Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

TC3002B.501: Desarrollo de aplicaciones avanzadas de ciencias computacionales
(Gpo 501)

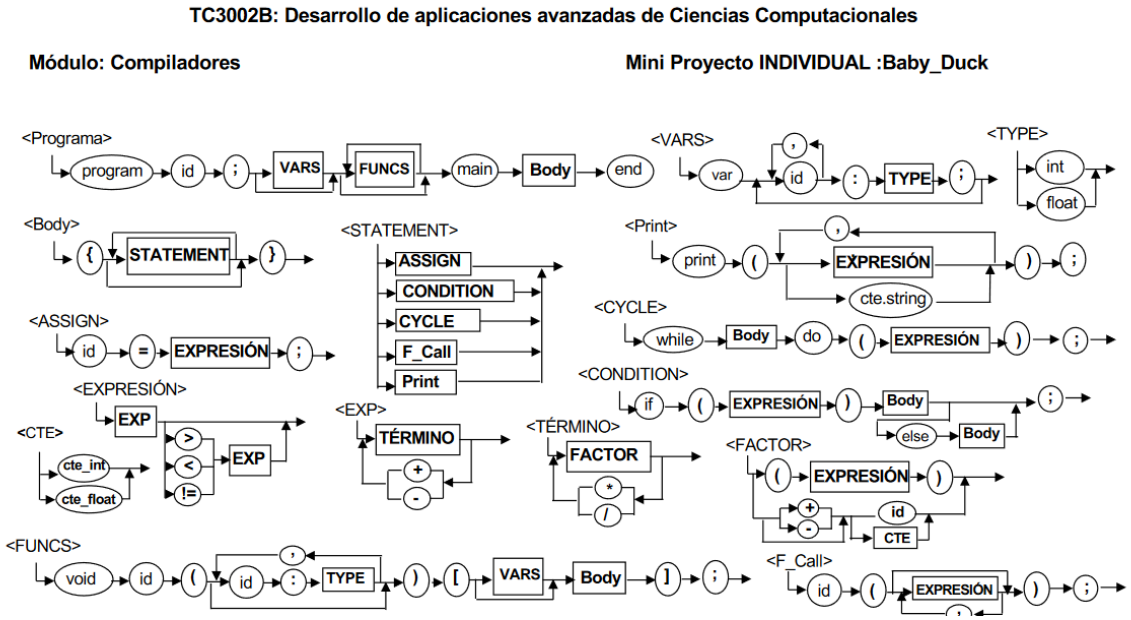
Reto Final: Mini-Reto 4

Narhari Olalde Guajardo | A01284077

15 de Noviembre del 2023

Introducción

En la sociedad actual, los compiladores han sido una pieza fundamental en el ecosistema de la informática moderna, actuando como la puerta de enlace entre el programador y la computadora. A medida que la complejidad del software y los sistemas informáticos han aumentado, la eficiencia, precisión y versatilidad de los compiladores se convierten en factores críticos para el rendimiento y la escalabilidad. Debido a lo anterior, se vuelve vital el entender el cómo los compiladores no sólo traducen el código fuente de alto nivel a lenguaje máquina, sino que también optimizan el código, gestionan los recursos y facilitan la depuración y el mantenimiento. Asimismo, con el actual auge de los campos como la inteligencia artificial, la computación en la nube y la Internet de las Cosas (IoT), donde el rendimiento y la eficiencia son clave, los compiladores juegan un papel crucial al permitir la rápida iteración y mejora de los algoritmos y aplicaciones. Debido a lo anterior, se nos fue encargada la tarea de desarrollar un pequeño compilador capaz de ejecutar las capacidades básicas para un diagrama que se mostrará a continuación :



Fase 1:

Expresiones Regulares para elementos léxico

Para poder llevar a cabo la misión de desarrollar nuestro compilador “from scratch”, es necesario poder generar las expresiones regulares de los elementos de nuestro léxico, en este caso serán 4 las principales:

- Letter
  - Una Regex que establecerá cualquier letra (o conjunto de letras) possible
- Number
  - Una Regex que pueda representar un número (o conjunto de estos)
- String
  - Una Regex para poder detectar un string el cual es un conjunto de caracteres entre símbolos de comillas
- WS
  - Una Regex para identificar (y skipear) todos los blank spaces

A continuación se muestra el resultado de estas Regex:

```
LETTER: [a-zA-Z]+;
NUMBER: [0-9]+;
STRING: '"' .*? '"';
WS: [ \t\r\n]+ -> skip;
```

Reglas gramaticales equivalentes a los diagramas

A continuación se muestran las regla para el diagrama anterior:

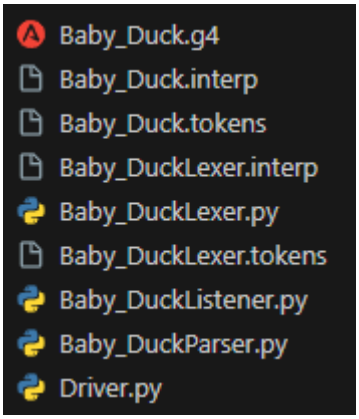
```
programa: PROGRAM id SEMICOLON  vars?  (funcs)*  MAIN  body <EOF>;

statement: assign | condition | cycle | f_call | print;

vars: VAR ((id (COMMA id)* COLON TYPE)? SEMICOLON)+;

funcs: VOID id OPEN_PAREN (id COLON TYPE (COMMA id COLON TYPE)*)?
CLOSE_PAREN OPEN_BRACKET vars? body CLOSE_BRACKET SEMICOLON;

f_call: id OPEN_PAREN (expresion (COMMA expresion)*)? CLOSE_PAREN
SEMICOLON;
```

```
print: (PRINT_WORD OPEN_PAREN (expresion | STRING) (COMMA (expresion |
STRING))*  CLOSE_PAREN SEMICOLON);

condition: IF OPEN_PAREN expresion CLOSE_PAREN body (ELSE
body)?SEMICOLON;

cycle: WHILE  body DO OPEN_PAREN expresion CLOSE_PAREN SEMICOLON;

body: OPEN_KEY (statement)* CLOSE_KEY;

assign: id EQUAL expresion SEMICOLON;

factor: (OPEN_PAREN expresion CLOSE_PAREN) | ((ADD|SUB)? (CTE | id));

termino: factor | ((factor (MULT|DIV)) + factor);

exp: (termino) | ((termino (ADD|SUB)) + termino);

expresion: exp | (exp (GREATER|LESS|DIFFERENT) exp);

TYPE: INT | FLOAT;
CTE: NUMBER | DECIMAL;

FLOAT: 'float';
INT: 'int';

VAR: 'var';
MAIN: 'main';
VOID: 'void';
PROGRAM: 'program';
WHILE: 'while';
DO: 'do';
IF: 'if';
```

```
ELSE: 'else';
PRINT_WORD: 'print';

OPEN_PAREN: '(';
CLOSE_PAREN: ')';
OPEN_KEY: '{';
CLOSE_KEY: '}';
OPEN_BRACKET: '[';
CLOSE_BRACKET: ']';

MULT: '*';
DIV: '/';
EQUAL: '=';
ADD: '+';
SUB: '-';
GREATER: '>';
LESS: '<';
DIFFERENT: '!=';
UNDERSCORE: '_';
DOT: '.';
COMMA: ',';
SEMICOLON: ';';
COLON: ':';

ID: '_'? LETTER (LETTER | NUMBER | '_')*;
```

Scanners y Parsers

Una vez obtenido lo anterior, y utilizando antlr, podemos generar los interpreters y lexers como se mostrará a continuación:



Fase 2: Uso de Inyección de código

Para poder trabajar en nuestro compilador y evitar que sea un simple cascarón, tenemos que utilizar se implementó la inyección directa para poder establecer puntos neurálgicos que nos permitan interactuar con los tokens generados en nuestro lenguaje patito. Durante esta etapa estaremos viendo la implementación de un cubo semántico (para poder establecer las relaciones que poseen las variables entre ellas al hacer operaciones), las estructuras de datos para el Directorio de Funciones y a las Tablas de Variables, así como el establecer los puntos neurálgicos previamente mencionados.

Cubo Semántico

Para el manejo correcto de variables, se implementó un cubo semántico que define las interacciones entre las variables y sus tipos (en este caso INT y FLOAT), donde, dependiendo de qué operación se lleve a cabo, el resultado de la operación se establece en un tipo. En este caso, se creó el siguiente cubo semántico dentro del archivo MyFunctions.py en la clase SemanticCube:

```
class SemanticCube:
  def __init__(self):
    self.cube = {
      ('int', 'int', '+'): 'int',
      ('int', 'float', '+'): 'float',
      ('float', 'int', '+'): 'float',
      ('float', 'float', '+'): 'float',
      ('int', 'int', '-'): 'int',
      ('int', 'float', '-'): 'float',
      ('float', 'int', '-'): 'float',
      ('float', 'float', '-'): 'float',
      ('int', 'int', '*'): 'int',
      ('int', 'float', '*'): 'float',
```

```
            ('float', 'int', '*'): 'float',
            ('float', 'float', '*'): 'float',
            ('int', 'int', '/'): 'int',
            ('int', 'float', '/'): 'float',
            ('float', 'int', '/'): 'float',
            ('float', 'float', '/'): 'float',
            ('int', 'int', '<'): 'bool',
            ('int', 'float', '<'): 'bool',
            ('float', 'int', '<'): 'bool',
            ('float', 'float', '<'): 'bool',
            ('int', 'int', '>'): 'bool',
            ('int', 'float', '>'): 'bool',
            ('float', 'int', '>'): 'bool',
            ('float', 'float', '>'): 'bool',
            ('int', 'int', '!='): 'bool',
            ('int', 'float', '!='): 'bool',
            ('float', 'int', '!='): 'bool',
            ('float', 'float', '!='): 'bool',
            ('int', 'int', '='): 'int',
            ('int', 'float', '='): 'error',
            ('float', 'int', '='): 'float',
            ('float', 'float', '='): 'float',
        }
    def get_type(self, left, right, operator):
        return self.cube.get((left, right, operator), 'error')
```

Estructura de datos

Tabla de Variables

Además del cubo semántico, dentro del mismo archivo, se creó la clase VariableTable, la cual se asemeja a un diccionario de Python, la cual permite, mediante funciones de la clase, agregar variables a la tabla de acuerdo a su scope así como poder registrar su tipo y address.

```
class VariableTable:
    def __init__(self):
        self.variables = {"global": {}}
    def set_scope(self, scope):
        if scope not in self.variables:
            self.variables[scope] = {}
    def add_variable(self, name, var_type, scope):
        if name in self.variables[scope]:
```

```
            raise Exception(f"Variable {name} already declared.")
        self.variables[scope][name] = {
            'type': var_type,
            'address': next(AVAIL)
        }
```

Asimismo, dentro de esta función se permite acceder al valor de cada variable, a su tipo, encontrar en qué scope está la variable y finalmente, poder limpiar la tabla de variables en caso de ser necesario

```
def get_variable_type(self, name, scope):
        return self.variables[scope][name].get('type', None)
    def get_variable_value(self, name, scope):
        return self.variables[scope][name].get('value', None)
    def set_variable_value(self, name, value, scope):
        self.variables[scope][name]['value'] = value
    def clean_variables(self, scope):
        self.variables[scope].clear()
    def find_scope(self, name):
        for scope in self.variables:
            if name in self.variables[scope]:
                return scope
        return None
```

Tabla de Funciones

Similar a la tabla de variables, la tabla de funciones se crea mediante la clase FunctionTable, la cual se asemeja a un diccionario y posee funciones que le permiten añadir nuevas funciones a la tabla junto con su tipo (void, int o float), sus parámetros y de cuantos quads consiste la función.

```
class FunctionTzable:
    def __init__(self):
        self.functions = {}

    def add_function(self, func_id, func_type):
        if func_id in self.functions:
            raise Exception(f"Function {func_id} already declared.")
        self.functions[func_id] = {
            "type": func_type,
```

```
        "params": {},
        "quad_count": 0,
    }
    return True
```

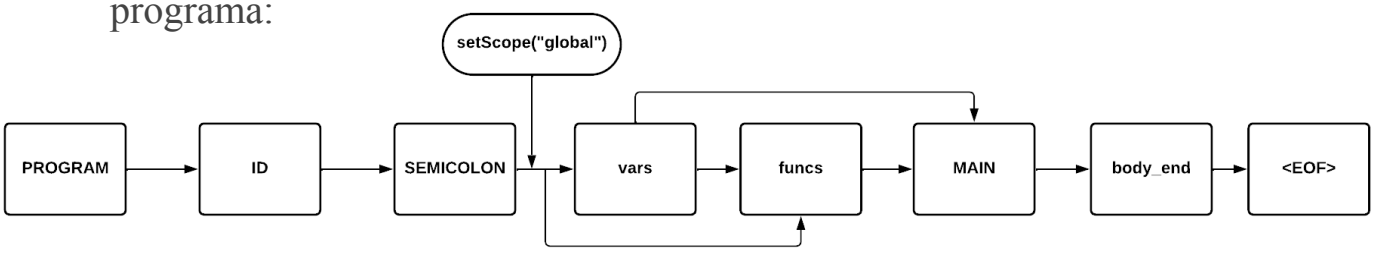
Asimismo, dentro de esta función se permite retornar el diccionario de la función, obtener su tipo de retorno, sus parámetros, los tipos de sus parámetros así como la cantidad de parámetros que posee. Finalmente también permite limpiar la tabla de funciones en caso de ser necesario

```
def get_function(self, func_id):
    return self.functions.get(func_id, None)
def get_function_type(self, func_id):
    return self.functions[func_id]["type"]
def get_function_params(self, func_id):
    return self.functions[func_id]["params"]
def get_function_param(self, func_id, param_id):
    return self.functions[func_id]["params"].get(param_id, None)
def get_function_param_id(self, func_id, index):
    return list(self.functions[func_id]["params"].keys())[index]
def get_function_param_type(self, func_id, param_id):
    return self.functions[func_id]["params"][param_id]
def get_function_param_count(self, func_id):
    return len(self.functions[func_id]["params"])
def clean_functions(self):
    self.functions.clear()
```

Puntos Neurálgicos

Para implementar y llenar las Tablas de variables y funciones, se establecieron puntos neurálgicos en los siguientes puntos del lenguaje ANTLR (.g4)

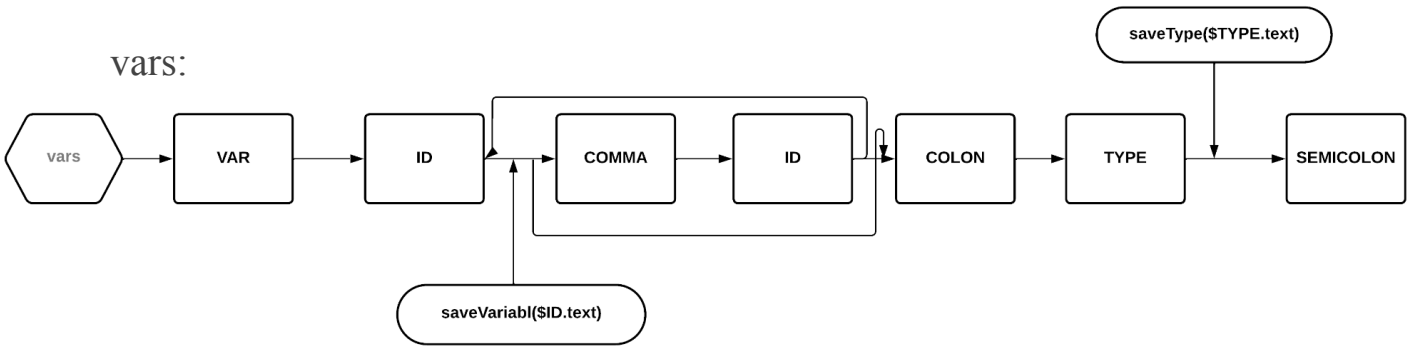
programa:



- setScope(“global)

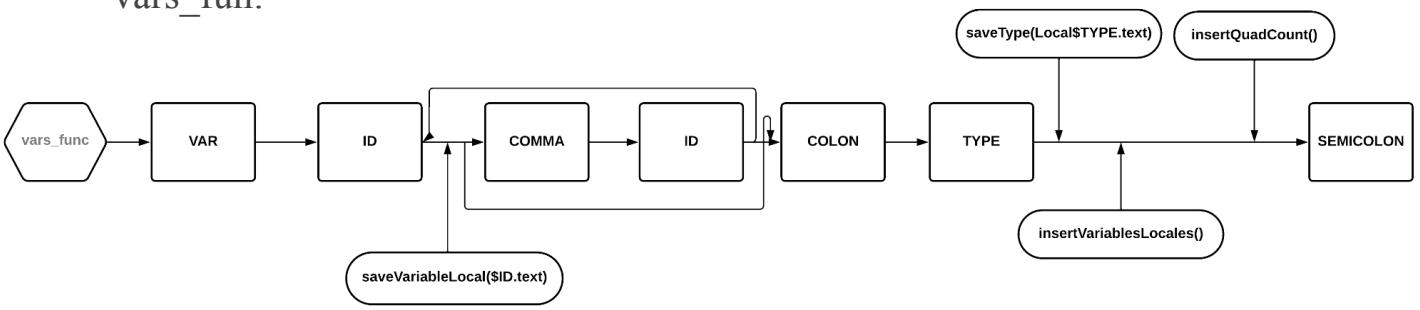
- Se establece el scope donde se declaran las variables (en este caso es global al ser antes de entrar a las funciones).

vars:



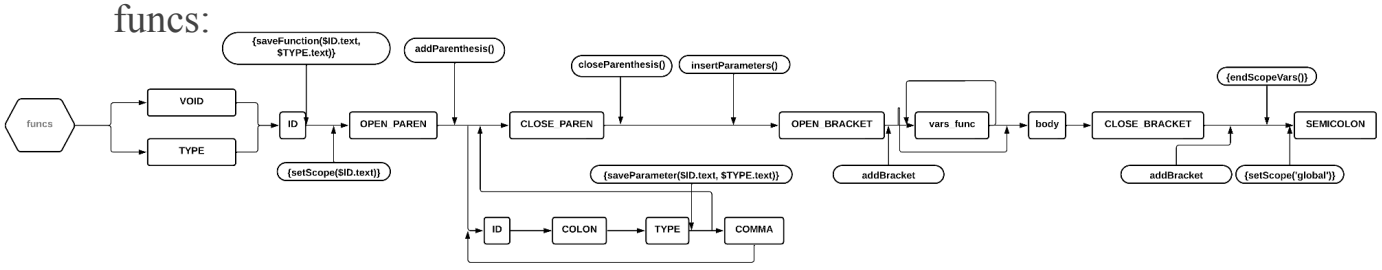
- saveVariable($ID.text)
  - Guarda de forma local todas las variables presentes hasta llegar a saveType
- saveType($TYPE.text)
  - Se agregan las variables (con su tipo) a la tabla de variables.

vars_fun:



- saveVariableLocal($ID.text)
  - Guarda de forma local todas las variables presentes hasta llegar a saveTypeLocal
- saveTypeLocal($TYPE.text)
  - Se agregan las variables (con su tipo) a la tabla de variables.
- insertVariablesLocales()
  - Se insertan las variables a la tabla de funciones
- insertQuadCount()
  - Se inserta en la tabla de funciones, la cantidad de cuadruplos que hay en la función

funcs:



- saveFunction($ID.text, $TYPE.text)
    - Se establece como la función actual el id de la función y se inserta la función en la tabla de funciones con su tipo, en caso de ya existir, regresa un error
- setScope($ID.text)
    - Se setea el scope de la función (basada en el id)
- saveParameter($ID.text, $TYPE.text)
    - Se añade la variable de parámetro en la tabla de parametros(local) y en la tabla de variables en el scope de la función
- insertParameters()
    - Se pushea la tabla parámetros a la tabla de funciones dentro del scope y se inserta de igual manera la cantidad de parámetros
- endScopeVars()
    - Se limpian las tablas locales (de funciones y variables) y se pushea el cuádruplo de ENDFUNC
- setScope("global")
    - Se setea el scope de nuevo a global al salir de la función

Fase 3: Implementación de Cuádruplos

Estructuras de datos

Cuádruplos

A diferencia de las últimas dos estructuras presentadas, los cuádruplos en este caso se basaron en un array con la funciones primordial, de poder agregar un cuádruplo al array y sumar uno en su contado:

```
class Quadruple:
    def __init__(self):
        self.quadruples = []
        self.quadruple_count = 0

    def push(self, quadruple):
        self.quadruples.append(quadruple)
        self.quadruple_count += 1
```

Stacks

Se implementó una clase stack especializada para todas las pilas necesarias (PilaO, POper, PTypes, PJumps, QuadStack, ResultStack). En esta clase stack, se puede obtener el último valor añadido con un pop o insertar uno nuevo con un push.

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, element):
        self.stack.append(element)
    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        raise Exception("Stack is empty")
```

Asimismo, se puede observar el último valor insertado sin la necesidad de hacerle pop, esto mediante un peek, y verificar si el stack está vacío.

```
    def is_empty(self):
        return len(self.stack) == 0
    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        raise Exception("Stack is empty")
```

Lenguaje modificado

Una vez se pueden almacenar los operandos, operadores, los tipos y los resultados, se pueden implementar nuevos puntos neurálgicos que permitan generar los cuádruplos para posteriormente ejecutarlos dentro de nuestra máquina virtual. El lenguaje con los nuevos puntos neurálgicos es el sigueinte:

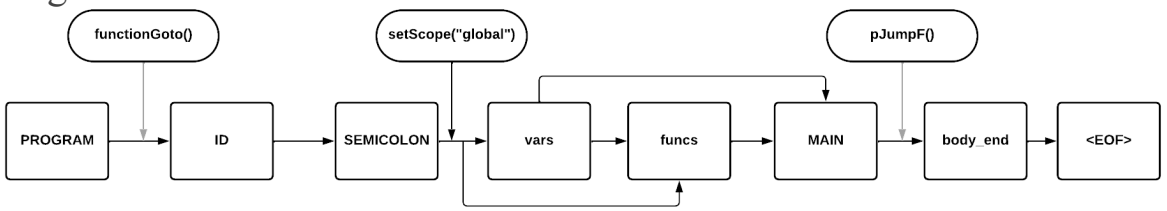
```
    programa: {functionGoto()} PROGRAM  ID SEMICOLON  {setScope("global")} vars*
(funcs)*  MAIN {pJumpF()}  body_end  <EOF>;

vars: (VAR ID {saveVariable($ID.text)} (COMMA ID {saveVariable($ID.text)} )*  COLON
TYPE {saveType($TYPE.text)}  SEMICOLON);

vars_func: (VAR ID {saveVariableLocal($ID.text)} (COMMA ID {saveVariableLocal($ID.text)}
)*  COLON TYPE {saveTypeLocal($TYPE.text)}  {insertVariablesLocales()}
{insertQuadCount()} SEMICOLON);

body_end: OPEN_KEY {addKey()} (statement)* CLOSE_KEY {closeKey()}
{solveQuadruples()};
```

```
body: OPEN_KEY {addKey()} (statement)* CLOSE_KEY {closeKey()};

statement: assign | condition | cycle | f_call | print_;

assign: ID EQUAL expresion {assignExpression($ID.text)} SEMICOLON;

funcs:  (VOID | TYPE) ID {saveFunction($ID.text, $TYPE.text)} {setScope($ID.text)}
OPEN_PAREN {addParenthesis()} (ID COLON TYPE {saveParameter($ID.text, $TYPE.text)}
(COMMA ID COLON TYPE {saveParameter($ID.text, $TYPE.text)})*)? CLOSE_PAREN
{closeParenthesis()} {insertParameters()} OPEN_BRACKET {addBrackets()} (vars_func)* body
CLOSE_BRACKET {closeBrackets()} {endScopeVars()} {setScope("global")} SEMICOLON;

f_call: ID {proveFunction($ID.text)} OPEN_PAREN {generateERA()} {addParenthesis()}
(expresion {arguments()} (COMMA {addK()} expresion {arguments()})*)? CLOSE_PAREN
{closeParenthesis()} {isNull()} SEMICOLON;

print_: (PRINT_WORD  OPEN_PAREN {addParenthesis()} {addPrint()} (expresion | STRING
{saveString()}) {printExp()} (COMMA {addPrint()} (expresion | STRING {saveString()})
{printExp()})*  CLOSE_PAREN {closeParenthesis()} SEMICOLON);

condition: IF OPEN_PAREN {addParenthesis()} expresion CLOSE_PAREN
{closeParenthesis()} {csTrue()} body (ELSE {GOTO()} body)? SEMICOLON {pJumpF()};

cycle: WHILE {whileLoop()} OPEN_PAREN {addParenthesis()} expresion  CLOSE_PAREN
{closeParenthesis()} {expresionWhile()} body SEMICOLON {endWhile()};

factor: (OPEN_PAREN {addParenthesis()} expresion CLOSE_PAREN {closeParenthesis()}) |
((ADD |SUB )? (CTE {stackConstant($CTE.text)}| ID {stackId($ID.text)}));

termino: ((factor {multDiv()} ((MULT {addMultDiv($MULT.text)}) | (DIV
{addMultDiv($DIV.text)}) ))* factor {multDiv()});

exp:  (termino {sumRes()} ((ADD {addSumSub($ADD.text)}) | (SUB
{addSumSub($SUB.text)}) ))* termino {sumRes()};

expresion: (exp {compareExp()} (TYPE_CONDITIONAL
{addCompare($TYPE_CONDITIONAL.text)}))* exp {compareExp()};
```

En este caso se omitieron las reglas del Lexema ya que no sufrieron modificaciones y se mantendrán sólo las reglas del parser para mostrar los puntos neurálgicos nuevos.

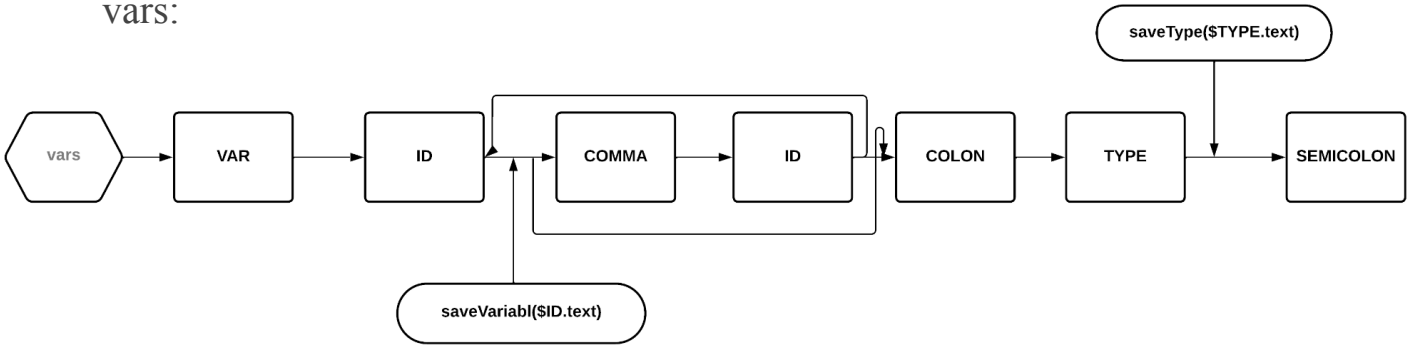
Puntos Neurálgicos

Programa:



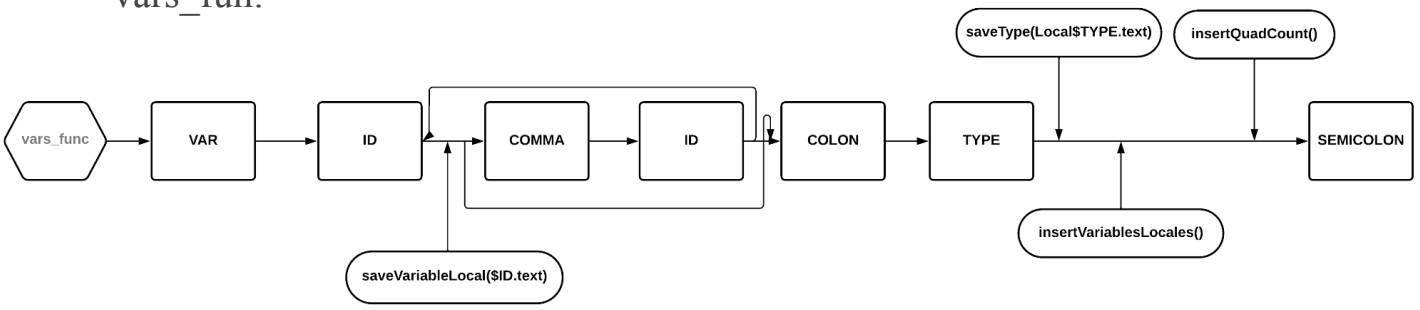
- functionGoto():
  - Se genera un cuádruplo con la función de GOTO que posteriormente pueda almacenar a donde se debería iniciar el codigo (apuntará al main)
- setScope(“global”):
  - Esta ya fue explicada anteriormente y sirve para setear el scope en global
- pJumpF():
  - Se inserta el salto para el GOTO del main

vars:



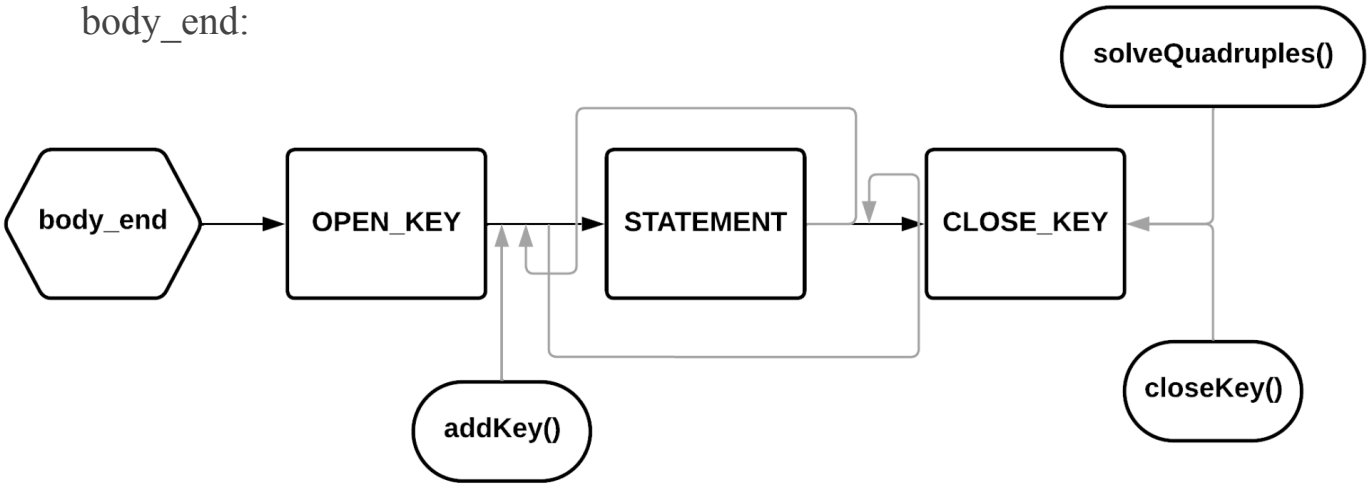
- saveVariable($ID.text):
  - Guarda de forma local todas las variables presentes hasta llegar a saveType
- saveType($TYPE.text):
  - Se agregan las variables (con su tipo) a la tabla de variables.

vars_fun:



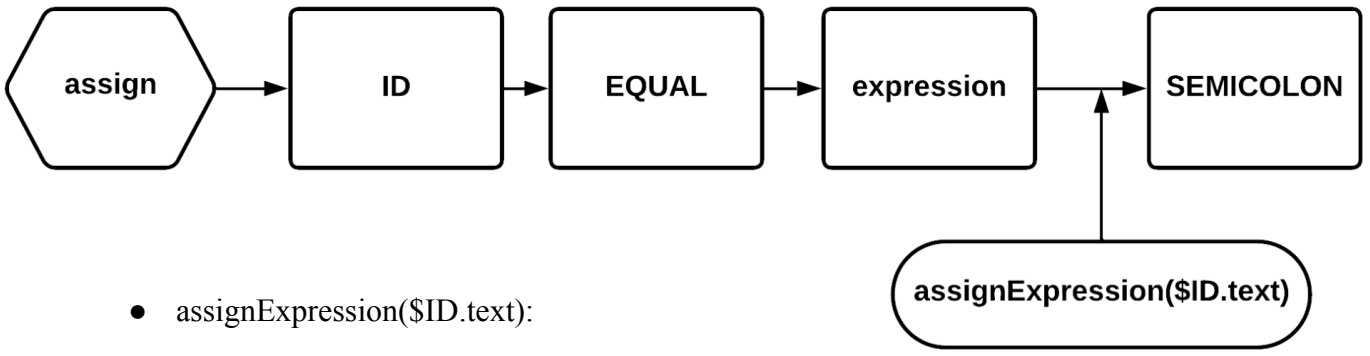
- saveVariableLocal($ID.text):
    - Guarda de forma local todas las variables presentes hasta llegar a saveTypeLocal
- saveTypeLocal($TYPE.text):
    - Se agregan las variables (con su tipo) a la tabla de variables.
- insertVariablesLocales():
    - Se insertan las variables a la tabla de funciones
- insertQuadCount():
    - Se inserta en la tabla de funciones, la cantidad de cuadruplos que hay en la función

body_end:



- addKey():
    - Añade un símbolo de llave al stack POper
- closeKey():
    - Verifica que las llaves sean cerradas correctamente, si no, levanta el error “Missing key”
- solveQuadruples():
    - Funcion encargada de resolver los cuadrupledos ejecutando la VM.

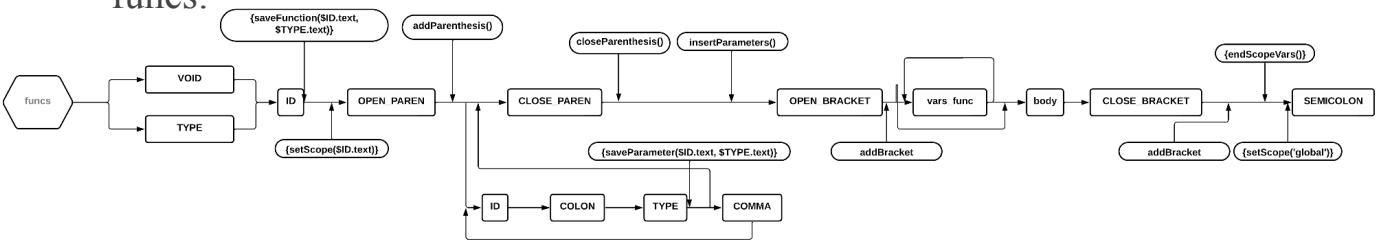
body:



- assignExpression($ID.text):

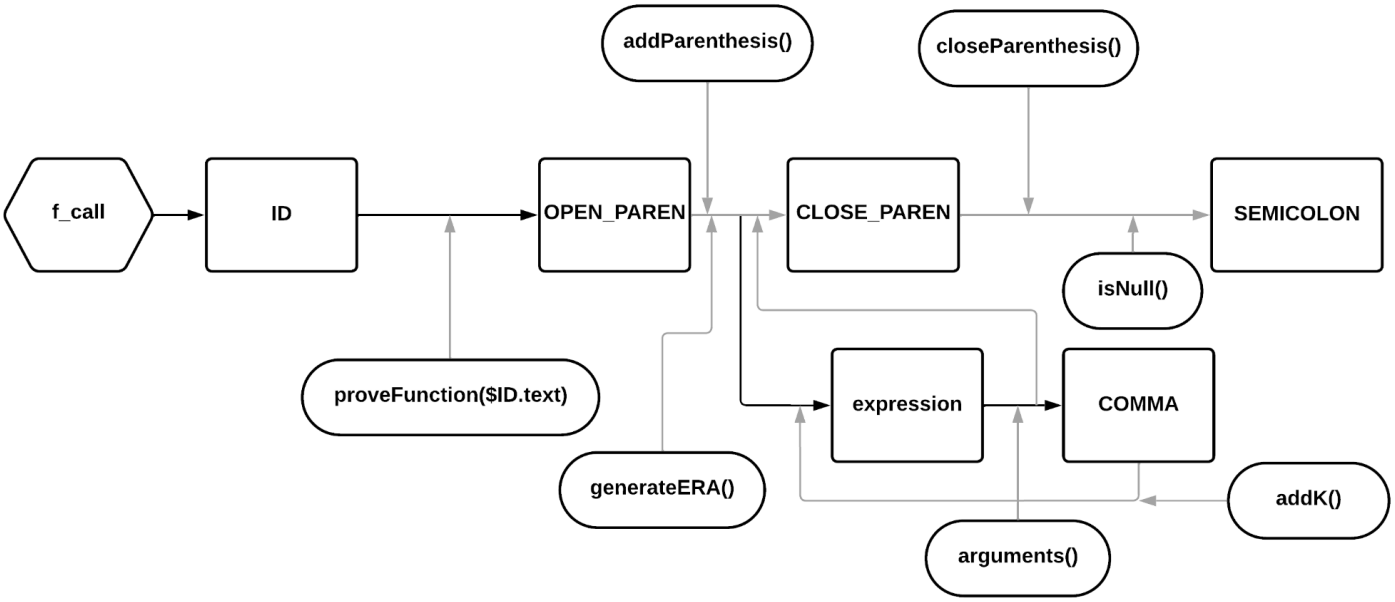
- Función que permite generar el cuádruple relacionado con la asignación. En esta función se comprueba que los tipos de las variables sean válidos de acuerdo al cuadro semántico presentado anteriormente.

funcs:



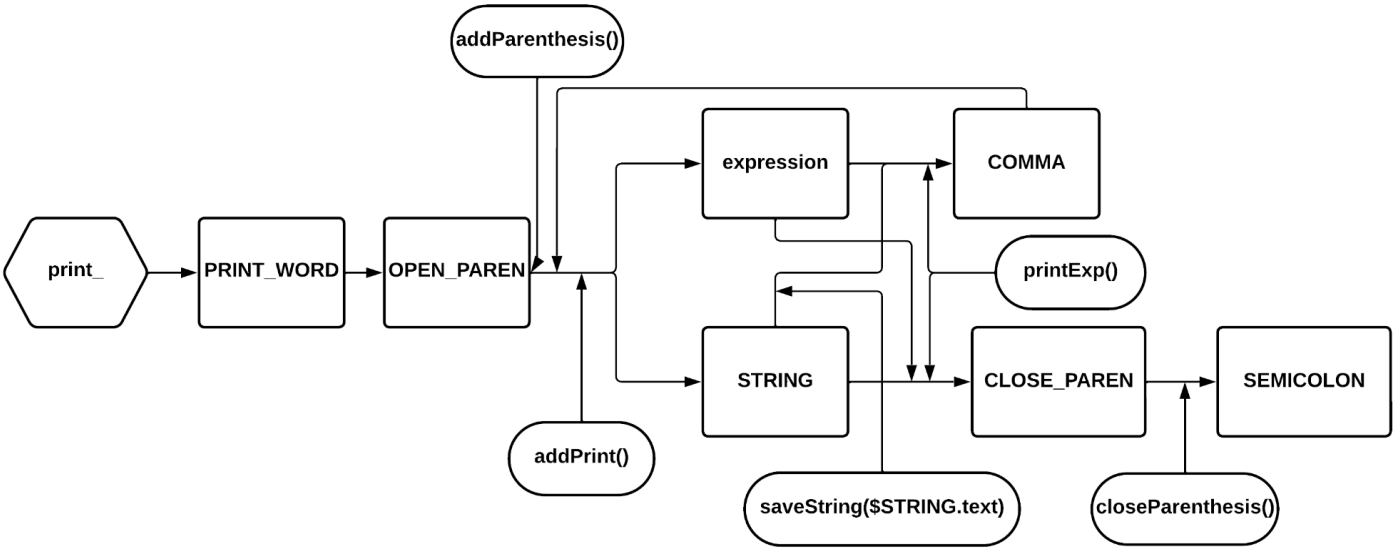
- saveFunction($ID.text, $TYPE.text):
    - Se establece como la función actual el id de la función y se inserta la función en la tabla de funciones con su tipo, en caso de ya existir, regresa un error
- setScope($ID.text):
    - Se setea el scope de la función (basada en el id)
- addParenthesis():
    - Añade un símbolo de paréntesis al stack POper
- saveParameter($ID.text, $TYPE.text):
    - Se añade la variable de parámetro en la tabla de parametros(local) y en la tabla de variables en el scope de la función
- closeParenthesis():
    - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”
- insertParameters():
    - Se pushea la tabla parámetros a la tabla de funciones dentro del scope y se inserta de igual manera la cantidad de parámetros
- addBracket():
    - Añade un símbolo de bracket al stack POper
- closeBrackets():
    - Verifica que los brackets sean cerradas correctamente, si no, levanta el error “Missing bracket”
- endScopeVars():
    - Se limpian las tablas locales (de funciones y variables) y se pushea el cuádruplo de ENDFUNC
- setScope("global"):
    - Se setea el scope de nuevo a global al salir de la función

f_call:



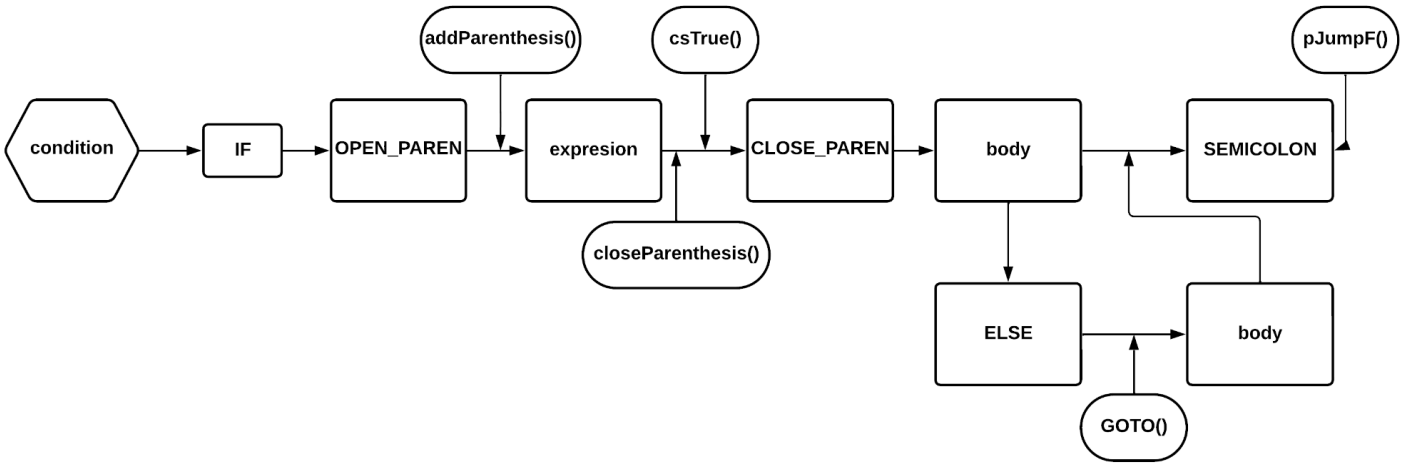
- proveFunction($ID.text):
    - Verifica que la función actual exista y cambia el puntero de la función actual a la función a la que se entrará (esto con el id). Si la función no existe, lanza el error “Function not declared”
- addParenthesis():
    - Añade un símbolo de paréntesis al stack POper
- generateERA():
    - Se reinicia el contador de parámetros y se hace push del cuadrupledo “ERA” el cual guarda la función actual
- arguments():
    - Genera el cuadrupledo de los parametros (“PARAM”), en donde se valida primero que los tipos de los argumentos y del parámetro sean compatibles, en caso de no serlo, lanza el error “Invalid type”
- addK():
    - Función que aumenta el contador de parámetros
- closeParenthesis()
    - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”
- isNull():
    - Funcion que genera el cuadrupledo “GOSUB” con un puntero a la función actual así como revisando que el número de parámetros cuadre, en caso de no cuadrar, arroja el error “Invalid number of arguments”

print_:



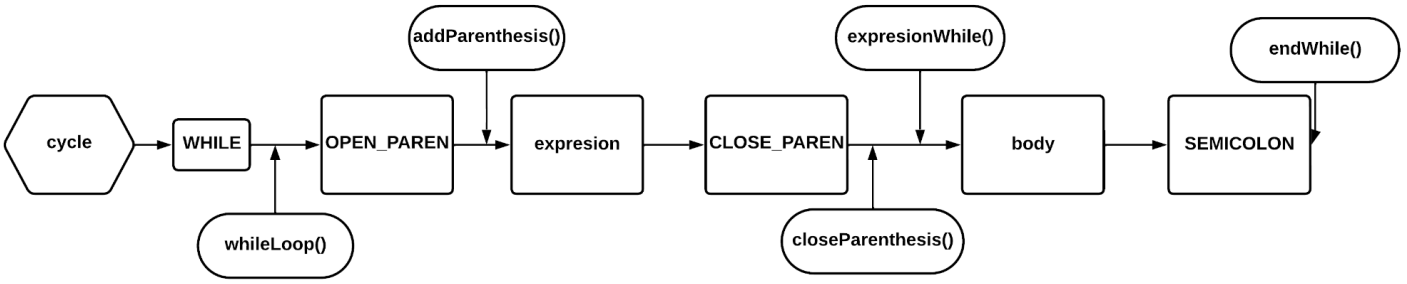
- addParenthesis():
  - Añade un símbolo de paréntesis al stack POper
- addPrint():
  - Hace push de “print” al stack POper
- saveString($STRING.text)
  - Hace push a PilaO con el valor del string y a PTypes con el tipo “string” para ser utilizados posteriormente
- printExp():
  - Genera el cuádruple de impresión si el peek de POper es “print” y lo empuja al stack de cuádruplos. En caso de que el peek de POper no sea “print”, levanta el error “Missing print”
- closeParenthesis():
  - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”

condition:



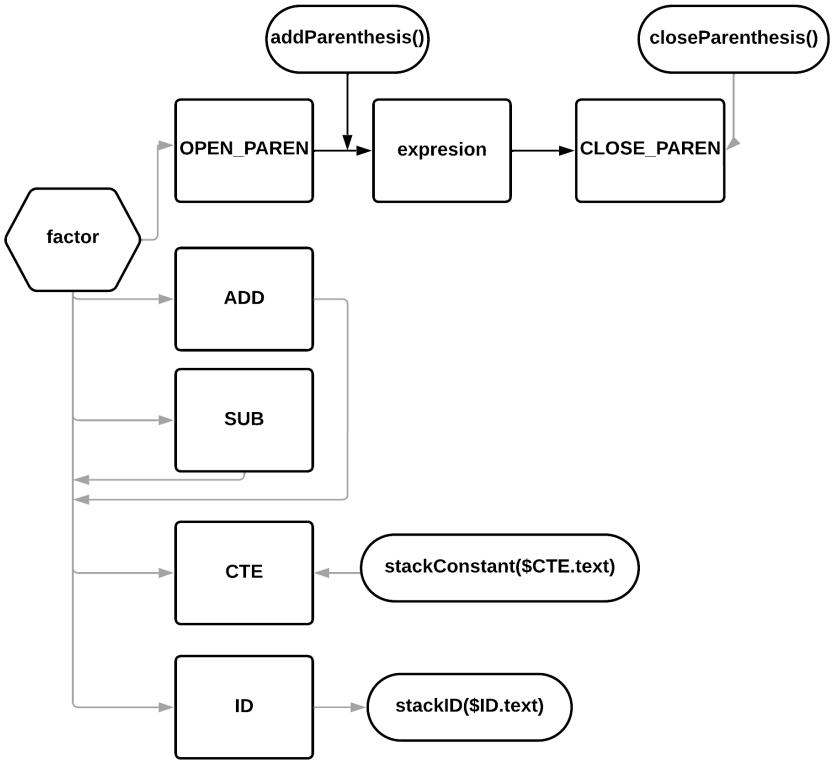
- addParenthesis():
  - Añade un símbolo de paréntesis al stack POper
- closeParenthesis():
  - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”
- csTrue():
  - Genera el cuádruple de “GOTOF” al hacer pop de PTypes y recibir un valor “bool”. El cuádruple de “GOTOF” se guarda con el resultado de la operación booleana y se empuja al stack de cuádruplos. Asimismo se hace push del salto necesario para el brinco (contador - 1). Si el tipo recibido no es booleano, levanta el error “Invalid type”
- GOTO():
  - Función que genera el cuádruple de “GOTO” al momento de encontrarse con un else. Se hace pop del brinco a false y se hace push del nuevo jump (contador - 1) y se llama a la función FILL para llenar el GOTOF
- pJumpF():
  - Función que hace pop del stack de Jumps y llama a la función FILL para indicar el brinco del cuádruplo.

cycle:



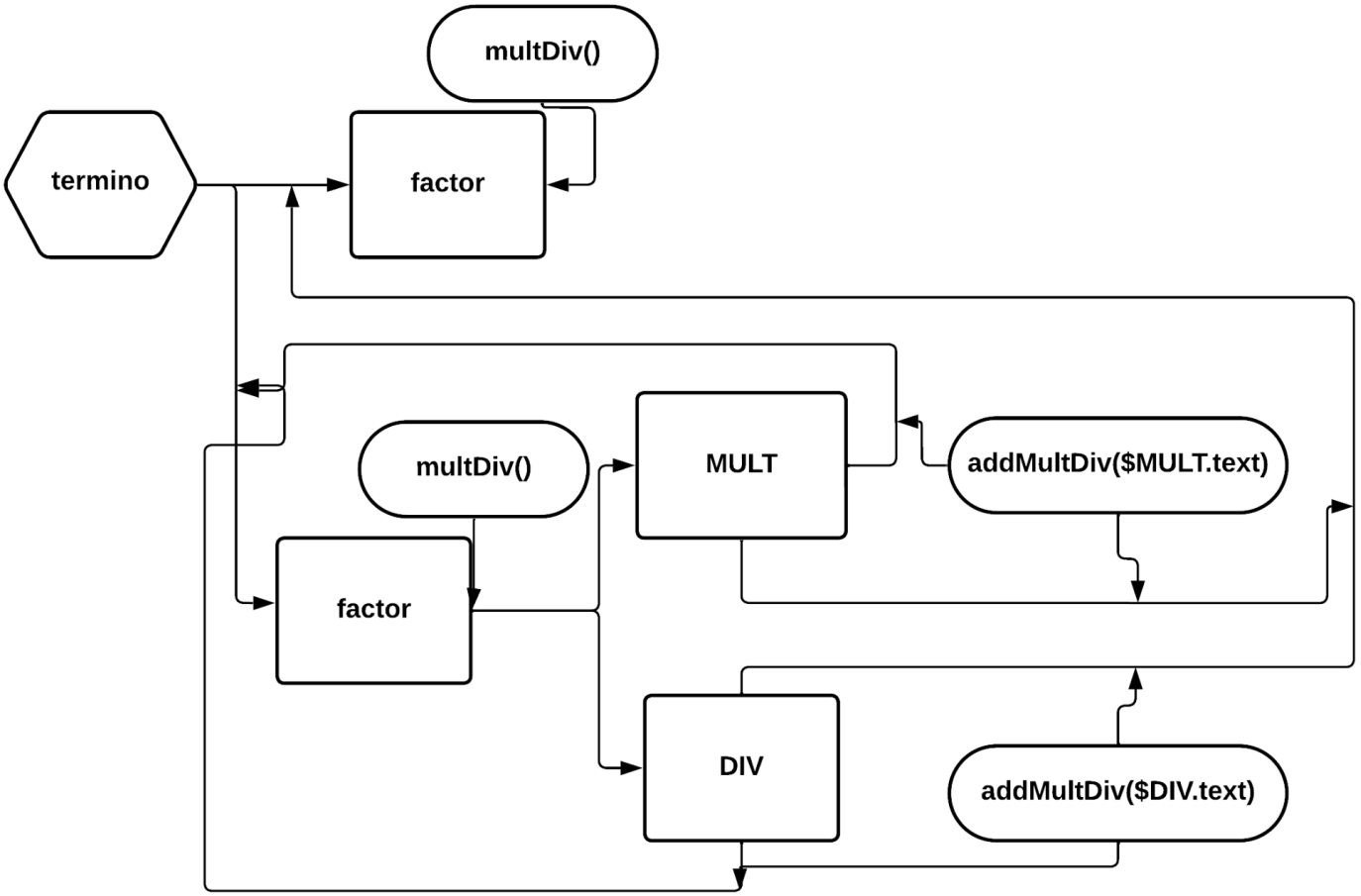
- whileLoop():
    - Marca el inicio del ciclo while empujando el contador de los cuádruplos al stack de los saltos (PJumps)
- addParenthesis():
    - Añade un símbolo de paréntesis al stack POper
- closeParenthesis():
    - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”
- expresionWhile():
    - Genera el cuádruplo del “GOTOF” del ciclo while. Básicamente espera el resultado de la expresión del while y empuja el cuádruple con el resultado. Asimismo empuja el contador de cuadruplos -1 al stack de saltos.
- endWhile():
    - Hace dos pops del stack de saltos. El primero para hacer el fill del while con el contador de cuadruplos y el segundo para hacer push del cuádruplo “GOTO” usando el segundo pop como el cuádruple objetivo.

factor:



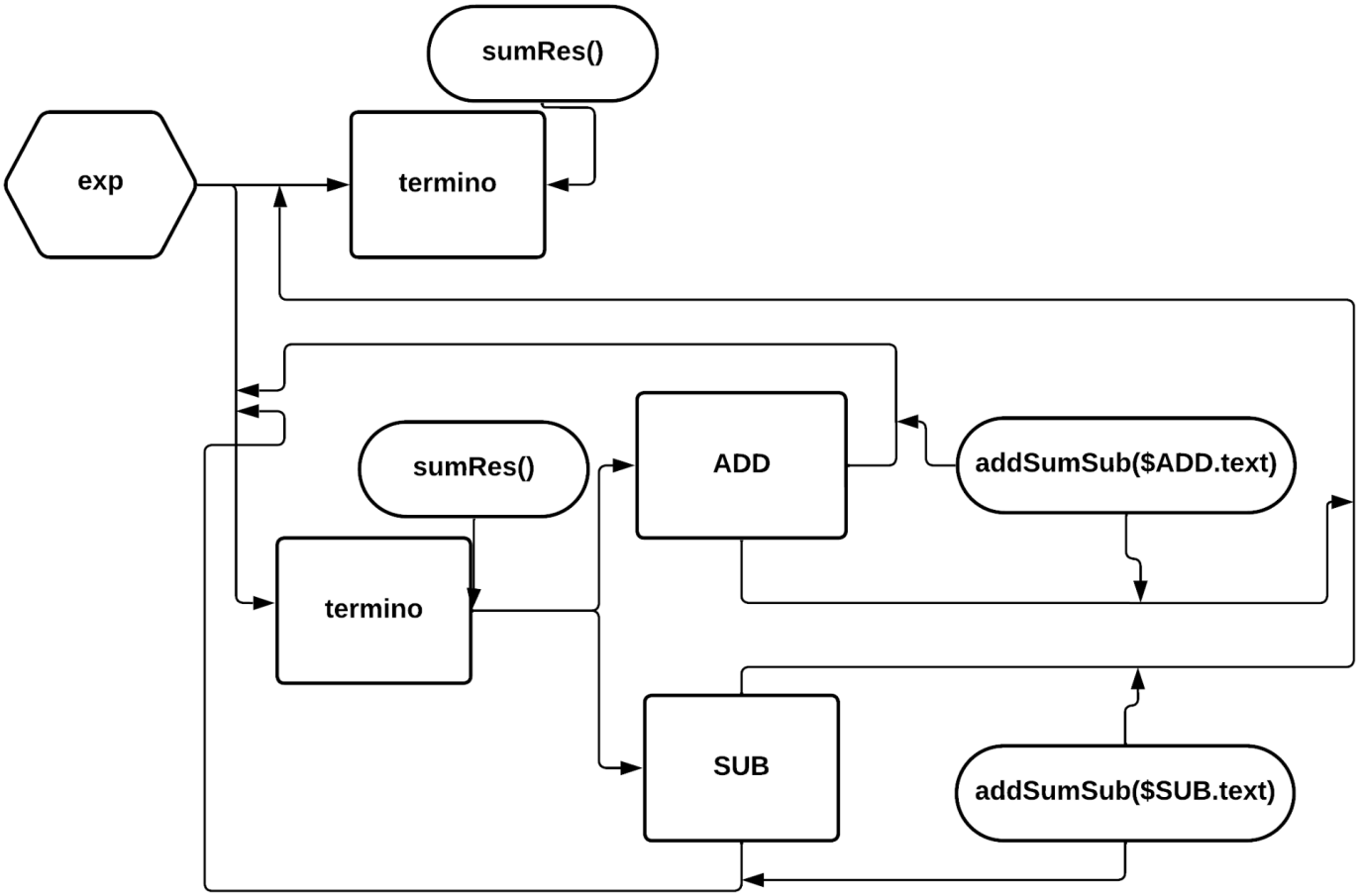
- addParenthesis():
  - Añade un símbolo de paréntesis al stack POper
- closeParenthesis():
  - Verifica que los paréntesis sean cerradas correctamente, si no, levanta el error “Missing parenthesis”
- stackConstant($CTE.text):
  - Se hace push de la constante parseada a lo que corresponda (int o float) al stack PilaO y se obtiene el tipo (mediante la función identifyNumberType) y se empuja al stack PTypes
- stackID($ID.text):
  - Similar a la anterior, se empuja el id al stack PilaO y se obtiene el tipo al que corresponde el id (de la tabla de variables) y se empuja al stack PTypes

término:



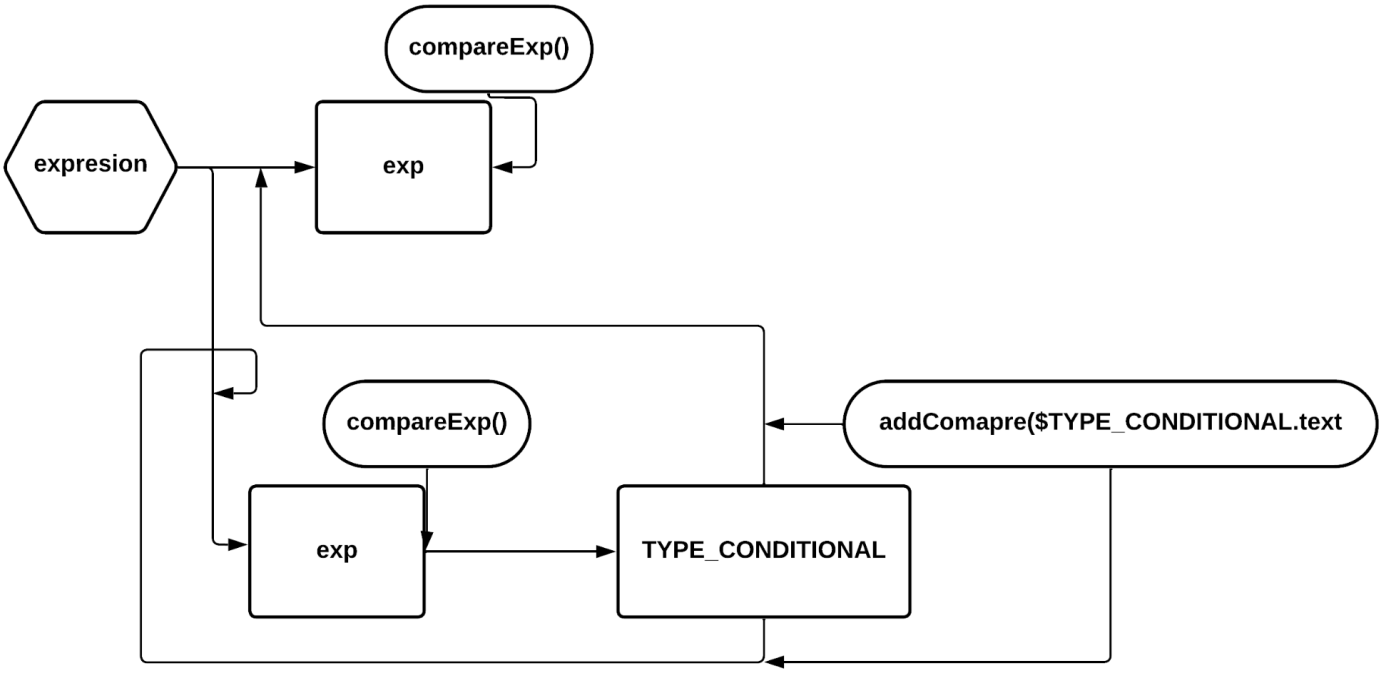
- multDiv():
    - Genera el cuádruple correspondiente a la multiplicación y división. Hace pop de los operandos izquierda y derecha y sus tipos y los compara con el cuadro semántico, si no regresa error, guarda el resultado en un espacio de memoria temporal , genera el cuádruplo y le hace push a: el cuádruplo al stack correspondiente, el resultado en memoria temporal al stack PilaO y el tipo del resultado en el stack PTypes. Lanzará un error (“Invalid type”) si el cuadro semántico regresa “error”
- addMultDiv($MULT.text) y addMultDiv($DIV.text) :
    - Empuja el símbolo correspondiente de multiplicacion o division al stack POper

exp:



- sumRes():
    - Genera el cuádruple correspondiente a la suma y resta. Hace pop de los operandos izquierda y derecha y sus tipos y los compara con el cuadro semántico, si no regresa error, guarda el resultado en un espacio de memoria temporal , genera el cuádruplo y le hace push a: el cuádruplo al stack correspondiente, el resultado en memoria temporal al stack PilaO y el tipo del resultado en el stack PTypes. Lanzará un error (“Invalid type”) si el cuadro semántico regresa “error”
- addSumSub($ADD.text) y addSumSub($SUB.text) :
    - Empuja el símbolo correspondiente de suma o resta al stack POper

exp:



- compareExp():
    - Genera el cuádruple correspondiente a la las comparaciones (<,>,!=). Hace pop de los operandos izquierda y derecha y sus tipos y los compara con el cuadro semántico, si no regresa error, guarda el resultado en un espacio de memoria temporal , genera el cuádruplo y le hace push a: el cuádruplo al stack correspondiente, el resultado en memoria temporal al stack PilaO y el tipo del resultado en el stack PTypes. Lanzará un error (“Invalid type”) si el cuadro semántico regresa “error”
- addCompare($TYPE_CONDITIONAL.text) :
    - Empuja el símbolo correspondiente de comparación al stack POper

Fase 4: Máquina Virtual

Para esta fase, se desarrolló una pequeña máquina virtual capaz de ejecutar el código presentado mediante los cuádruplos generados. Para esto, fue necesario el uso de Stacks que permitieran 2 cosas. La primera es hacer el recorrido de los cuádruplos, utilizando el generado con espacios de memoria y después, utilizando esos de base, en otro Stack, ir pusheando los que fueran apareciendo. Asimismo, debido a que los cuádruplos ya se encontraban en orden, para el manejo de memoria solo se utilizó dos stacks más (QuadStack y ResultStack) para poder ir convirtiendo los espacios temporales en los resultados esperados.

En torno al manejo de cuadruplos, se creo la funcion solveQuadruples(), en un ciclo While igual a true, el cual poseía un contador que iba siguiendo el cuádruplo que tocara. Este contador iba aumentando de 1 en 1 a menos que se cruzara con un cuádruple de condición y la condición se cumpliera (o no), esto mediante los cuádruplos de GOTO y GOTOF.

La función empieza revisando si QuadStack (nuestro stack que almacena los resultados temporalmente) esta lleno o vacío, y si contiene elementos, los compara con los valores de los índices 1 y 2 de los cuádruplos para ver si era necesario reemplazar un valor temporal por la de un resultados. Después, procede a revisar si no hay variables declaradas en los cuádruplos, y en caso de haberlas, reemplaza la variable por el valor de esta. A continuación ocurre una comparación de 8 ifs:

- El primer if procede a revisar si el índice 0 del cuádruplo es GOTOF y en caso de cumplir con las condiciones, este realiza un brinco al cuádruplo que marque la acción (esto es reemplazando el valor actual del contador por el del índice 3)
- El segundo if revisa si el índice 0 es un GOTO, y en caso de serlo realiza un brinco al cuádruplo que marque la acción (esto es reemplazando el valor actual del contador por el del índice 1)
- El tercer print ejecuta un print si detecta que el índice 0 es “print”
- Del cuarto al séptimo if, están en función de las llamadas de funciones, las cuales por el momento solo imprimen el valor del cuádruplo y aumentan en 1 el contador. Estos cuadruplos se activan si poseen un índice 0 de ENDFUNC (cuarto if), GOSUB (quinto if), ERA (sexto if) y PARAM (séptimo if)

Si no entro en ninguno de los anteriores ifs, entra al else donde se espera algún tipo de operación (lógica o aritmética) o una asignación, dependiendo del índice 0 del cuádruplo. En el caso de las operaciones, el proceso de ejecución es el siguiente:

1. Se ejecuta la operación entre el índice 1 y 2 del cuádruplo
2. El índice 3 se empuja a QuadStack
3. El valor del índice 3 es reemplazado por el resultado del punto 1
4. El resultado es empujado a ResultStack
5. Se imprime el cuádruplo resuelto

En el caso de la asignación ocurre los siguiente:

1. Se obtiene el tipo de la variable a la que se asigna el valor
2. Se fuerza el parseo de la operación al tipo de la asignación
3. Se asigna el valor
4. Se imprime el cuádruplo resuelto

A continuación se muestra un ejemplo de una ejecución con aritmética con los cuádruplos resolución y los cuádruplos post-solucion

- Cuadruplos pre-solucion:

```
Original Quad: 0, Quad:['GOTO', 1]
Original Quad: 1, Quad:['=', 4, None, 'a1']
Original Quad: 2, Quad:['-', 1, 2, 1006]
Original Quad: 3, Quad:['=', 1006, None, '_x']
Original Quad: 4, Quad:['=', 2.0, None, 'y5']
Original Quad: 5, Quad:['+', '_x', 'a1', 1007]
Original Quad: 6, Quad:['-', 1007, 5, 1008]
Original Quad: 7, Quad:['=', 1008, None, 'c']
Original Quad: 8, Quad:['=', 3, None, 'a1']
Original Quad: 9, Quad:['+', 2.0, 'c', 1009]
Original Quad: 10, Quad:['=', 1009, None, '_d4d']
Original Quad: 11, Quad:['-', '_d4d', 'y5', 1010]
Original Quad: 12, Quad:['=', 1010, None, 'f']
Original Quad: 13, Quad:['print', 'a1', None, None]
Original Quad: 14, Quad:['print', '_x', None, None]
Original Quad: 15, Quad:['print', 'c', None, None]
Original Quad: 16, Quad:['print', 'y5', None, None]
Original Quad: 17, Quad:['print', '_d4d', None, None]
Original Quad: 18, Quad:['print', 'f', None, None]
```

- Cuadruplos post-solucion

```
Quad:0 Operator:GOTO To:1
Quad:1 Assign:4 Operator:= To:a1
Quad:2 Left:1 Right:2 Operator:- Result:-1
Quad:3 Assign:-1 Operator:= To:_x
Quad:4 Assign:2.0 Operator:= To:y5
Quad:5 Left:-1 Right:4 Operator:+ Result:3
Quad:6 Left:3 Right:5 Operator:- Result:-2
Quad:7 Assign:-2 Operator:= To:c
Quad:8 Assign:3 Operator:= To:a1
Quad:9 Left:2.0 Right:-2 Operator:+ Result:0.0
Quad:10 Assign:0.0 Operator:= To:_d4d
Quad:11 Left:0.0 Right:2.0 Operator:- Result:-2.0
```

Quad:12 Assign:-2.0 Operator:= To:f
Quad:13 print 3
3
Quad:14 print -1
-1
Quad:15 print -2
-2
Quad:16 print 2.0
2.0
Quad:17 print 0.0
0.0
Quad:18 print -2.0
-2.0