excepción. Este método garantiza también que la lista esté lo suficientemente grande como para aceptar el valor, expandiéndola dinámicamente si es necesario.

Métodos auxiliares

- **ensureSize**: Asegura que una lista tenga al menos la longitud necesaria para escribir un valor en un índice específico.
- **resize**: Inicializa una lista con una cantidad determinada de elementos null.

Uso

Esta clase es útil para crear nuevos marcos de memoria al invocar funciones, permitiendo encapsular el estado local y temporal. Cada vez que se llama a una función, se instancia un nuevo MemoryFrame, que se apila y desapila en la VM conforme cambian los contextos de ejecución.

Pruebas Realizadas

Para validar el correcto funcionamiento de la Máquina Virtual y la clase MemoryFrame, se diseñó una serie de pruebas agrupadas en la carpeta fact_fib, que contiene cuatro archivos de entrada escritos en el lenguaje Webby. Cada archivo representa un caso distinto de ejecución que involucra el manejo de variables locales, temporales, llamadas a funciones y retorno de resultados.

fact.web – Factorial en main

Este archivo implementa el cálculo del factorial directamente en la función principal (main), utilizando una estructura de control cíclica (while). Se verifica que la Máquina Virtual genera y ejecuta correctamente los cuádruplos asociados a la iteración, realiza multiplicaciones sucesivas y almacena el resultado en memoria local. Se aplico concretamente el ciclo mientras $i < 6$, por lo que hizo $1 \times 2 \times 3 \times 4 \times 5$ que es 120.

```
==== Ejecutando programa ====
Factorial desde main:
120
==== Ejecución terminada ====

```

fact_func.web – Factorial con función

En esta versión, el cálculo del factorial se realiza dentro de una función definida por el usuario, la cual es invocada desde main. Se prueba el manejo correcto de un nuevo MemoryFrame al momento de llamar la función, así como la gestión adecuada de los parámetros, variables locales y el retorno del resultado hacia el contexto original. Se combinó el uso variables globales, llamada a la función, y se llevo con $n = 5$, es decir haciendo $1 \times 2 \times 3 \times 4$

```
==== Ejecutando programa ====
Factorial desde función:
24
==== Ejecución terminada ====

```

fib.web – Fibonacci en main

Este archivo calcula la sucesión de Fibonacci desde la función principal, utilizando variables auxiliares y ciclos. Se valida que el manejo de direcciones temporales en MemoryFrame sea correcto y que los valores intermedios no sobrescriban información crítica. La secuencia generada por la Máquina Virtual es consistente con los valores esperados.

```
==== Ejecutando programa ====
0
1
1
2
3
5
==== Ejecución terminada ====

```

fib_func.web – Fibonacci con función

La versión funcional del cálculo de Fibonacci pone a prueba el sistema de llamadas anidadas a funciones. Se comprueba que múltiples MemoryFrame puedan coexistir de

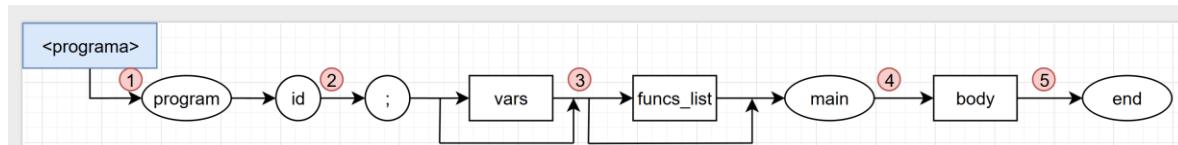
forma aislada en la pila de ejecución de la Máquina Virtual y que los resultados retornados sean utilizados apropiadamente en el contexto que hizo la llamada.

```
==== Ejecutando programa ====
0
1
1
2
3
5
==== Ejecución terminada ====

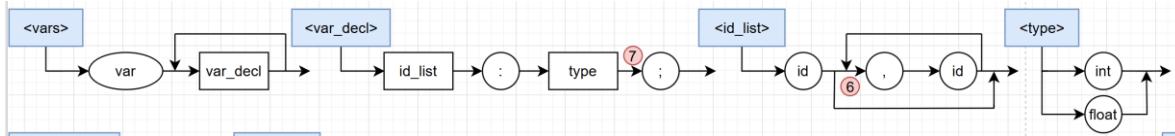
```

Para ver con más detalle se puede ver en el repositorio y correr el Main con las instrucciones especificadas al final de esta documentación.

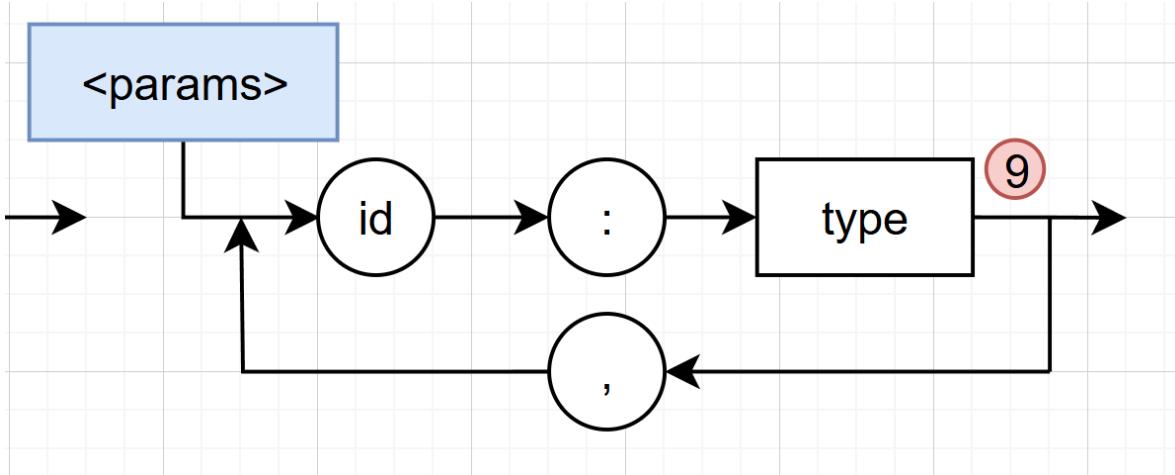
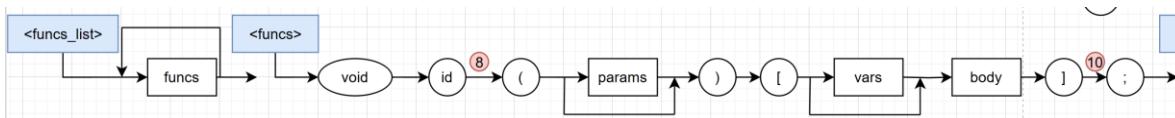
Diagramas y Puntos neurálgicos



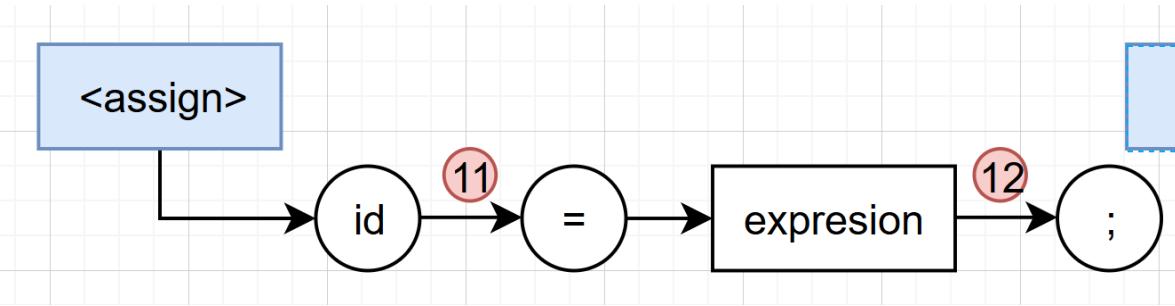
1. Creación de List<Quadruples> quadruples, MemoryManager memory, ConstTable constTable, SemanticStackContext stackContext, int tempCounter
2. Creación de DirFunc dirFunc, dirFunc.globalDataName = id, dirFunc.currentFunction = id, creación de HashMap Functions en dirFunc e insertar id con nueva FunctionInfo
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 cuádruplo ENDPORG



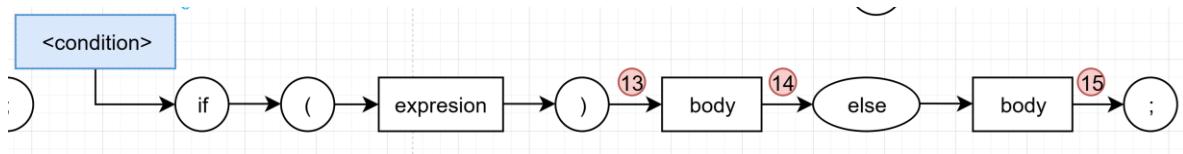
6. Si dirFunc.variableExistsInCurrentScope(id) lanzar error de variable doblemente declarada.
7. varType = type, iterar por cada id de id_list, definir address, Si dirFunc.globalIsCurrent() entonces address = memoria.assignGlobalAddress(varType) sino address = memoria.assignLocalAddress(varType), finalmente dirFunc.addVariable(id, varType, address)



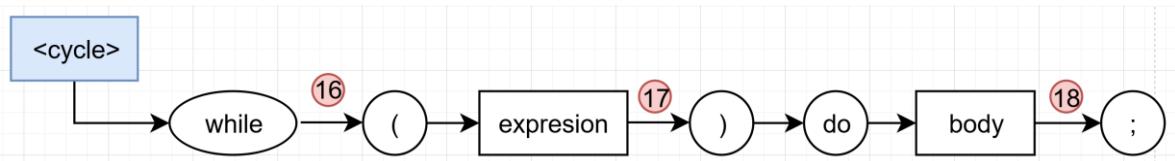
8. funcName = id, Si !dirFunc.addFunction(funcName) lanzar error de función doblemente declarada, dirFunc.setCurrentFunction(funcName), dirFunc.setFunctionStartQuad(funcName, quadruples.size())
9. Si dirFunc.variableExistsInCurrentScope(id) lanzar error de variable doblemente declarada, sino address = memoria.assignLocalAddress(varType) y dirFunc.addParameter(id, varType, address)
10. Resetear contadores para asignar dirección de memoria con memoria.resetLocalAndTemp() por si viene otra función, insertar cuádruplo de ENDFUNC



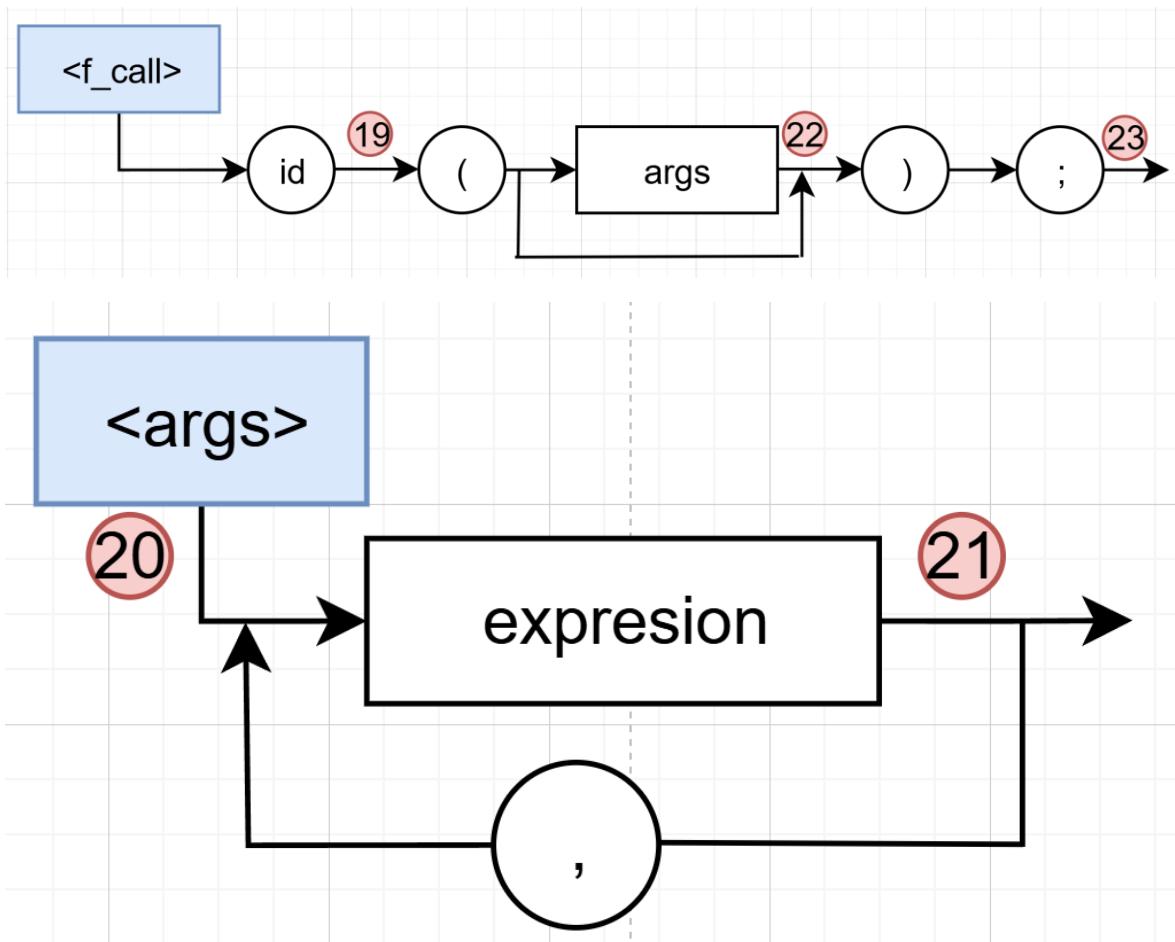
11. Si !dirFunc.variableExists(id) lanzar error de uso de variable no declarada
12. Obtener varType = dirFunc.getVariableType(id), obtener exprType = stackContext.popType() y exprValue = stackContext.popOperand(), si cuboSemantico.getResultType("=", varType, exprType) es null entonces lanzar error de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



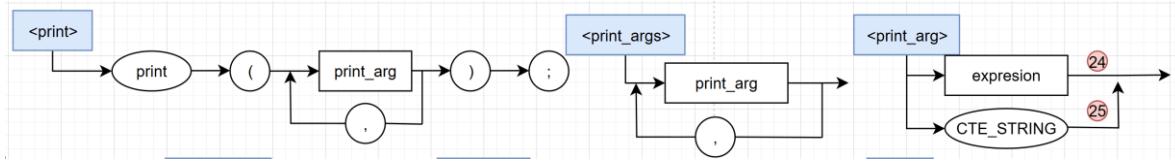
13. Obtener variable/temporal conditionResult = visit(ctx.expresion()), generar cuádruplo GOTOF de conditionResult y guardar índice de cuádruplo en gotofIndex
14. Crear GOTO para hacer salto si es que se entro al body verdadero del if y guardar el índice del cuádruplo en gotoEndIndex, modificar cuádruplo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



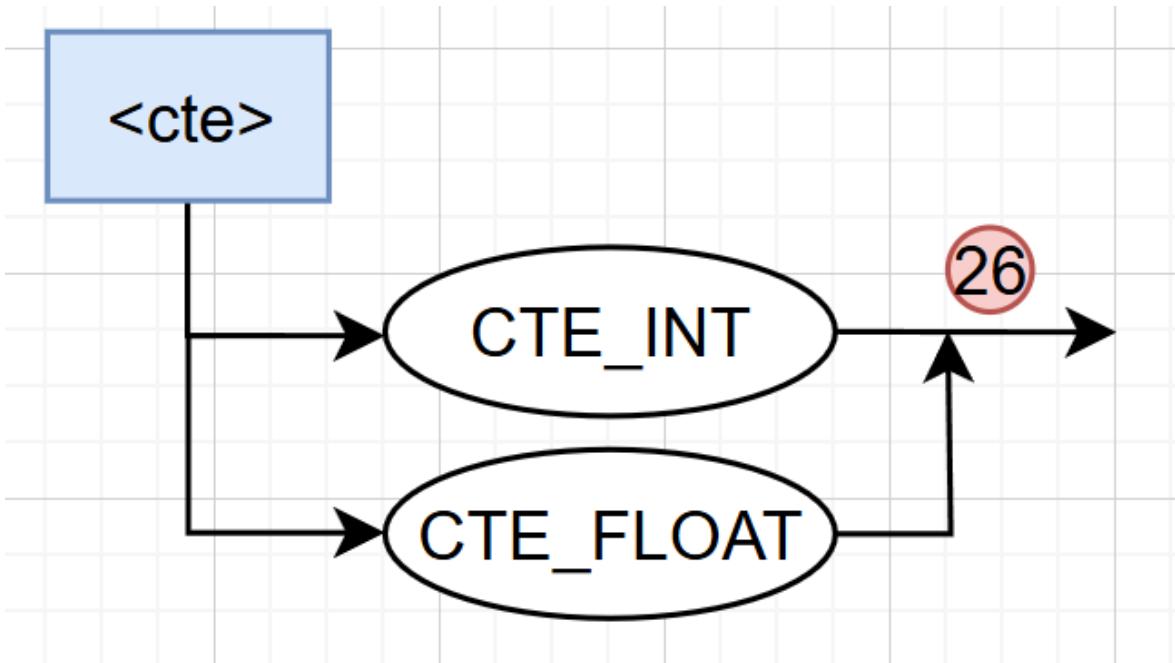
16. Guardar en loopStart = quadruples.size()
17. Generar cuádruplo GOTOF y guardar índice de cuádruplo en gotofIndex
18. Generar cuádruplo GOTO e indicar salto en loopStart y modificar cuádruplo GOTOF con quadruples.size()



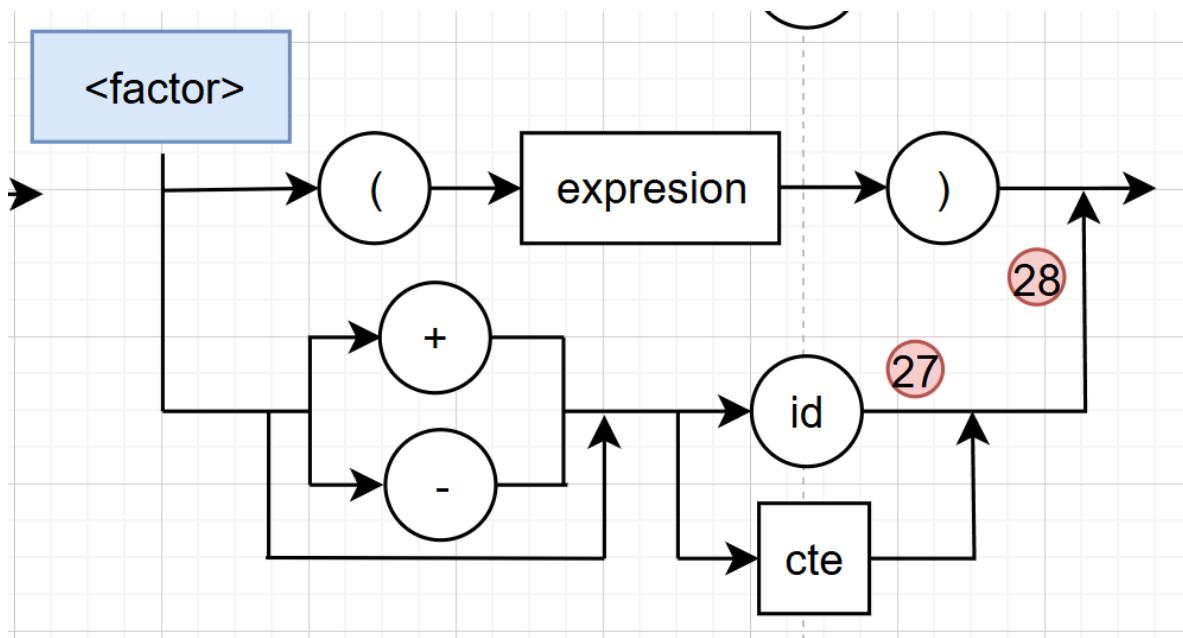
19. Si !dirFunc.hasFunction(funcName) lanzar error de llamado de función no existente, sino generar cuádruplo ERA con funcName
20. Se vacían argumentQueue y argumentTypeQueue
21. Se hacen argAddr = stackContext.popOperand() y argType = stackContext.popType() y se agregan argumentQueue.add(argAddr) y argumentTypeQueue.add(argType)
22. Si argumentQueue.size() != expectedParams.size() se lanza error de número de argumentos en la función no son los esperados, 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, se agrega cuádruplo PARAM
23. Se obtiene startQuad = dirFunc.getFunctionStartQuad(funcName) y se genera cuádruplo GOSUB con el startQuad



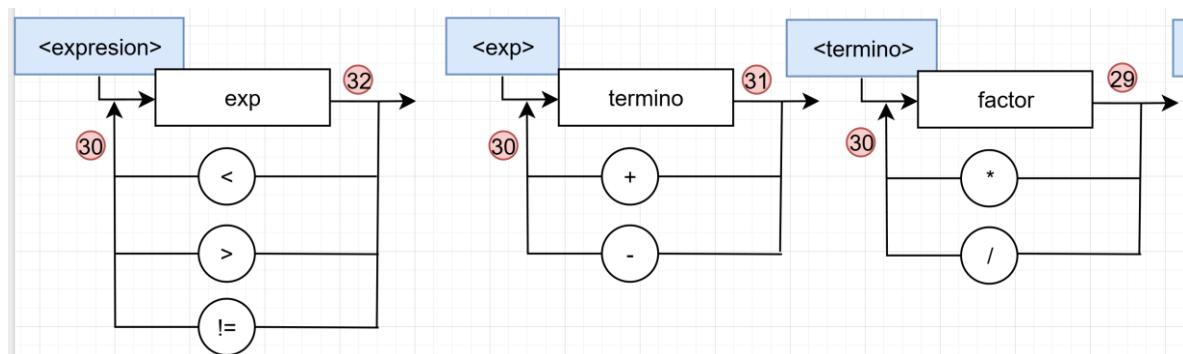
24. Generar cuádruplo PRINT con el resultado de la expresión
25. Si constTable.hasConstant(str) agregar CTE_STRING a la tabla de constantes, asignandole una dirección de memoria, Generar cuádruplo PRINT con el string



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



27. Si !dirFunc.variableExists(id) lanzar error de variable no existente, sino
 $\text{varType} = \text{dirFunc.getVariableType(id)}$
28. stackContext.pushOperand(var, varType) donde se hacen los respectivos
 push al stackOperand y al typeOperand



29. Si stackContext.topOperator() es * o / entonces
- $\text{rightOperand} = \text{stackContext.popOperand}()$, $\text{rightType} = \text{stackContext.popType}()$, $\text{leftOperand} = \text{stackContext.popOperand}()$, $\text{leftType} = \text{stackContext.popType}()$, $\text{op} = \text{stackContext.popOperator}()$
 - se hace validación de $\text{resultType} = \text{cuboSemantico.getResultType(op, leftType, rightType)}$ y si es null regresa error por tipos incompatibles

- c. tempName = generateTemp() y se genera tempAddress = memoria.assignTempAddress(resultType) para hacer dirFunc.addVariable(tempName, resultType, tempAddress)
 - d. Genera cuádruplo con operador, leftOperand, rightOperand y tempName
 - e. stackContext.pushOperand(tempName, resultType)
- 30.stackContext.pushOperator(op)
- 31.Si stackContext.topOperator() es + o – hacer número #29
- 32.Si stackContext.topOperator() es <, > o != hacer número #29

¿Cómo ejecutar las pruebas?

Las pruebas son ejecutadas usando:

- Compilar archivos
 - javac -cp ".;libs/antlr-4.13.2-complete.jar" src/lex_par/*.java src/mem/*.java src/sem/exp/*.java src/sem/funcs_vars/*.java src/sem/*.java src/*.java
- Correr archivo test
 - java -cp ".;libs/antlr-4.13.2-complete.jar;src" Test

Cabe aclarar que en el archivo Test.java es necesario cambiar el path de la línea 19 a las pruebas que se deseen ejecutar.