Tecnológico de Monterrey

[Diseño de Compiladores - TC 3048]

Documentación final

Lenguaje Team++

Integrantes:

A00819796
Néstor Alejandro Martínez Cárcamo

A01338798
Edgar Rubén Salazar Lugo

02/junio/2021

Índice

Descripción del proyecto

I. Propósito y alcance del proyecto.

Este proyecto tiene como objetivo la creación de de un lenguaje de programación básico que pueda manejar objetos creados por el usuario. Este lenguaje también soporta el uso de arreglos de una dimensión (vectores) y de dos dimensiones (matrices), así como condicionales y ciclos (for y while). El lenguaje está basado en el lenguaje de C++ en cuestión de su gramática y sintaxis.

II. Análisis de requerimientos y descripción de los principales test cases.

A. Requerimientos

Número de Requerimiento	R01
Nombre de Requerimiento	Declaración de Variables
Descripción	El usuario puede declarar variables de tipo entero, flotante, booleano o char. El usuario puede declarar más de una variable de un tipo. La declaración de variables se hace antes del desarrollo de una función (en caso de las variables locales de cada función) o antes de las declaraciones de funciones (en caso de las funciones globales)

Número de Requerimiento	R02
Nombre de Requerimiento	Declaración de Clases
Descripción	El usuario puede crear objetos definidos por el usuario. La declaración de estos se da antes de la declaración de las variables globales y justo después del comienzo del programa.

Número de Requerimiento	R03
Nombre de Requerimiento	Declaración de Funciones
Descripción	El usuario puede crear funciones las cuales contendrán variables locales y podrá recibir parámetros. Las funciones se declaran luego de las variables globales

Número de Requerimiento	R04
Nombre de Requerimiento	Asignación de variables
Descripción	El usuario puede asignar valores a las variables luego de haberlas declarado. Los valores dependen del tipo del cual se declaró la variable. Las asignaciones se pueden hacer dentro del desarrollo de funciones o en el main del programa.

Número de Requerimiento	R05
Nombre de Requerimiento	Asignación de arreglos con indexación
Descripción	El usuario puede asignar valores a cualquier elemento del arreglo usando indexación ya sea con constantes o variables.

Número de Requerimiento	R06
Nombre de Requerimiento	For loop
Descripción	El usuario puede crear un ciclo basado en un contador implícito poniendo los límites dentro de la declaración del loop.

Número de Requerimiento	R07
Nombre de Requerimiento	While loop
Descripción	El usuario puede crear un ciclo basado en un condicional que se revisa cada vez que el bloque dentro del loop termina.

Número de Requerimiento	R08
Nombre de Requerimiento	Condicionales
Descripción	El usuario puede crear condicionales basadas en una expresión booleana.

Número de Requerimiento	R09
Nombre de Requerimiento	Operaciones con arreglos
Descripción	El usuario puede usar arreglos con indexación dentro de operaciones para generar un valor. Es decir, en vez de asignarle el valor de una posición del arreglo, se puede usar directamente el arreglo con la posición.

Número de Requerimiento	R10
Nombre de Requerimiento	Escribe
Descripción	El usuario puede desplegar mensajes con variables o constantes usando la función de escribe.

Número de Requerimiento	R11
Nombre de Requerimiento	Lee
Descripción	El usuario puede ingresar valores desde

	el teclado usando la función de lee.

B. Casos de Uso

Test Case	Input	Expected Result	Actual Result	Pass
Llenar arreglo	**1, 2, 3, 4, 5**	**Vector lleno 1 2 3 4 5**	**Vector lleno 1 2 3 4 5**	**Si**
Llenar matriz	**-**	**0 1 2 3 4 5 6 7 8**	**0 1 2 3 4 5 6 7 8**	**Si**
Fibonacci	**-**	**“Fibonacci iterativo:” 13 “Fibonacci Recursivo:” 21**	**“Fibonacci iterativo:” 13 “Fibonacci Recursivo:” 21**	**Si**
Factorial	**-**	**“Factorial iterativo” 120 “Factorial recursivo” 720**	**“Factorial iterativo” 120 “Factorial recursivo” 720**	**Si**
Bubble Sort		**“Arreglo desordenado” 5 2 9 3 1**	**“Arreglo desordenado” 5 2 9 3 1**	**Si**

		10 **“BUBBLE SORT”** **“Arreglo ordenado”** **1** **2** **3** **5** **9** **10**	**10** **“BUBBLE SORT”** **“Arreglo ordenado”** **1** **2** **3** **5** **9** **10**	
Quick Sort	**-**	**"Arreglo desordenado"** **5** **2** **9** **3** **1** **10** **"Arreglo ordenado"** **1** **2** **3** **5** **9** **10**	**"Arreglo desordenado"** **5** **2** **9** **3** **1** **10** **"Arreglo ordenado"** **1** **2** **3** **5** **9** **10**	**Si**
Find Value	**2** **4**	**“Lo encontré”** **“no existe”**	**“Lo encontré”** **“no existe”**	**Si** **Si**

III. Descripción del proceso general.

A. Descripción del Proceso

Durante la semana nos juntamos dos o tres veces para avanzar en el proyecto. Hacíamos lluvia de ideas para poder crear las estructuras necesarias y trabajabamos en una sola computadora debido a que liveserver de vs code no nos funcionó. Luego de hacer las cosas que nos parecían más importantes de cada avance, terminabamos la sesión de trabajo y nos dividiamos los detalles o bugs que necesitaran arreglo.

Cuando estábamos trabajando en tareas diferentes, manejamos diferentes ramas, luego en las sesiones nos juntabamos a unir nuestros avances y seguir con los avances necesarios. Durante las sesiones, tratamos de cubrir la mayor cantidad de funcionalidades posibles que pidiera cada avance. También nos juntabamos si surgía alguna duda de cómo íbamos a implementar alguna función o método.

B. Lista de Commits de la rama main

https://github.com/Chito-XD/proyectoCompiladores/commits/main

C. Bitácora

https://github.com/Chito-XD/proyectoCompiladores#readme

D. Párrafos de reflexión

1. **Ruben:** Personalmente, el aprendizaje que más me llevo de este proyecto es entender de mejor manera cómo es que el lenguaje trabaja a bajo nivel. Saber que todas las abstracciones que uno puede hacer en algún lenguaje conlleva costo computacional y por ende, debe de ser eficiente en tiempo y recursos. Así que, cada vez que programe, haga una nueva línea de código, sabré que eso afecta directamente a la eficiencia de mi programa.

2. **Néstor:** En lo personal, este fue el proyecto más retador que he tenido en toda la carrera. Fue un proyecto bastante interesante y que te ayuda a pensar que hay detrás de todo programa. Me hizo pensar lo complicado que es crear un lenguaje por más sencillo que sea y me enseñó a apreciar los lenguajes actuales, especialmente python que es muy user friendly y bastante fácil de usar. Me imagino las dificultades que los desarrolladores tuvieron que pasar para poder crear un lenguaje de programación como python y me hace apreciarlo más. Creo que es un muy buen proyecto, bastante interesante pero al mismo tiempo bastante pesado.

Descripción del lenguaje

I. **Nombre del lenguaje**.
Lenguaje Team++ 5

II. Descripción genérica de las principales características del lenguaje.
Nuestro lenguaje estaba basado en la misma estructura de C + +, por lo que se puede decir que es un lenguaje orientado a objetos. La estructura de nuestro lenguaje es la siguiente:

- Nombre del programa
- Declaración de clases
- Declaración de variables globales
- Declaración de funciones
- Main

Al ser un lenguaje de programación orientado a objetos, es implícito que soporta la creación de objetos/clases creadas por el usuario. En este lenguaje se tiene que seguir de manera estricta la estructura antes mencionada, es decir, la declaración de variables globales va después de la declaración de clases , la declaración de funciones va después de la declaración de variables globales y el main va de último después de todas las declaraciones anteriores.

Como cualquier lenguaje de programación, el nuestro soporta la creación de variables de diferentes tipos, para ser específicos de tipo *Entero, flotante, booleano* y *char*. Así como también soporta constantes *enteras, flotantes, booleano* y *string*.

También, nuestro lenguaje de programación soporta la creación de arreglos bi y tridimensionales. Soporta las operaciones con arreglos usando indexación ya sea con constantes, expresiones o variables. También hace verificaciones para saber que el índice está dentro de los límites establecidos.

Nuestro lenguaje también soporta funciones con y sin valores de retorno. También verifica el número de parámetros en las llamadas para que coincidan con el número de parámetros con las que se declararon las funciones.

III. Listado de errores.

A. El programa siempre tiene que empezar con la palabra *programa*, si no manda error de sintaxis.
B. El programa tiene que seguir la estructura determinada, si no regresa error de gramática.

C. Las variables tienen que estar declaradas ya sea de manera local o de manera global para poder usarlas.
D. Las variables locales sólo pueden ser usadas dentro de las funciones, si se trata de usar una variable local en otra función arroja error de semántica.
E. Las variables sólo pueden recibir valores del tipo del que fueron definidas, si no reciben un valor del mismo tipo, arroja un error semántica.
F. Si una función recibe un número diferente de parámetros, el programa arroja un error de semántica.
G. Si se operan variables no compatibles según el cubo semántico, el programa arroja un error de semántica.
H. Si se omiten los punto y coma al final de cada comando, se arroja un error de sintaxis.
I. Si se omiten los corchetes en un bloque de función o del main, se arroja un error de sintaxis.
J. Si se omiten los paréntesis en las condiciones de las condicionales y los loops, se arroja un error de sintaxis.
K. Si no se declaran primero las variables y luego los tipos, arroja un error de sintaxis.
L. Si el valor de índice está fuera del rango de los arreglos, se tira un error en compilación de fuera de rango.

Descripción del compilador

I. Equipo de cómputo, lenguaje y utilerías especiales.
    A. Equipo de cómputo: máquina propia.
    B. Lenguaje: Python
    C. Utilerias Especiales:
        1. Biblioteca SLY de Python
II. Descripción del análisis de léxico.
    A. Tokens:
        1. PROGRAMA : “programa”
        2. CLASE : “Clase”
        3. HEREDA : “hereda”
        4. ATRIBUTOS : “atributos”
        5. METODOS : “metodos”
        6. PRINCIPAL : “principal”
        7. VARIABLES : “variables”
        8. ENTERO : “entero”
        9. FLOTANTE : “flotante”

10. CHAR : “char”
11. VOID : “void
12. BOOLEAN : “booleano
13. FUNCION : “funcion”
14. LEE : “lee”
15. REGRESA : “regresa”
16. ESCRIBE : “escribe”
17. MIENTRAS : “mientras”
18. HACER : “hacer”
19. DESDE : “desde”
20. HASTA : “hasta”
21. SI : “si”
22. ENTONCES : “entonces”
23. SINO : “sino”
24. ID : identificador
25. SEMICOLON : “ ; ”
26. COMMA : “ , ”
27. LP : (
28. RP :)
29. LK : {
30. RK: }
31. ASSIGN: =
32. OP_REL: <> | < | > | <= | >= | ==
33. OP_ARIT_SEC: + | -
34. OP_ARIT_PRIM: * | /
35. OP_LOG: & | |
36. CTE_F: número flotante
37. CTE_I: número entero
38. CTE_STRING: cadena de texto
39. DOTS: “ : ”
40. DOT: “ . ”
41. LC [
42. RC]

B. Expresiones Regulares

```
ID = r'[a-zA-Z_][a-zA-Z_0-9]*'

OP_REL = r'(<>|<=|>=|<|>|(==)|!=)'
SEMICOLON = r'\;'
COMMA = r'\,'
LP = r'\('
RP = r'\)'
```

```
LC = r'\['
RC = r'\]'
LK = r'\{'
RK = r'\}'
ASSIGN = r'\='


OP_ARIT_SEC = r'(\+|-)'
OP_ARIT_PRIM = r'(\*|\/)'
OP_LOG = r'(&|\|)'

CTE_F = r'\-?([0-9]+)(\.)([0-9]+)'
CTE_I = r'\-?[0-9]+'
CTE_STRING = r'\"(\w| )*\"'
DOTS = r'\:'
DOT = r'\.'
```

III. Descripción del análisis de sintaxis.

Nuestra gramática formal para representar las estructuras sintácticas fue de la siguiente manera:

```
PROGRAMA -> Programa id ; DEC_CLAS DEC_VARS DEC_FUNC PRINCIPAL
         -> Programa id ; DEC_CLAS DEC_VARS PRINCIPAL
         -> Programa id ; DEC_CLAS DEC_FUNC PRINCIPAL
         -> Programa id ; DEC_CLAS PRINCIPAL
         -> Programa id ; DEC_VARS DEC_FUNC PRINCIPAL
         -> Programa id ; DEC_VARS PRINCIPAL
         -> Programa id ; DEC_FUNC PRINCIPAL
         -> Programa id ; PRINCIPAL

PRINCIPAL -> principal () BLOQUE

DEC_CLAS -> Clase id hereda id { ATRIBUTOS METODOS }
         -> Clase id hereda id { ATRIBUTOS }
         -> Clase id hereda id { METODOS }
         -> Clase id hereda id { }
         -> Clase id { ATRIBUTOS METODOS }
         -> Clase id { ATRIBUTOS }
         -> Clase id { METODOS }
         -> Clase id { }
         -> Clase id hereda id { ATRIBUTOS METODOS } DEC_CLAS
         -> Clase id hereda id { ATRIBUTOS } DEC_CLAS
         -> Clase id hereda id { METODOS } DEC_CLAS
         -> Clase id hereda id { } DEC_CLAS
         -> Clase id { ATRIBUTOS METODOS } DEC_CLAS
         -> Clase id { ATRIBUTOS } DEC_CLAS
         -> Clase id { METODOS } DEC_CLAS
         -> Clase id { } DEC_CLAS
```

```
ATRIBUTOS -> atributos : DEC_VARS_AUX

METODOS -> metodos : dec_func

DEC_VARS -> variables : dec_vars_aux

DEC_CLAS_AUX -> dec_var_aux2 : TIPO_SIMPLE ; dec_vars_aux
             -> dec_var_aux2 : TIPO_SIMPLE ;
             -> dec_var_aux2 : TIPO_COMPUESTO ; dec_vars_aux
             -> dec_var_aux2 : TIPO_COMPUESTO ;

DEV_VARS_AUX2 -> VAR , DEV_VARS_AUX2
              -> VAR

VAR -> id [ cte_i ] [ cte_i ]
    -> id [ cte_i ]
    -> id

PARAMETROS -> ID : TIPO_SIMPLE , PARAMETROS
           -> ID : TIPO_SIMPLE
```

```
DEC_FUNC -> TIPO_SIMPLE funcion id ( PARAMETROS ) ; DEC_VARS BLOQUE
         -> TIPO_SIMPLE funcion id ( PARAMETROS ) ; BLOQUE
         -> TIPO_SIMPLE funcion id ( PARAMETROS ) ; DEC_VARS BLOQUE DEC_FUNC
         -> TIPO_SIMPLE funcion id ( PARAMETROS ) ; BLOQUE DEC_FUNC
         -> TIPO_SIMPLE funcion id ( ) ; DEC_VARS BLOQUE
         -> TIPO_SIMPLE funcion id ( ) ; BLOQUE
         -> TIPO_SIMPLE funcion id ( ) DEC_VARS BLOQUE DEC_FUNC
         -> TIPO_SIMPLE funcion id ( ) ; BLOQUE DEC_FUNC
         -> VOID funcion id ( PARAMETROS ) ; DEC_VARS BLOQUE
         -> VOID funcion id ( PARAMETROS ) ; BLOQUE
         -> VOID funcion id ( PARAMETROS ) ; DEC_VARS BLOQUE DEC_FUNC
         -> VOID funcion id ( PARAMETROS ) ; BLOQUE DEC_FUNC
         -> VOID funcion id ( ) ; DEC_VARS BLOQUE
         -> VOID funcion id ( ) ; BLOQUE
         -> VOID funcion id ( ) DEC_VARS BLOQUE DEC_FUNC
         -> VOID funcion id ( ) ; BLOQUE DEC_FUNC

BLOQUE -> { BLOQUE_AUX }
       -> { }

BLOQUE_AUX -> ESTATUTO BLOQUE_AUX
           -> ESTATUTO
```

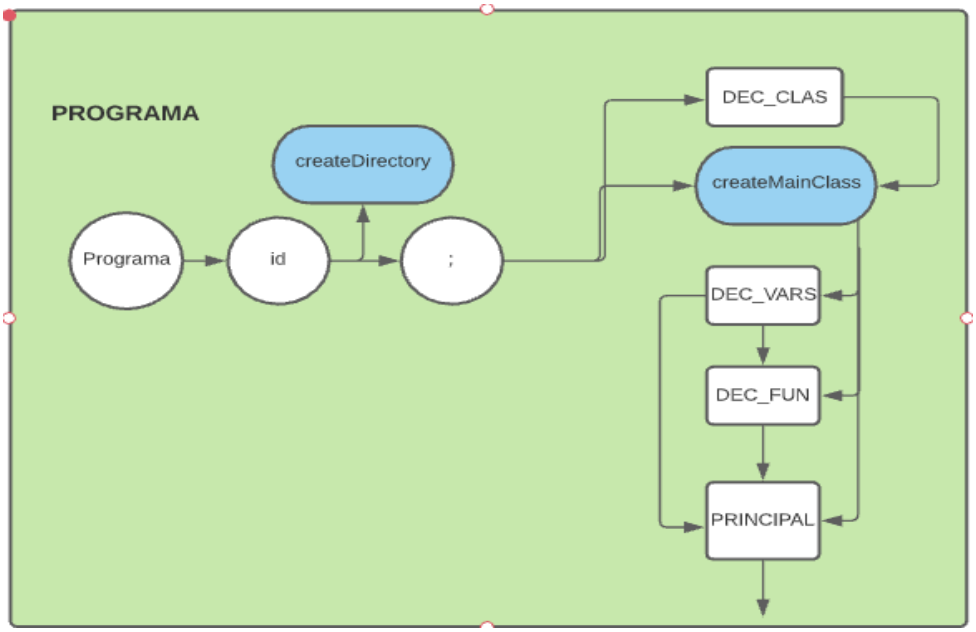
```
ESTATUTO -> FUNCION ;
         -> ASIGNACION
         -> RETORNO
         -> ESCRITURA
         -> LECTURA
         -> DECISION
         -> REPETICION

ASIGNACION -> VARIABLE = SUPER_EXP ;

FUNCION -> id ( FUNCION_AUX )
        -> id ( )

FUNCION_AUX -> SUPER_EXP , FUNCION_AUX
            -> SUPER_EXP

RETORNO -> regresa ( SUPER_EXP ) ;

ESCRITURA -> escribe ( escritura_aux ) ;
```

```
ESCRITURA_AUX -> SUPER_EXP , ESCRITURA_AUX
              -> SUPER_EXP

LECTURA -> lee ( lectura_aux ) ;

LECTURA_AUX -> variable , LECTURA_AUX
            -> variable

LECTURA_AUX_VARS -> CLASS_VAR
LECTURA_AUX_VARS -> ID

DECISION -> si ( SUPER_EXP ) entonces BLOQUE sino BLOQUE
         -> si ( SUPER_EXP ) entonces BLOQUE

REPETICION -> mientras ( SUPER_EXP ) hacer BLOQUE
REPETICION -> desde id = SUPER_EXP  hasta  SUPER_EXP hacer BLOQUE

TIPO_SIMPLE -> ENTERO
            -> FLOTANTE
            -> CHAR
            -> BOOLEAN

TIPO_COMPUESTO -> id
```

```
SUPER_EXP -> EXPRESION OP_LOG SUPER_EXP
          -> EXPRESION

EXPRESION -> EXP OP_REL EXPRESION
          -> EXP

EXP -> TERMINO OP_ARIT_SEC EXP
    -> TERMINO

TERMINO -> FACTOR OP_ARIT_PRIM TERMINO
        -> FACTOR

FACTOR -> LP SUPER_EXP RP
       -> VAR_CTE
       -> VARIABLE
       -> FUNCION
```

```
VAR_CTE -> cte_i
        -> cte_f
        -> cte_string

VARIABLE -> id [ super_exp ] [ super_exp ]
         -> id [ super_exp ]
         -> id . id ( VARIABLE_UAX )
         -> id . id
         -> id
```

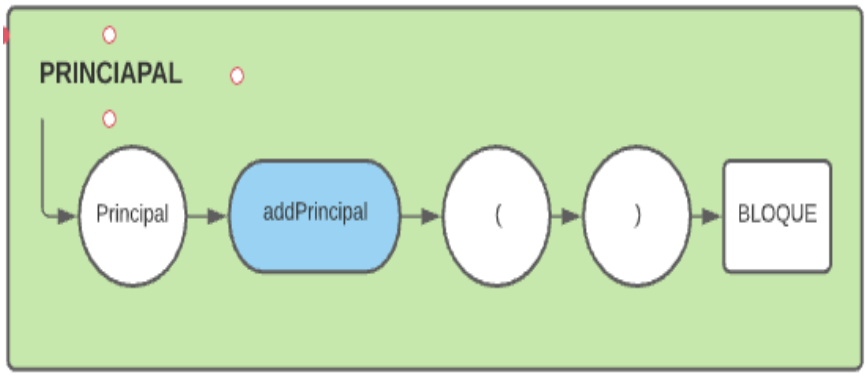
IV. Descripción de generación de código intermedio y análisis semántico.



PROGRAMA:

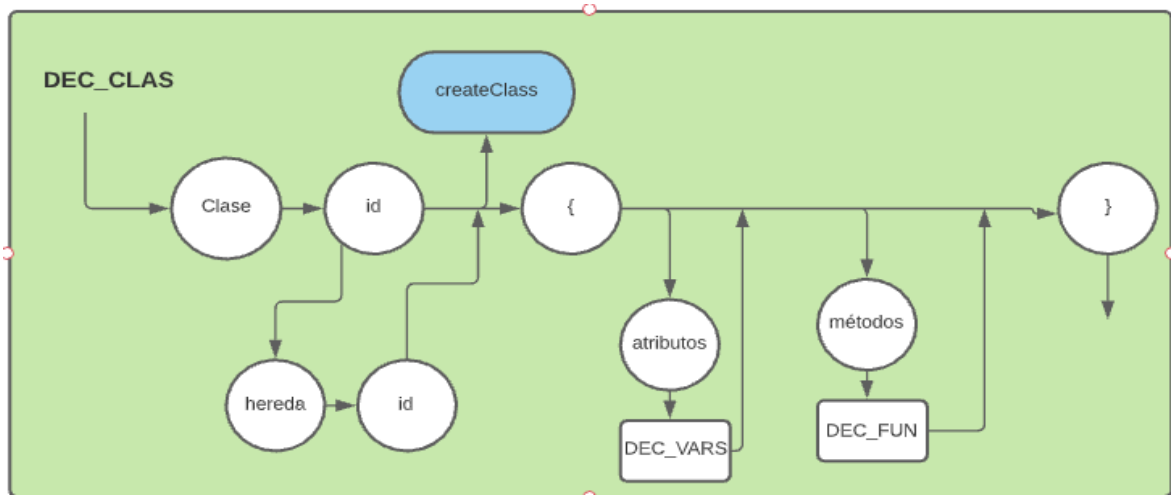
createDirectory: crea el primer cuádruplo que corresponde a un *GOTO* al main.

createMainClass: configura que se está compilando en la clase y función main y al final agrega la función al directorio de funciones.



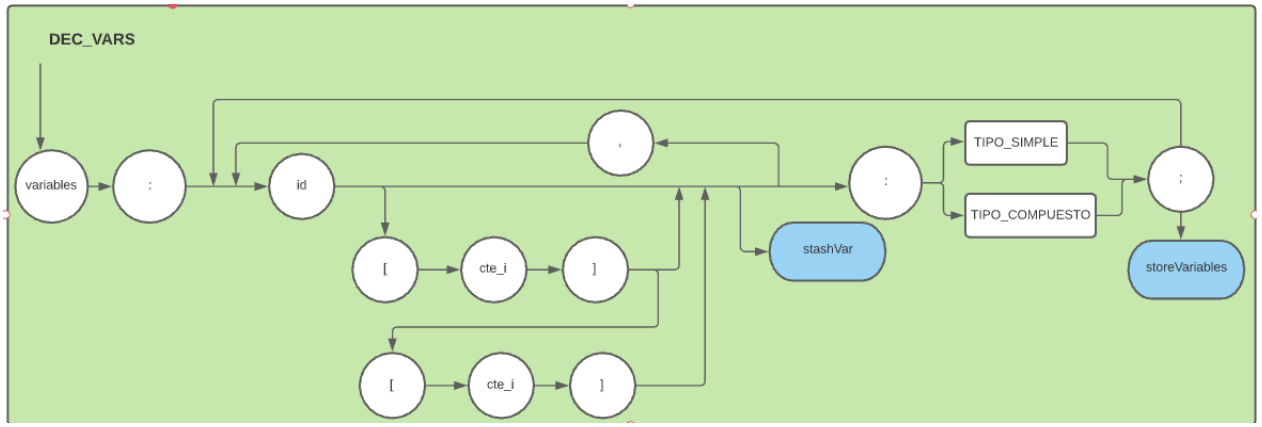
PRINCIPAL:

addPrincipal: configura que se está compilando en la función principal, añade la función al directorio de funciones y rellena el *GOTO* del main al contador de cuádruplos



DEC_CLAS:

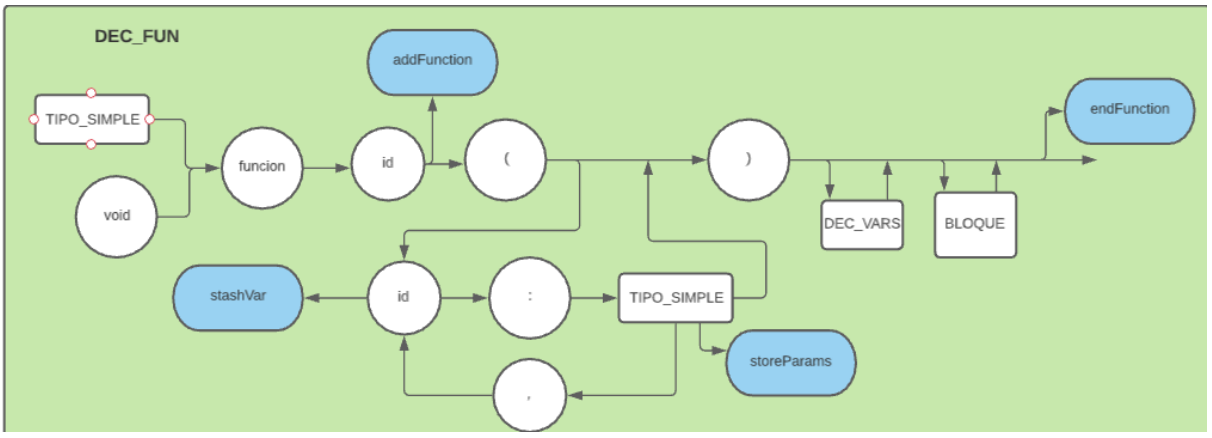
createClass: configura que se está compilando en la clase y función del nombre de la clase dada, crear la clase y función al directorio de funciones.



DEC_VARS:

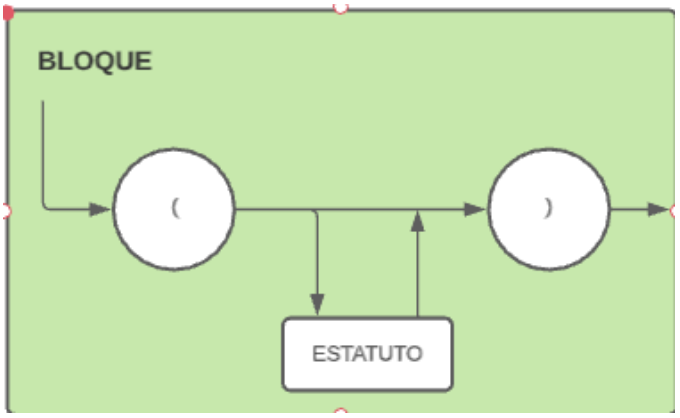
stashVar: guarda la variable en un stack de variables

storeVariables: recorre el stack de variables para guardarlas en la tabla de variables de la clase y función en la que se esté compilando.



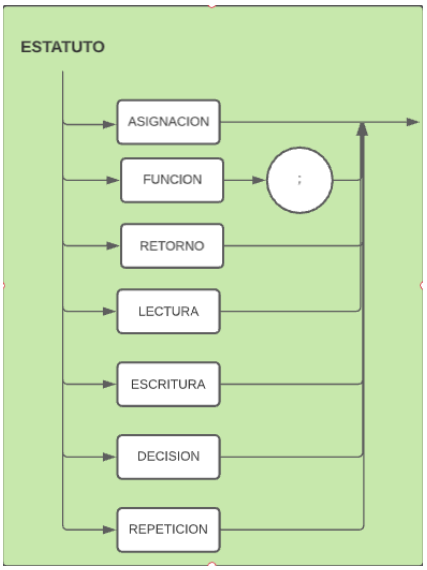
DEC_FUN:

addFunction: setea el método en el que se esté compilando y da de alta la función en el directorio de funciones.
stashVar: guarda en un stack los parámetros.
storeParams: guarda en la tabla de variables del método y clase en la que se esté compilando los parámetros del stack.
endFunction: crea cuádruplo de *END_FUNCTION*.



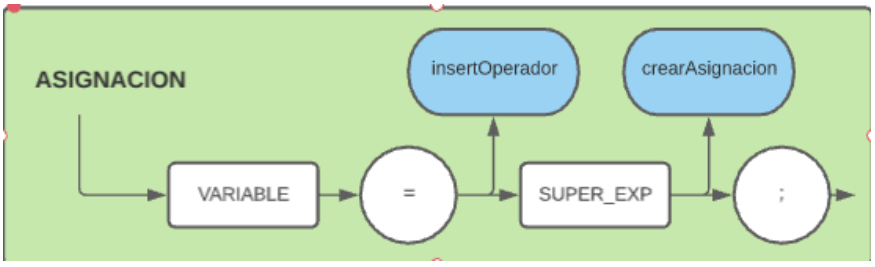
BLOQUE

Sin puntos neurálgicos.



ESTATUTO:

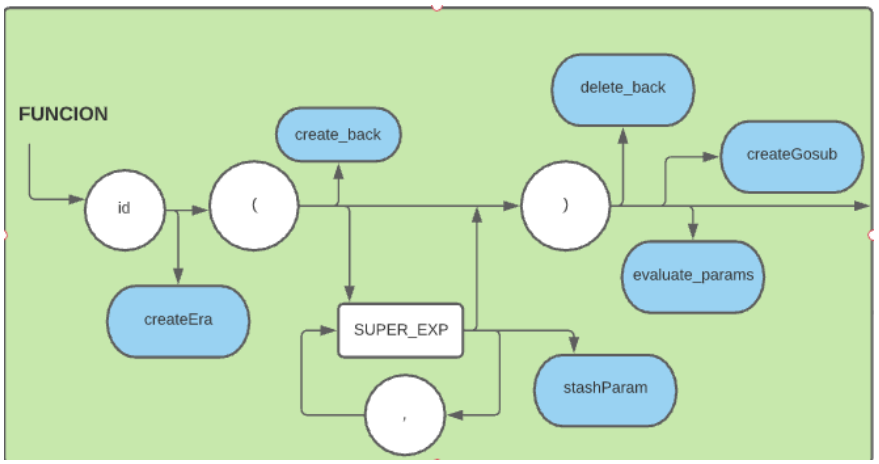
Sin puntos neurálgicos.



ASIGNACION:

insertOperador: agregar a la pila de operadores el signo de igual.

crearAsignación: obtiene los últimos dos operadores, evalúa sus tipos con base en el cubo semántico y de ser correcto, crea el cuádruplo de asignación.



FUNCION:

createEra: añadir a la pila de métodos llamados el id del método y crear el cuádruplo de *ERA*.

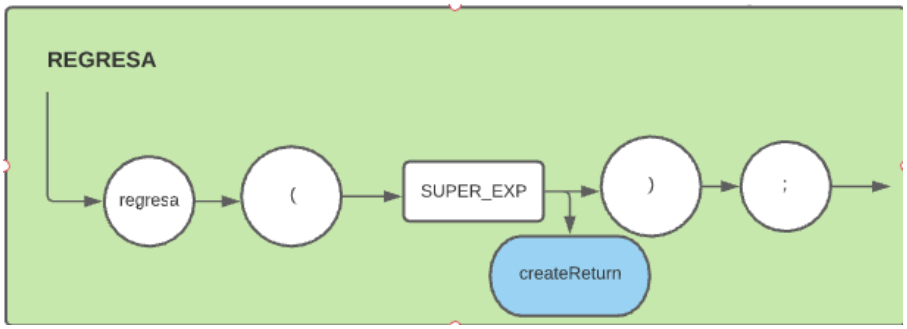
create_back: meter a la pila de operadores el fondo falso.

delete_back: remover de la pila de operadores el fondo falso

createGosub: crear el cuádruplo de gosub con base en si fue llamada de clase de función o función sola. Revisa el tipo de retorno de la función para agregar el "parche".

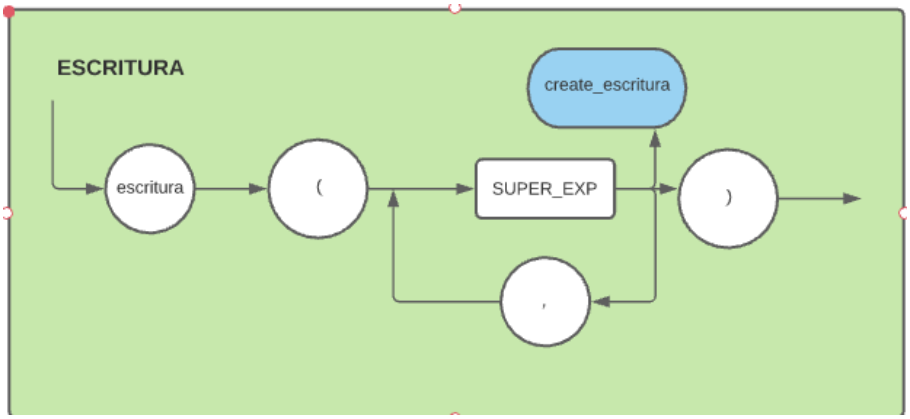
stashParam: agregar al stack de parámetros el último operando.

evaluate_params: valida la cantidad de parámetros recibidos con los declarados al igual que el tipo de los mismos.



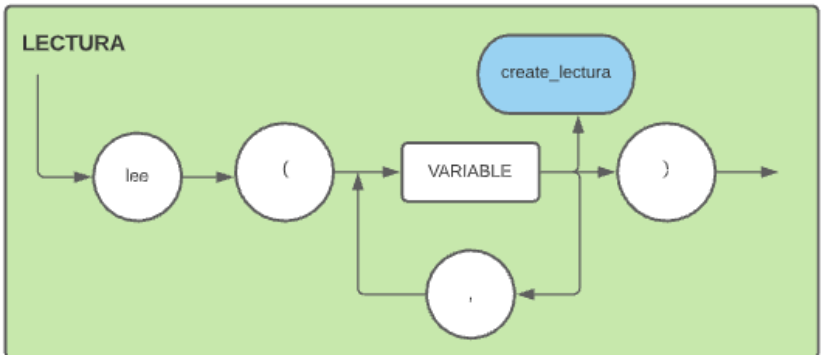
REGRESA:

createReturn: valida el tipo de retorno declarado con el tipo de operando a regresar y si es el mimso, crea el cuádruplo de regresar.



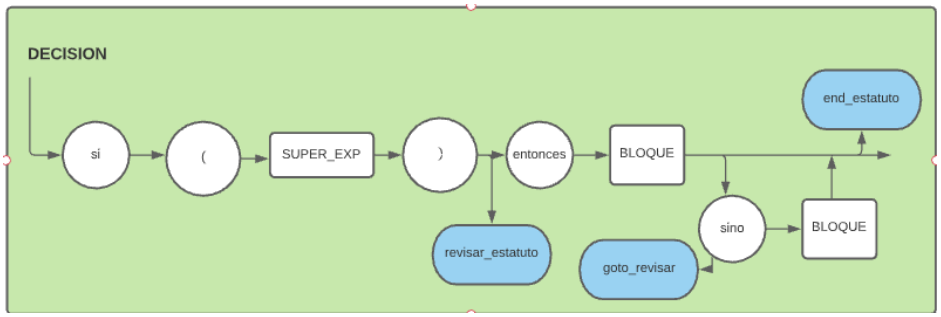
ESCRITURA:

create_escritura: obtiene el último operando y crea el cuádruplo de escritura.



LECTURA:

create_lectura: obtiene el último operando y crea el cuádruplo de lectura.

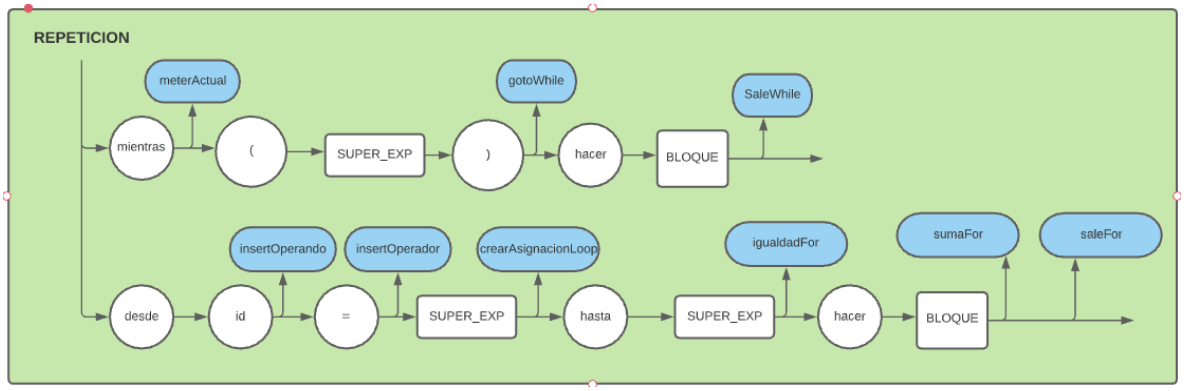


DECISION:

revisar_estatuto: obtiene el último operando, crea el cuádruplo de *GotoF* y deja la "migaja de pan".

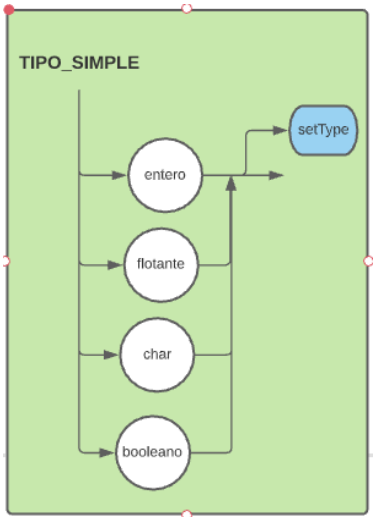
goto_revisar: crea cuádruplo de *Goto,* deja la "migaja de pan" y rellena el falso con el contador.

end_estatuto: rellena fin con el contador.



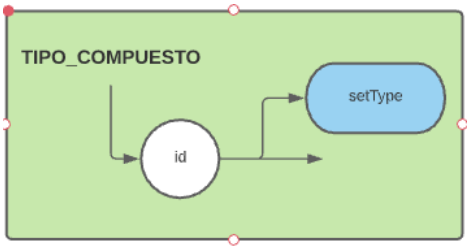
REPETICIÓN:

meterActual: meter el contador a la pila de saltos
gotoWhile: generar el cuadruplo de *GotoF* con base en la condición y meter a la pila de saltos el contador menos 1.
SaleWhile: generar el *GotoF* con el return y rellenar el falso con el contador
insertOperando: agregar a la pila de operadores el id
insertOperador: agregar a la pila de operadores el =
crearAsignacionLoop: crear cuádruplo de asignación
igualdadFor: verifica que la asignación cumpla las condiciones del la super_exp
sumaFor: suma un valor de uno a la variable de asignación.
saleFot: Genera el cuádruplo de *Goto* para revisar nuevamete la super_exp



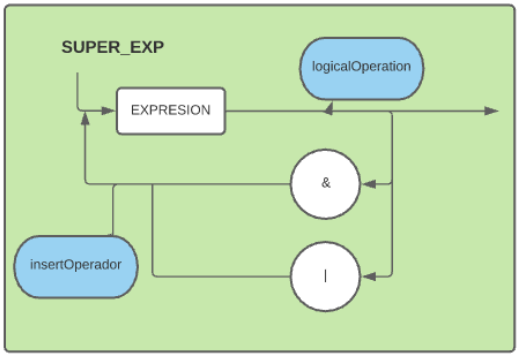
TIPO_SIMPLE:

setType: setea el tipo actual.



TIPO_COMPUESTO:

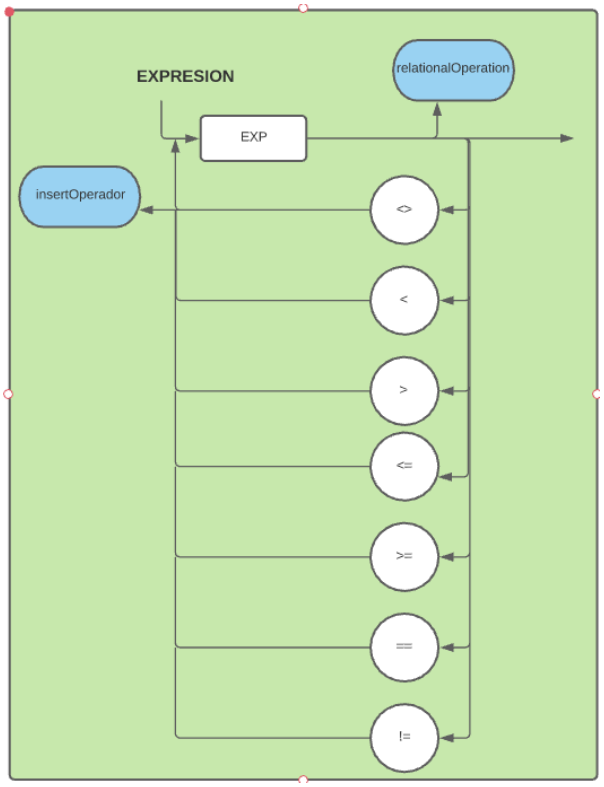
setType: setea el tipo actual.



SUPER_EXP:

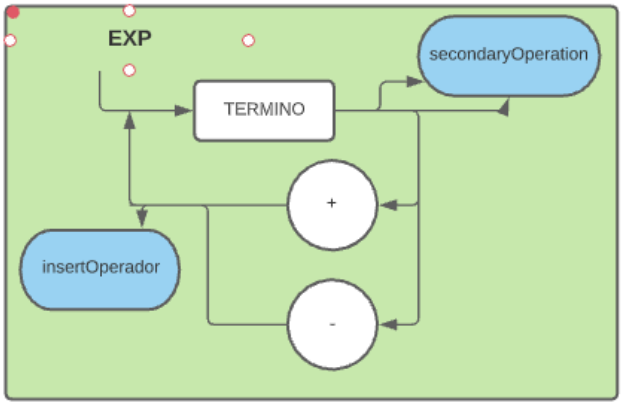
insertOperador: agrega a la pila de operadores

logicalOperation: revisa si en el top de la pila de operadores es un operador lógico. En caso positivo, se ejecuta la operación.



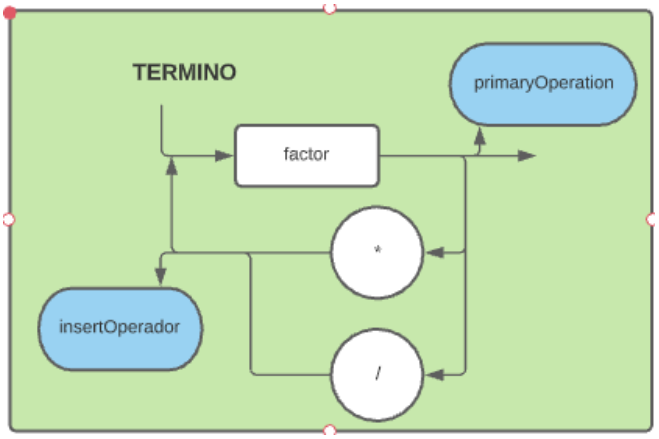
EXPRESION:

insertOperador: agrega a la pila de operadores
relationalOperation: revisa si en el top de la pila de operadores es un operador relacional. Si sí, ejecuta la operación.



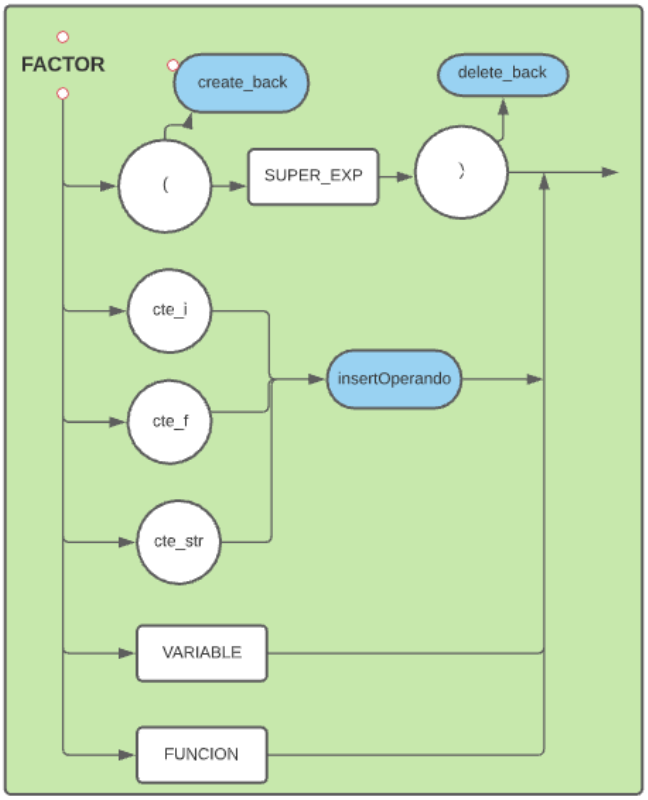
EXP:

insertOperador: agrega a la pila de operadores
secondaryOperation: revisa si en el top de la pila de operadores es un operador aritmético de suma o resta. Si sí, ejecuta la operación.



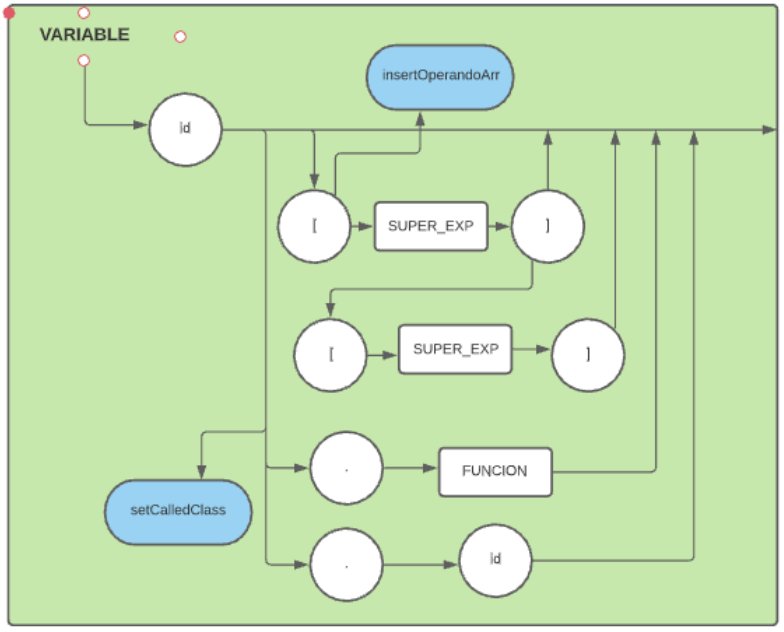
TERMINO:

insertOperador: agrega a la pila de operadores
primaryOperation: revisa si en el top de la pila de operadores es un operador aritmético de multiplicación o división. Si sí, ejecuta la operación.



FACTOR:

create_back: agrega la pila de operadores el fondo falso.
delete_back: quita de la pila de operadores el fondo falso.
insertOperando: Añadir a la pila de operandos.



VARIABLE:

insertOperandoArr: crea un fondo falso y agrega a la pila de operandos el id
setCalledClass: añadir al stack de clases llamadas el nombre de la clase

Cubo semántico:

Diccionario de python (*TYPE_MATCHING*) con una profundidad de tres llaves. La primera y segunda llave hacen referencia a cada operador, mientras que la tercera llave es el tipo de operación que se quiere ejecutar (aritmética, lógica, relacional o de asignación). Como valor final está el tipo resultante de dicha operación.

Los vales de cada operación están como constantes y sus valores son los siguientes:

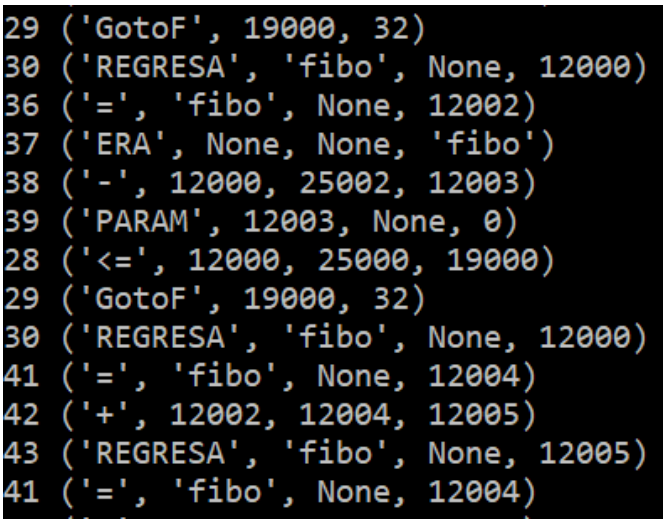
- ARIT_PRIM: operaciones aritméticas de multiplicación y división.
- ARIT_SEC: operaciones aritméticas de suma y resta.
- ARIT_LOG: operaciones lógicas.
- ARIT_REL: operaciones relacionales.
- ASSIGN: operación de asignación.

```
TYPE_MATCHING = {
    ENTERO: {
        ENTERO: {
            ARIT_PRIM : ENTERO,
            ARIT_SEC : ENTERO,
            ARIT_LOG : ERROR,
            ARIT_REL : BOOLEANO,
            ASSIGN : ENTERO
        },
        FLOTANTE: {
            ARIT_PRIM : FLOTANTE,
            ARIT_SEC : FLOTANTE,
            ARIT_LOG : ERROR,
            ARIT_REL : BOOLEANO,
            ASSIGN : ERROR
        },
        VOID: {
            ARIT_PRIM : ERROR,
            ARIT_SEC : ERROR,
            ARIT_LOG : BOOLEANO,
            ARIT_REL : ERROR,
            ASSIGN: ERROR,
        },
        BOOLEANO: {
            ARIT_PRIM : ERROR,
            ARIT_SEC : ERROR,
            ARIT_LOG : ERROR,
            ARIT_REL : ERROR,
            ASSIGN : ERROR
        },
```

```
FLOTANTE: {
    ENTERO: {
        ARIT_PRIM : FLOTANTE,
        ARIT_SEC : FLOTANTE,
        ARIT_LOG : ERROR,
        ARIT_REL : BOOLEANO,
        ASSIGN : ERROR
    },
    FLOTANTE: {
        ARIT_PRIM : FLOTANTE,
        ARIT_SEC : FLOTANTE,
        ARIT_LOG : ERROR,
        ARIT_REL : BOOLEANO,
        ASSIGN : FLOTANTE
    },
    VOID: {
        ARIT_PRIM : ERROR,
        ARIT_SEC : ERROR,
        ARIT_LOG : ERROR,
        ARIT_REL : ERROR,
        ASSIGN : ERROR
    },
    BOOLEANO: {
        ARIT_PRIM : ERROR,
        ARIT_SEC : ERROR,
        ARIT_LOG : ERROR,
        ARIT_REL : ERROR,
        ASSIGN : ERROR
    },
```

V. Descripción detallada del proceso de administración de memoria.

- **Stack:** Clase de python que contiene solamente un arreglo como atributo. Sus métodos permiten el acceso a los valores por medio de una metodología LIFO.
- **Queue:** Clase de python que contiene solamente un arreglo como atributo. Sus métodos permiten el acceso a los valores por medio de una metodología FIFO.
- **Cuádruplos:** arreglo de tuplas el cual es llenado en el análisis semántico. Cada tupla tiene tres o cuatro elementos dependiendo de la operación a realizar. Para el caso de flujos lineales y operaciones aritméticas, la tupla tiene un tamaño de cuatro. En la primera posición está el operador y en las demás las direcciones de memoria a las que se les va a aplicar dicha operación.
  Para el caso de flujos no lineales, la tupla tiene un tamaño de tres. Como primer posición está el comando a ejecutar, la segunda posición en la condición de la que depende el comando (formato de dirección de memoria). Como última posición, el número del cuádruplo al que debería de moverse si se cumple la condición.



- **Directorio de funciones:** clase que contiene como atributo solamente un diccionario de python.
  Como llave, tiene el nombre de la clase (para la función principal se llama *main*). Esta llave contiene dos valores. El primero es el nombre del método principal de la clase (*proceso_global).* Mientras que el segundo, contiene diccionarios que guardan la información de cada función.
  Para la información de cada función, su llave es el nombre intrínseco de la función y sus valores son:
  - tipo_retorno: tipo simple que puede regresar la función.
  - directorio_variables: objeto de la clase de TabVars() la cual contiene la información de la tabla de variables.
  - inicio: número del cuádruplo en el que inicia la función.
  - parametros: arreglo que contiene los tipos de los parámetros que puede recibir la función.

```
dir: {
    "Calculo": {
        "proceso_global": "Calculo,

        "Calculo": {
            "tipo_retorno": "PROCESO",
            "directorio_variables": TabVars(),
            "inicio": 1,
            "parametros": []
        }
    },
    "Main": {
        "proceso_global": "Main",

        "Main": {
            "tipo_retorno": "PROCESO",
            "directorio_variables": TabVars(),
            "inicio": 8,
            "parametros": []
        },

        "uno": {
            "tipo_retorno": "entero",
            "directorio_variables": TabVars(),
            "inicio": 5,
            "parametros": [entero]
        }
    }
}
```

- **Directorio de variables:** clase contiene como atributo solamente un dicctionario de python. Como llave es el nombre de la variable declarada y sus valores están relacionados a características intrínsecas de la variables. Los valores son los siguientes:

    - tipo: tipo simple o compuesto
    - valor: el valor que contiene la variable
    - direccion: direccion de memoria de acuerdo al rango.
    - dimension: valor nulo o diccionario indicando la cantidad de dimensiones de la variable (variable normal, arreglo o matriz).

```
{
    "f": {
        "tipo": flotante,
        "valor": 1,
        "direccion": 6000,
        "dimension: None
    },
    "e": {
        "tipo": entero,
        "valor": 1,
        "direccion": 5000,
        "dimension: {
            "d1": 2,
            "d2": 2
        }
    }
}
```

- **Memoria:** clase que maneja la cantidad de variables para cada método y clase que se esté compilando. De tal manera, funcionaba como un "despachador" de direcciones de memoria al tener un conteo dependiendo del *scope*.
  La clase maneja tres variables principales en la que está el tipo de variable y el rango de direcciones.
  El diccionario de variables globales se reinicia cada vez que se detecta que se está compilando una nueva clase. Por el lado de las variables locales, éstas se reinician cada vez que se detecte un nuevo método a guardar en el directorio de funciones. Por último, el diccionario de constantes, jamás se reinicia.

```
LOCAL_SPACE_ADDRESS = {
    ENTERO: {
        "min": 12000,
        "max": 14999,
        "current": 12000
    },
    FLOTANTE: {
        "min": 13000,
        "max": 15999,
        "current": 13000
    },
```

```
GLOBAL_SPACE_ADDRESS = {
    ENTERO: {
        "min": 5000,
        "max": 5999,
        "current": 5000
    },
    FLOTANTE: {
        "min": 6000,
        "max": 6999,
        "current": 6000
    },
```

```
CTE_SPACE_ADDRESS = {
    ENTERO: {
        "min": 25000,
        "max": 27999,
        "current": 25000
    },
    FLOTANTE: {
        "min": 28000,
        "max": 30999,
        "current": 28000
    },
```

Descripción de la máquina virtual

I. Equipo de cómputo, lenguaje y utilerías especiales usadas.
Proceso que corre en la terminal de la computadora. Está programado en Python, el mismo lenguaje que nuestro analizador léxico y parser. Los tres programas están unidos en un archivo principal (main.py).
Para poder ejecutarlo, sólo necesita la dirección del archivo a probar (tests/archivo1.txt).
No utiliza ninguna librería especial.

II. Descripción detallada del proceso de administración de memoria en ejecución.

Como se describió anteriormente, existen tres variables para cada *scope* con un rango de direcciones que en ningún momento se empalman. Esas direcciones son asignadas para las variables (globales y locales), temporales (globales y locales) y constante. Éstas son guardadas en la tabla de variables de cada función al momento de ejecución.

Para poder ahorrar espacio al momento de ejecución, no se guardó el tamaño de la función. Esto surge gracias a los flujos no lineales puesto que, dentro de ellos se generan variables temporales que podrían o no ocupar memoria. Para ello, se creó una clase la cual contiene tres atributos. El primero es un stack de memoria para las variables locales y funciones que se vayan invocando. El segundo, un diccionario simple para la memoria global en la que la llave es la dirección de memoria y su valor. El último atributo, es para las variables constantes que funciona similar a las variables globales ya que, la llave es la dirección de memoria y su valor el resultado.

- stack de memoria:

Cada elemento de la memoria del stack es un diccionario simple el cual tiene como llave las direcciones de memoria, apuntadores o direcciones base; como valor del diccionario contiene el valor al cual está relacionado esa dirección de memoria.

Cuando se crea un nuevo método, se agrega un diccionario vacío al stack de memoria, sin direcciones. Al momento de realizar alguna operación aritmética, igualación, ciclo o llamada a función; es cuando verifica si existe la dirección del cuádruplo. En caso de que no, agrega la dirección y valor al stack actual; en caso contrario, se devuelve o re asigna el valor. De esta manera, se evita reservar el espacio para variables temporales que posiblemente no se vayan a utilizar.

Al momento de terminar la llamada a función, se elimina el último elemento del stack. De esta manera simulamos el “matar”, “despertar” y “dormir” la memoria.

```
STACK
[
    {
        12000: 120,
        12001: 720,
        12002: 13
    },
    {
        12000: 8,
        19000: False,
        12001: 7
    }
]
```

- diccionario de memoria global:

Simple diccionario de python en la que la llave es la dirección de memoria y el valor del diccionario es el valor calculado. Esta memoria no

```
GLOBAL
{
    5000: 6,
    5001: 5
    "factIterative": 120,
    "fact": 720
}
```

- diccionario de constantes:

Diccionario que sirve para todas las constantes del programa independientemente del tipo. Si bien la llave es un entero, el valor puede ser cualquier tipo simple que se haya declarado (entero, flotante, char, booleano).

```
CTE_MEMORY
{
    25001: 1,
    25002: 2,
    25003: 0,
    31000: "Factorial iterativo",
    25004: 5
}
```

Pruebas del funcionamiento del lenguaje

I. Pruebas del funcionamiento del proyecto.

- Arreglo ordenado por un *bubbleSort*

```
programa pruebaPrograma ;

variables:
    k[6]: entero;
    i, j, n, temp: entero;

void funcion bubbleSort()
{
    desde i = 0 hasta n-1 hacer {
        desde j = 0 hasta n-i-1 hacer {
            si ( k[j] > k[j+1] ) entonces {
                temp = k[j];
                k[j] = k[j+1];
                k[j+1] = temp;
            }
        }
    }
}

principal () {

    n = 5;

    k[0] = 5;
    k[1] = 2;
    k[2] = 9;
    k[3] = 3;
    k[4] = 1;
    k[5] = 10;

    escribe("Arreglo desordenado");
    desde i = 0 hasta n hacer {
        escribe(k[i]);
    }

    escribe("BUBBLE SORT");
    bubbleSort();

    escribe("Arreglo ordenado");
    desde i = 0 hasta n hacer {
        escribe(k[i]);
    }

}
```

```
tests/bubbleSort.txt
"Arreglo desordenado"
5
2
9
3
1
10
"BUBBLE SORT"
"Arreglo ordenado"
1
2
3
5
9
10
```

- Ejecutar funciones de factorial y binacci (iterativo y recursivo).

```
entero funcion factIterative (n: entero)
variables:
    f : entero;
{
    si (n <= 1) entonces {
        regresa(1);
    } sino {
        f = 1;
        desde i = 1 hasta n hacer {
            f = f * i;
        }
        regresa(f);
    }
}

entero funcion fibo (n: entero)
{
    si (n <= 1) entonces {
        regresa(n);
    } sino {
        regresa(fibo(n-1) + fibo(n-2));
    }
}

entero funcion fiboIterative(n: entero)
variables:
    a, b, c: entero;
{
    a = 0;
    b = 1;
    mientras (n > 1) hacer {
        c = a + b;
        a = b;
        b = c;
        n = n-1;
    }
    regresa(c);
}

principal () {
    escribe("Factorial iterativo" , factIterative(5));
    escribe("Factorial recursivo", fact(6));
    escribe("Fibonacci iterativo", fiboIterative(7));
    escribe("Fibonacci recursivo", fibo(8));
}
```

```
tests/funciones.txt
"Factorial iterativo"
120
"Factorial recursivo"
720
"Fibonacci iterativo"
13
"Fibonacci recursivo"
21
```

- Llamada de funciones locales y de objetos.

```
programa pruebaPrograma ;

Clase Calculo {
    atributos:
        f: flotante;
        e: entero;

    metodos:
        entero funcion uno (a: entero, b:entero)
        variables:
            c: entero;
        {
            e = a * b;
            regresa(e);
        }

        entero funcion dos (y: entero) {
            regresa(y*3);
        }
}

variables:
    f : flotante;
    a, b : entero;
    cadena: char;
    cal: Calculo;

entero funcion uno(x: entero) {
    b = 99;
    regresa(x);
}

entero funcion dos(y: entero) {
    regresa(y*2);
}

principal () {
    escribe(cal.uno(5, 5));
    escribe(uno(7 + 2));
    escribe(dos(cal.uno(5, 5)));
    escribe(cal.dos(cal.uno(3,3)));
    escribe(b);
    cadena = "HOLA";
    escribe(cadena);
}
```

```
tests/objetos.txt
25
9
50
27
99
"HOLA"
```

Documentación del código del proyecto

Ahora bien, a lo largo del proyecto, hubo cuestiones retadoras tanto en compilación como en ejecución.

Por el lado de compilación, el tema que más tomó tiempo fue incorporar las dimensiones de arreglos y matrices. Principalmente en cómo tendríamos que crear los cuádruplos de su memoria y si se tenía que realizar un cambio en la fórmula original. De tal manera, guardamos en un stack los operandos dentro de cada índice y se fue validando y calculando cada uno:

```
# Calcular el offset de la dirección base para un arreglo
if dim == 1:
    superior_limit = var_dimension["dim1"]
    superior_limit_address = self.memory.get_cte_address(ENTERO, superior_limit)
    zero_addres = self.memory.get_cte_address(ENTERO, 0)
    dim_operando = dimensions.pop()
    self.create_cruadruplo('VERIFICA', dim_operando, zero_addres, superior_limit_address)
    temporal_address2 = dim_operando
elif dim == 2:
    # calcular el offset de la direccion base de una matriz
    for i in range(dim):
        superior_limit = var_dimension[f"dim{(i+1)}"]

        superior_limit_address = self.memory.get_cte_address(ENTERO, superior_limit)
        zero_addres = self.memory.get_cte_address(ENTERO, 0)

        dim_operando = dimensions.pop()
        self.create_cruadruplo('VERIFICA', dim_operando, zero_addres, superior_limit_address)
        if i == 0:
            temporal_address = self.memory.set_memory_address(ENTERO, None)
            m1 = superior_limit_address = self.memory.get_cte_address(ENTERO, var_dimension["dim2"])
            self.create_cruadruplo('*', dim_operando, m1, temporal_address)
        else:
            temporal_address2 = self.memory.set_memory_address(ENTERO, None)
            self.create_cruadruplo('+', temporal_address , dim_operando, temporal_address2)


temporal_address3 = self.memory.set_memory_address(ENTERO, None)
temporal_address3 = f"({temporal_address3})"
base_dir = f"dir-{the_var['direccion']}"
self.create_cruadruplo("+", base_dir, temporal_address2, temporal_address3)
self.operandos.add(temporal_address3)
self.tipos.add(the_var["tipo"])
self.operadores.pop()
```

Por el lado de ejecución, consideramos que el manejo y lectura de la memoria en momento de ejecución fue el tema que más nos costó estandarizar. Esto se debe a que a leer memoria se está trabajando con memoria ordinaria, apuntadores o direcciones base. Con base en estos tres tipo de direcciones, teníamos que identificar cómo diferenciar cómo llegaban a los cuádruplos y cómo guardarlas en el stack de memoria. De tal forma, creamos el siguiente método que fue clave al momento de leer cada memoria. Su función principal es identificar el tipo de variable que era (string o int), remover la información innecesaria y regresar la dirección correcta o el mismo valor.

```python
# metodo para leer la direccion de memoria dependiente si es dir-base, apuntador o dirección normal
def get_address_format(self, address_pointer: str) -> int:
    if isinstance(address_pointer, str):
        # direccion base
        if "dir-" in address_pointer:
            address_pointer = address_pointer.replace('dir-', '')
            return int(address_pointer)

        # apuntador de memoria
        if address_pointer[0] == "(" and address_pointer[-1] == ")":
            temp = self.memory.get_value_from_address(address_pointer)
            if temp != None:
                return temp

            address_pointer = address_pointer.replace('(', '').replace(')', '')
            address_pointer = int(address_pointer)
            address = self.memory.get_value_from_address(address_pointer)
            return address

    # dirección de memoria
    return address_pointer
```

Por otro lado, al momento de hacer una llamada a función, sabíamos que teníamos que guardar un registro de dónde nos habíamos quedado antes de invocar la función. Por lo tanto, existe un método general que se encarga de correr los cuádruplos de manera linealmente (*run_cuadruplos*) y nos vimos en la necesidad de crear otro método que se encargara de llevar la lógica por a parte de un método, sus parámetros y su stack:

```python
# metodo que ejecuta la logica del la llamada de los métodos
# es específicamente para las llamadas, no para la declaración de los métodos
def run_method(self, current_cuadruplo: tuple, pointer: int) -> int:

    current_pointer = pointer
    param_values = []

    # Primero revisamos los params, ejecutamos las operacion
    # hasta el momento en que encontramos el gosub
    while self.cuadruplos[current_pointer][0] != GOSUB:
        if self.cuadruplos[current_pointer][0] == PARAM:
            address = self.get_address_format(self.cuadruplos[current_pointer][1])
            value = self.memory.get_value_from_address(address)
            param_values.append(value)
            current_pointer += 1
        else:
            current_pointer = self.run_cuadruplos(current_pointer)

    function_name = current_cuadruplo[3]
    class_name = MAIN

    if "-" in function_name:
        class_name, function_name = function_name.split("-")

    # Una vez que ejecutamos todos los params, ahora sí, movemos el pointer
    # al inicio de la función
    pointer = self.directory.get_inicio(class_name, function_name)

    # creamos nuevo stack de memoria
    self.memory.push_memory_stack(class_name, function_name, param_values)

    # corremos los cuadruplos del método
    self.run_cuadruplos(pointer)

    # Logica del gosub
    # Remueve del stack la memoria local
    self.memory.pop_memory_stack()

    current_pointer += 1
    return current_pointer
```

Manual de usuario

Link del manual de usuario:
https://github.com/Chito-XD/proyectoCompiladores/blob/main/manual.md

Link del video demo:

https://youtu.be/VtTWrzjSPu0