

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

Campus Monterrey



**Proyecto Final – Diseño de Compiladores TC3048**

**AGRO**

**Semestre febrero-junio 2021**

**Docentes:**

**Elda Guadalupe Quiroga González**

**Héctor Gibrán Ceballos Cancino**

Aglahir Jiménez Flórez A01364026

Roberto Ramírez Monroy                  A01366943

Índice

[**Descripción del proyecto** 4](#_Toc73391139)

[Propósito y alcance del proyecto 4](#_Toc73391140)

[Análisis de requerimientos y descripción de los principales *Test Cases* 4](#_Toc73391141)

[Descripción del PROCESO general seguido para el desarrollo del proyecto 4](#_Toc73391142)

[**Descripción del lenguaje** 4](#_Toc73391143)

[Nombre del lenguaje 4](#_Toc73391144)

[Descripción genérica de las principales características del lenguaje 4](#_Toc73391145)

[Listado de los errores que pueden ocurrir 4](#_Toc73391146)

[En compilación 4](#_Toc73391147)

[En ejecución 5](#_Toc73391148)

[**Descripción del compilador** 5](#_Toc73391149)

[Equipo de cómputo, lenguaje y utilerías especiales usadas 5](#_Toc73391150)

[Descripción del análisis léxico 6](#_Toc73391151)

[Patrones de construcción (expresados con expresiones regulares) de los elementos principales 6](#_Toc73391152)

[Enumeración de los “tokens” del lenguaje y su código asociado 6](#_Toc73391153)

[Descripción del análisis de sintaxis 6](#_Toc73391154)

[Gramática formal empleada para representar las estructuras sintácticas (sin “codificar”) 6](#_Toc73391155)

[Descripción de generación de código intermedio y análisis semántico 8](#_Toc73391156)

[Código de operación y direcciones virtuales asociadas a los elementos del código 8](#_Toc73391157)

[Diagramas de sintaxis con las acciones correspondientes marcadas sobre ellos 8](#_Toc73391158)

[Tabla de consideraciones semánticas (combinaciones factibles y errores de tipo) 17](#_Toc73391159)

[Descripción detallada del proceso de Administración de memoria en la compilación 18](#_Toc73391160)

[Especificación gráfica de CADA estructura de datos usada 18](#_Toc73391161)

[**Descripción de la máquina virtual** 20](#_Toc73391162)

[Equipo de cómputo, lenguaje y utilerías especiales usadas ( en caso de ser diferente que el compilador) 20](#_Toc73391163)

[Equipo de cómputo 20](#_Toc73391164)

[Lenguaje 20](#_Toc73391165)

[Utilerías de Python 3 usadas 20](#_Toc73391166)

[Descripción detallada del proceso de Administración de memoria en ejecución (arquitectura) 20](#_Toc73391167)

[Especificación gráfica de CADA estructura de datos usada para el manejo de scopes (Memoria local, global, etc.…) 21](#_Toc73391168)

[Asociación hecha entre las direcciones virtuales (compilación) y reales (ejecución) 21](#_Toc73391169)

[**Pruebas del funcionamiento del lenguaje** 21](#_Toc73391170)

[Incluir pruebas que “comprueben” el funcionamiento del proyecto 21](#_Toc73391171)

[Codificación de la prueba (en AGRO) 21](#_Toc73391172)

[Resultados arrojados por la generación de código intermedio y por la ejecución. 21](#_Toc73391173)

[**Documentación del código del proyecto** 21](#_Toc73391174)

# **Descripción del proyecto**

## Propósito y alcance del proyecto

Este proyecto es desarrollado con el propósito de acreditar la materia TC3048 (Diseño de compiladores) y conocer más a fondo el comportamiento verdadero de los lenguajes de programación que usamos en el día a día de nuestras vidas como desarrolladores de software.

El producto es un lenguaje orientado a objetos, con las características básicas de un lenguaje multiparadigma con conceptos también procedurales, al estilo de C++.

## Análisis de requerimientos y descripción de los principales *Test Cases*

## Descripción del PROCESO general seguido para el desarrollo del proyecto

# **Descripción del lenguaje**

## Nombre del lenguaje

El lenguaje que desarrollamos tiene el nombre de AGRO, por las dos iniciales de los nombres de los integrantes del equipo, Aglahir y Roberto.

## Descripción genérica de las principales características del lenguaje

## Listado de los errores que pueden ocurrir

Nuestro compilador, al constar de dos partes, tiene diferentes tipos de errores en cada una de las etapas de la corrida de un programa.

### En compilación

En compilación tenemos:

* Errores de sintaxis

Estos errores consisten en la incorrecta formación de los tokens contenidos en el lenguaje. (Los cuales están listados en los siguientes apartados de este documento). Ejemplo:

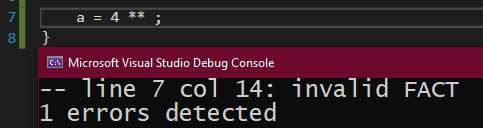
Una variable con nombre con caracteres especiales.

* Errores de gramática

Los errores de gramática consisten en el no cumplimiento de la estructura definida para los elementos.

* Incumplimiento regla

Ejemplo:

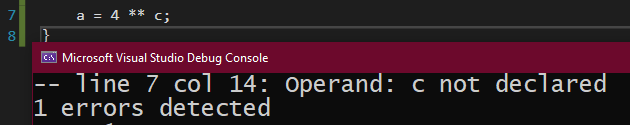


* Errores de semántica

Estos errores se presentan cuando no tiene lógica en compilación algo que se ha tecleado, aunque cumpla con la gramática y sintaxis definida.

* Variables no declaradas
* Variables haciendo operación no válida por tipos
* Falta de valor de retorno de funciones tipadas.
* Asignación de valor de funciones void

Ejemplo:



### En ejecución

# **Descripción del compilador**

## Equipo de cómputo, lenguaje y utilerías especiales usadas

Para el proyecto, utilizamos el generador de compiladores COCO/R para C#

## Descripción del análisis léxico

### Patrones de construcción (expresados con expresiones regulares) de los elementos principales

letter = 'A'..'Z' + 'a'..'z'.

digit = '0'..'9'.

tab = '\t'.

cr = '\r'.

lf = '\n'.

newLine = cr + lf.

notQuote = ANY - '"' - "\r\n".

id = letter { letter | digit }.

cte\_I = digit { digit }.

cte\_F = digit { digit } "." digit { digit }.

ctr\_Str = '"' {notQuote} '"'.

### Enumeración de los “tokens” del lenguaje y su código asociado

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Cbl** | { | **Add** | + | **Dot** | . | **Less** | < |
| **Cbr** | } | **Sub** | - | **Sadd** | += | **Greater** | > |
| **Bl** | [ | **Mul** | \* | **Ssub** | -= | **Lesseq** | <= |
| **br** | ] | **exponent** | \*\* | **sdiv** | /= | **greatereq** | >= |
| **Pl** | ( | **Div** | / | **Smul** | \*= | **Equaleq** | == |
| **Pr** | ) | **Intdiv** | // | **Increment** | ++ | **Different** | ¡= |
| **Comma** | , | **Module** | % | **Decrement** | -- | **And** | && |
| **Semicolon** | ; | **Equal** | = | **Colon** | : | **or** | || |

## Descripción del análisis de sintaxis

### Gramática formal empleada para representar las estructuras sintácticas (sin “codificar”)

**PROGRAM =** { DECLARATION } MAIN

**DECLARATION =** DEC\_VARS | DEC\_FUNC | DEC\_CLASS

**DEC\_VARS =** COMPOUND\_TYPE **id** {**, id**} **; |** SIMPLE\_TYPE **id** [ **[cte\_i]** [**[cte\_i]**]] **;**

**DEC\_FUNC =** TYPE\_FUNC **id (** [PARAMS\_FUNC] **) {** {DEC\_VARS | STATUTE} [RETURN]**}**

**DEC\_CLASS = class id** [: **id**] **{** CLASS\_DEF **}**

**COMPOUND\_TYPE = id**

**SIMPLE\_TYPE = int | float | char**

**MAIN = main {** {STATUTE | DEC\_VARS} **}**

**STATUTE =** FUNC\_CALL | METHOD\_CALL | INPUT | PRINT | CONDITIONAL | WHILE | FOR | ASSIGN

**TYPE\_FUNC = int | float | char | void**

**PARAMS\_FUNC =** SIMPLE\_TYPE **id** { **,** SIMPLE\_TYPE **id** }

**RETURN = return** HYPER\_EXP **;**

**EXP =** TERM { (**add | sub**) TERM}

**CLASS\_DEF =** ( **+ | -** ) (DEC\_VARS | DEC\_FUNC)

**ASSIGN =** VARIABLE\_ASSIGN **=** EXP **;** |

VARIABLE\_ASSIGN SHORT\_ASSIGN EXP**;** |

VARIABLE\_ASSIGN STEP**;**

**INPUT = input (**VARIABLE\_ASSIGN**);**

**PRINT = print (** {HYPER\_EXP} **) ;**

**CONDITIONAL = if (** HYPER\_EXP **)** BLOCK | **if (** HYPER\_EXP **)** BLOCK **else** BLOCK

**WHILE = while (** HYPER\_EXP **)** BLOCK

**FOR = for (** ASSIGN **;** HYPER\_EXP **;** ASSIGN**)** BLOCK

**FUNC\_CALL = id (); | id(** HYPER\_EXP {, HYPER\_EXP} **) ;**

**TERM =** FACT | FACT ( **\* | /** ) FACT

**SHORT\_ASSIGN = += | -= | \*= | /=**

**VARIABLE\_ASSIGN = id | id.id | id[**EXP**] | id[**EXP**] [**EXP**]**

**VARIABLE\_FACT = id | id() | id(**EXP {, EXP}**) | id.id | id.id(**EXP {,EXP}**) | id[**EXP**] | id[**EXP**] [**EXP**]**

**HYPER\_EXP =** SUPER\_EXP **|** SUPER\_EXP (REL\_EXP) SUPER\_EXP

**BLOCK = {** {STATUTE} **}**

**FACT = (** HYPER\_EXP **) |** (+ | -) (**cte\_i** | **cte\_f**) **|** VARIABLE\_FACT **| cte\_str**

**STEP = ++ | --**

**REL\_EXP = and | or | && | ||**

**SUPER\_EXP =** EXP **|** EXP (REL\_OP) EXP

**REL\_OP = < | > | <= | >= | == | !=**

## Descripción de generación de código intermedio y análisis semántico

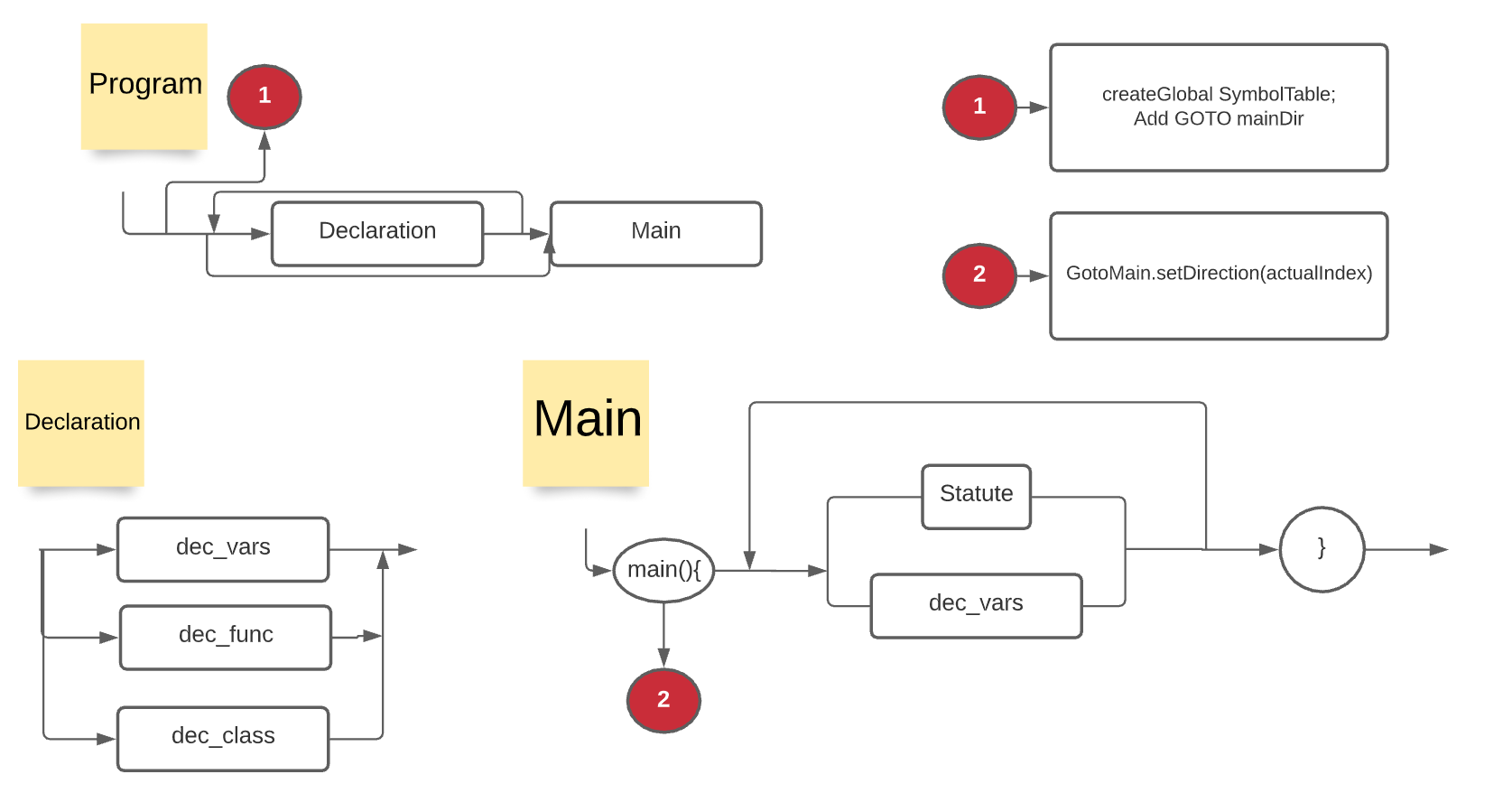
### Código de operación y direcciones virtuales asociadas a los elementos del código

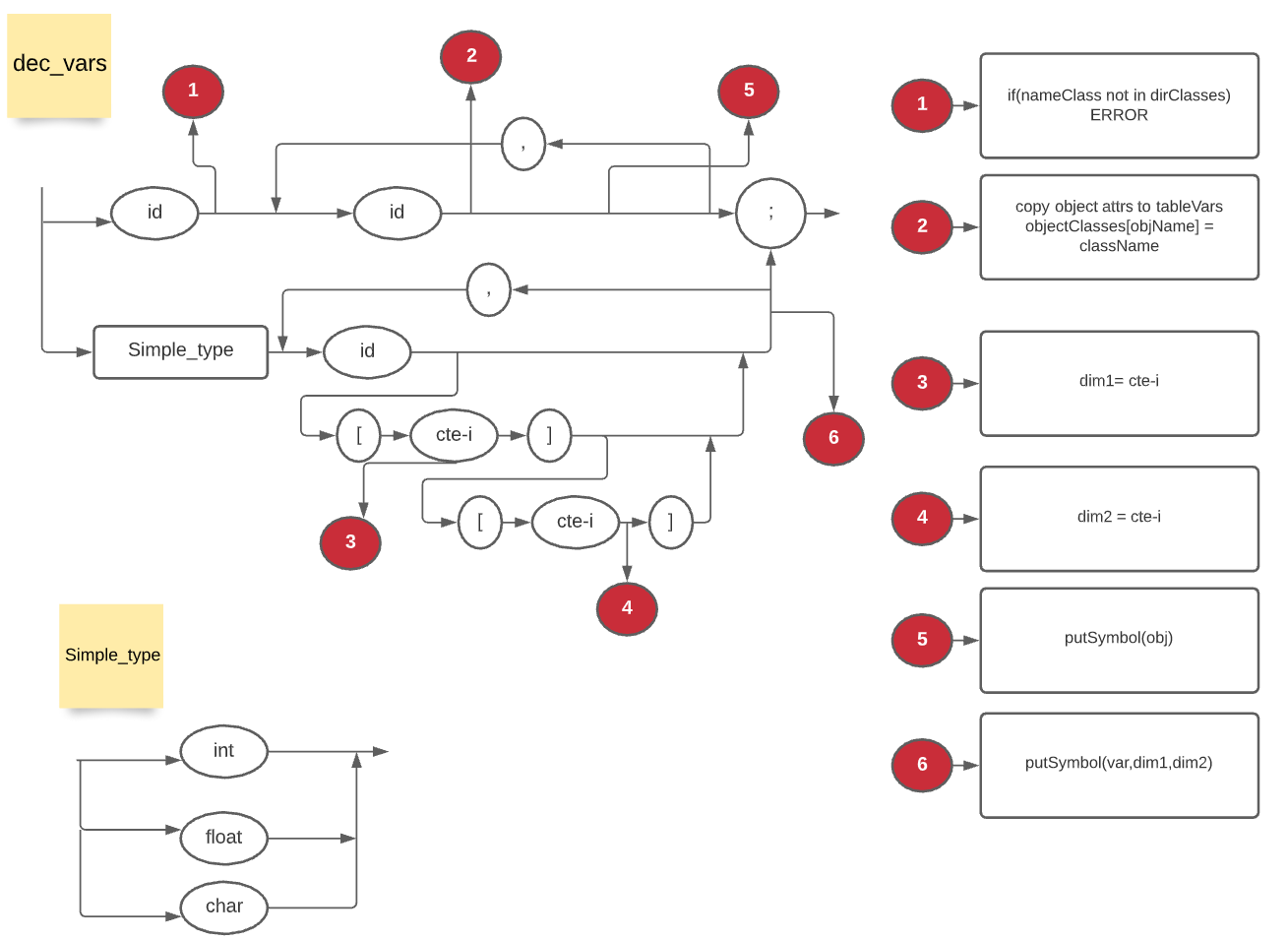
**Direcciones virtuales utilizadas:**

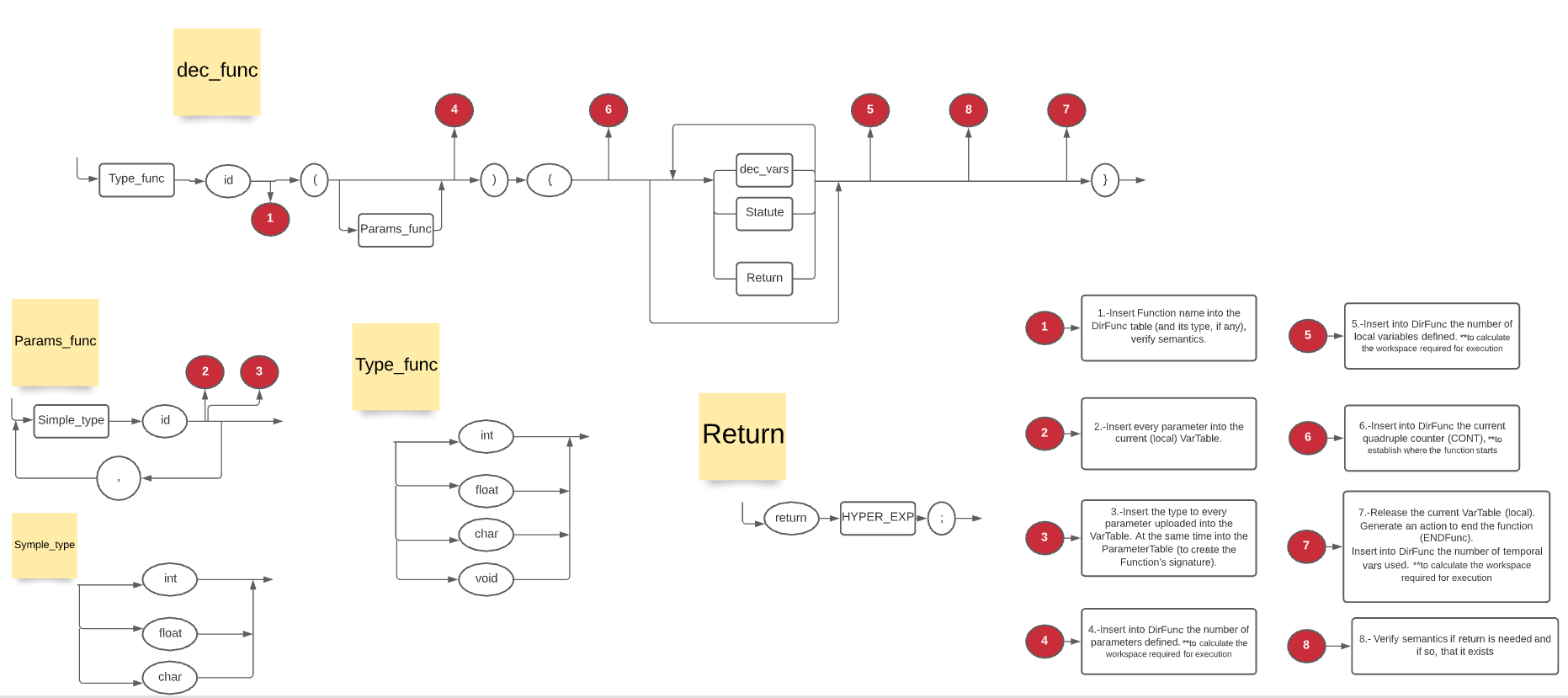
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **globalInt** | 1001 | **localInt** | 28001 | **constInt** | 42001 |
| **globalFloat** | 5001 | **localFloat** | 30001 | **constFloat** | 44001 |
| **globalChar** | 9001 | **localChar** | 32001 | **constChar** | 46001 |
| **globalTempInt** | 12001 | **localTempInt** | 34001 | **constString** | 48001 |
| **globalTempFloat** | 16001 | **localTempFloat** | 36001 | **Pointers** | 50001 |
| **globalTempChar** | 20001 | **localTempChar** | 38001 |  |  |

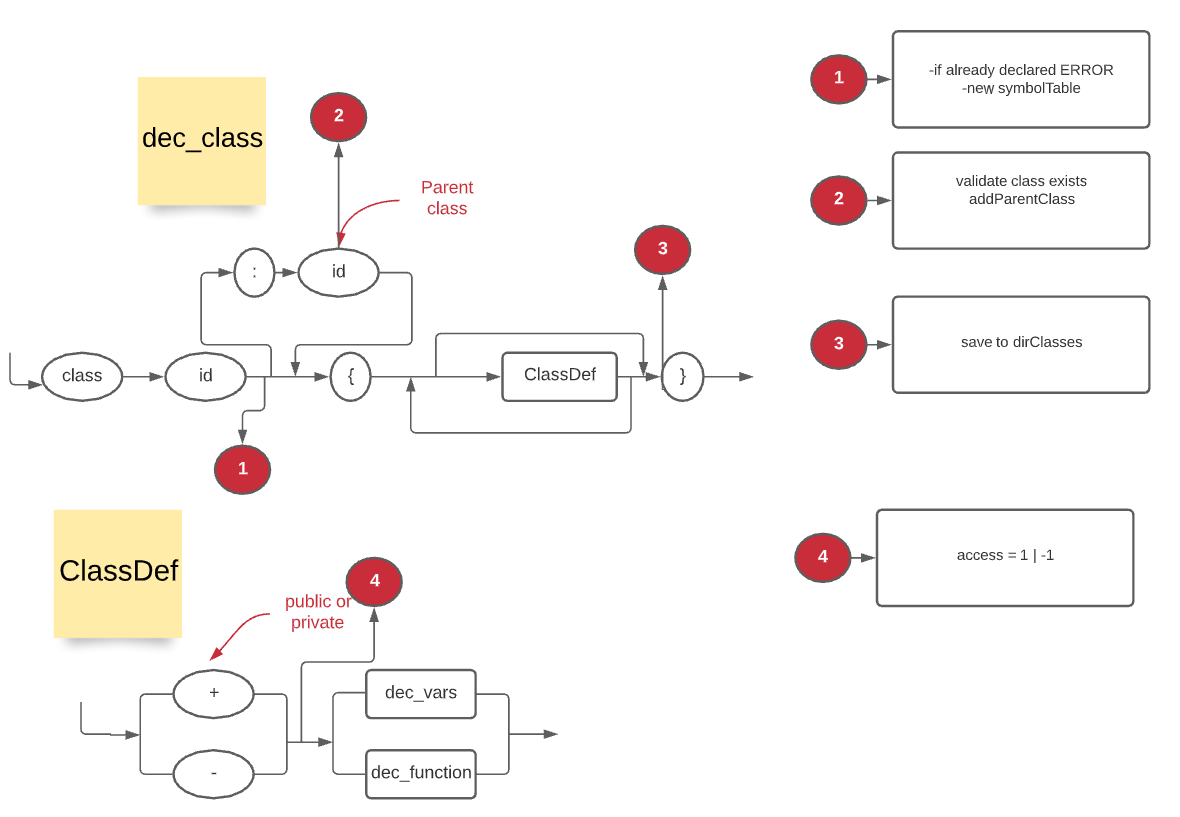
### Diagramas de sintaxis con las acciones correspondientes marcadas sobre ellos

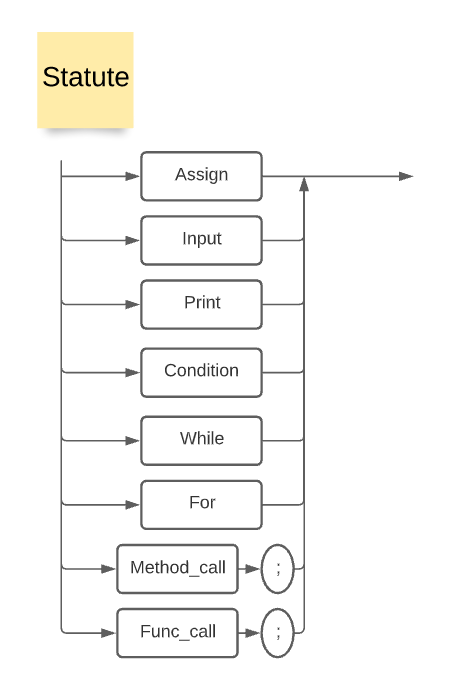
Link para lucidChart: [AGRO: Lucidchart](https://lucid.app/lucidchart/877200bf-d8c3-4432-bfdd-bdef3a8dfc60/view?page=XU9rVEFlgjx5) (diagramas en páginas)

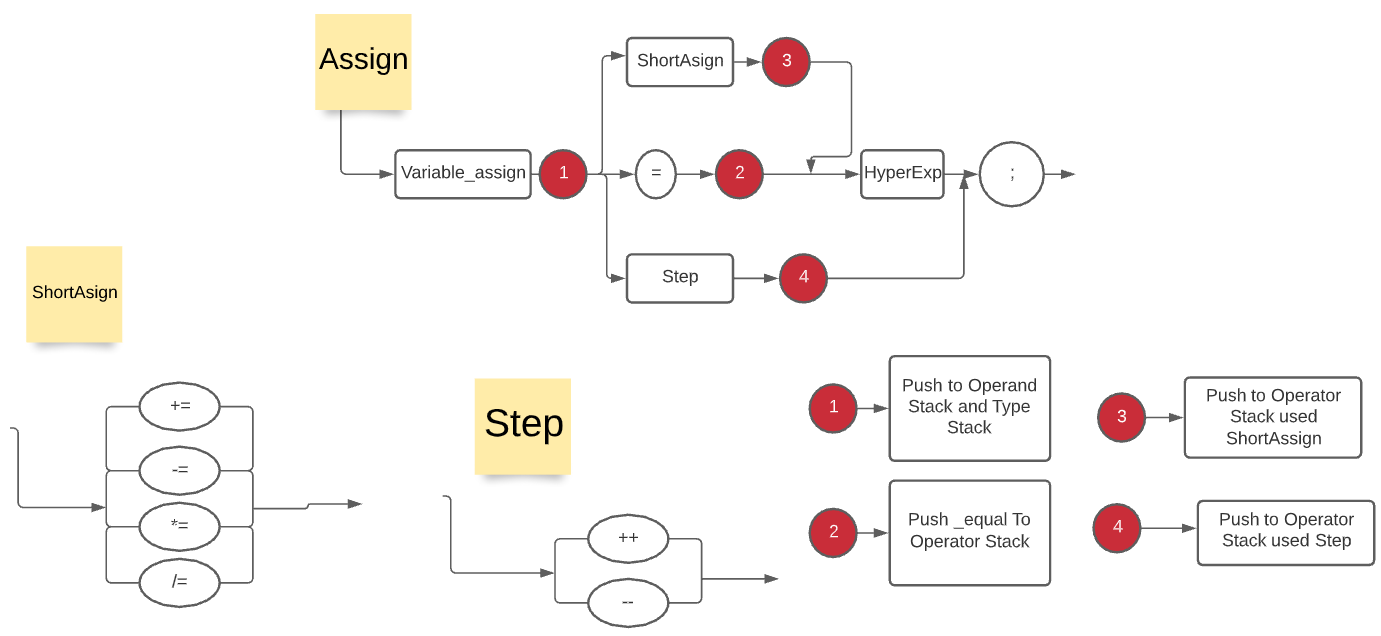


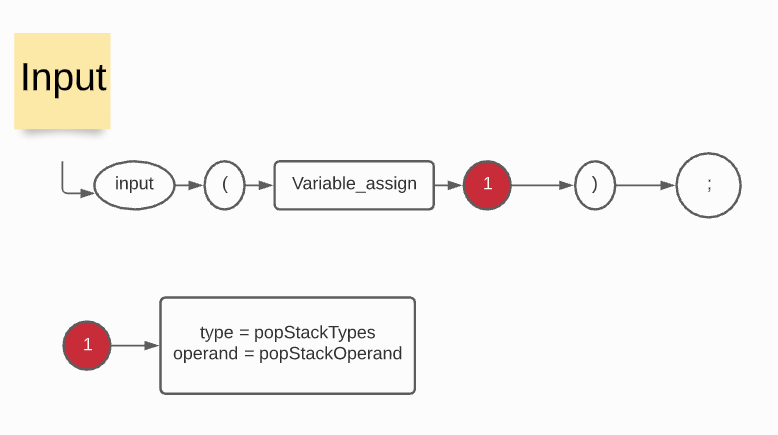


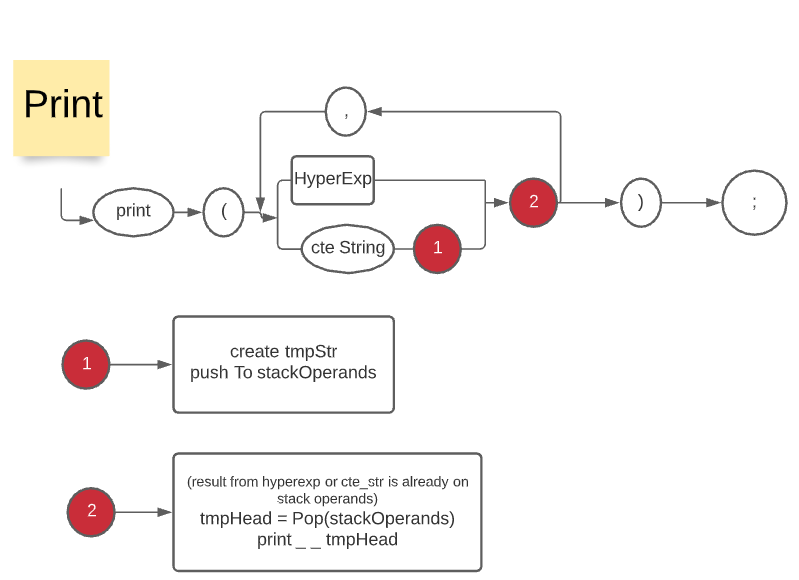


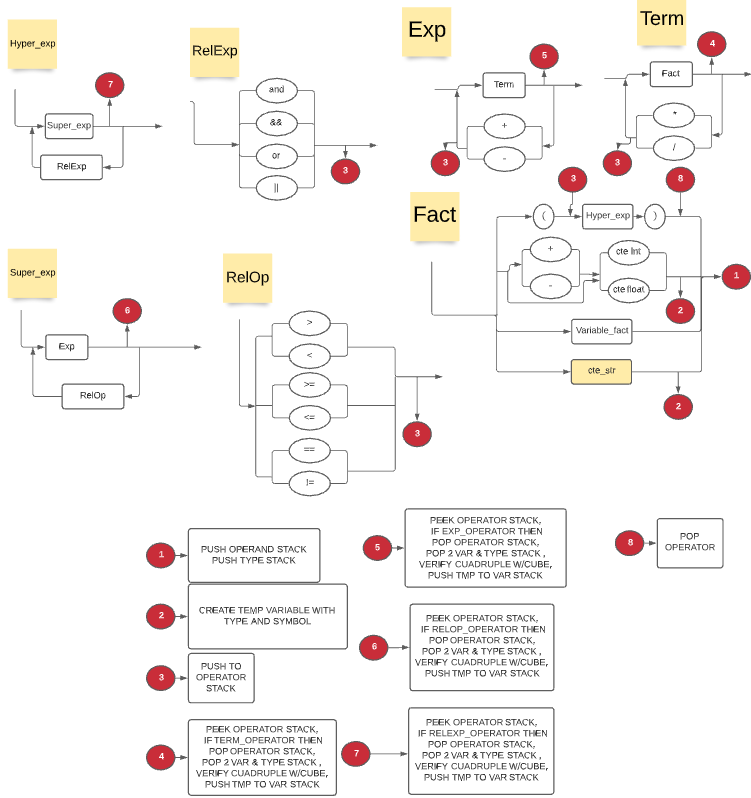


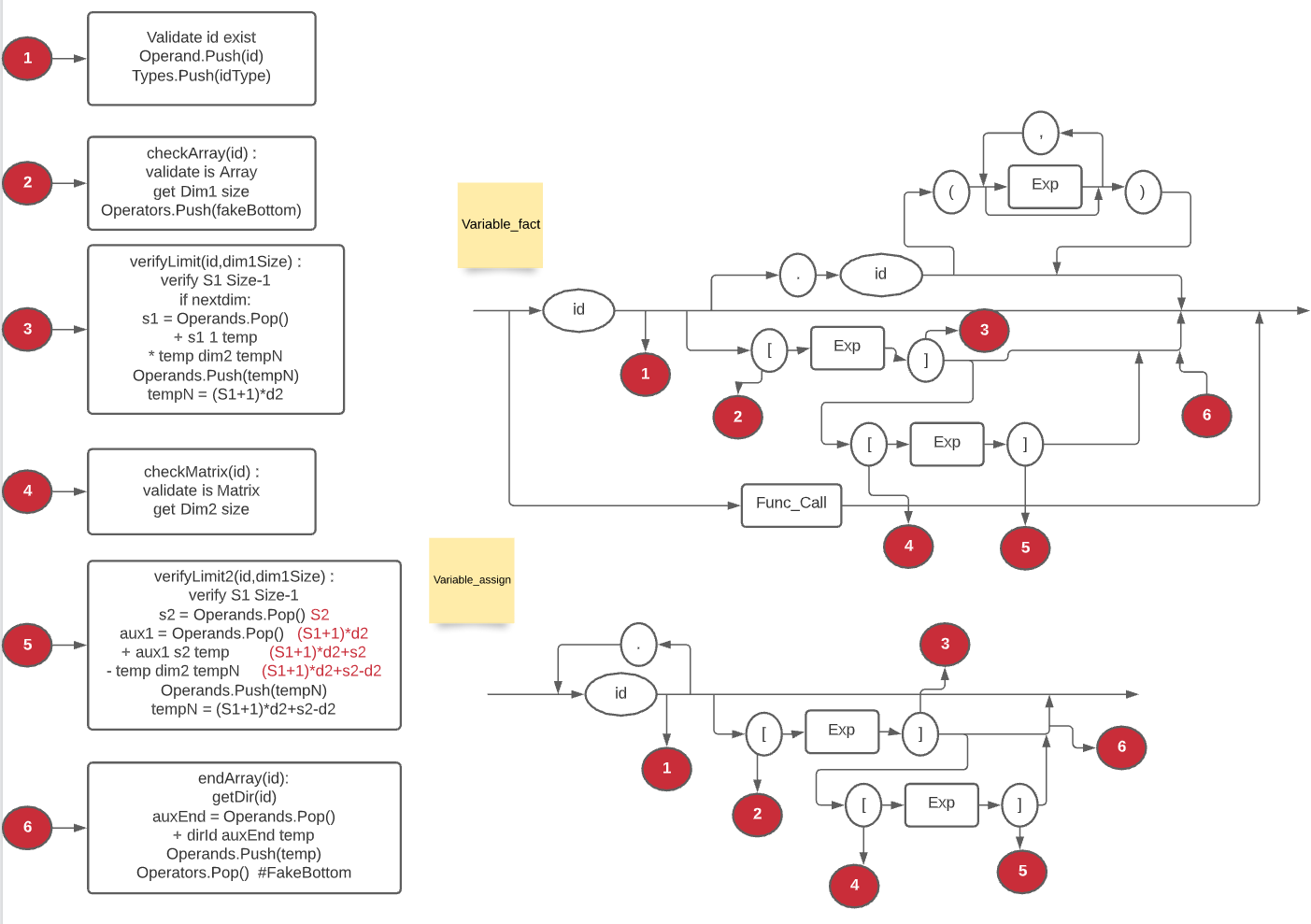


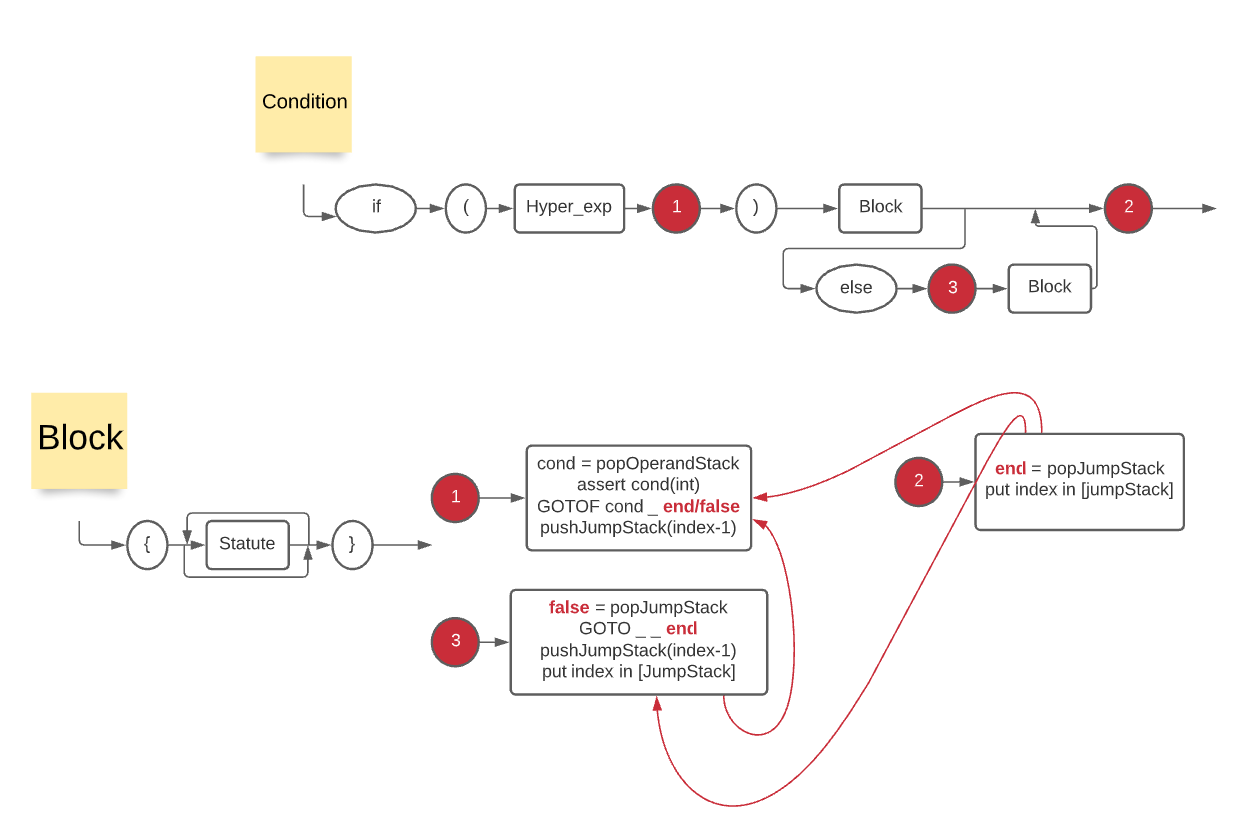


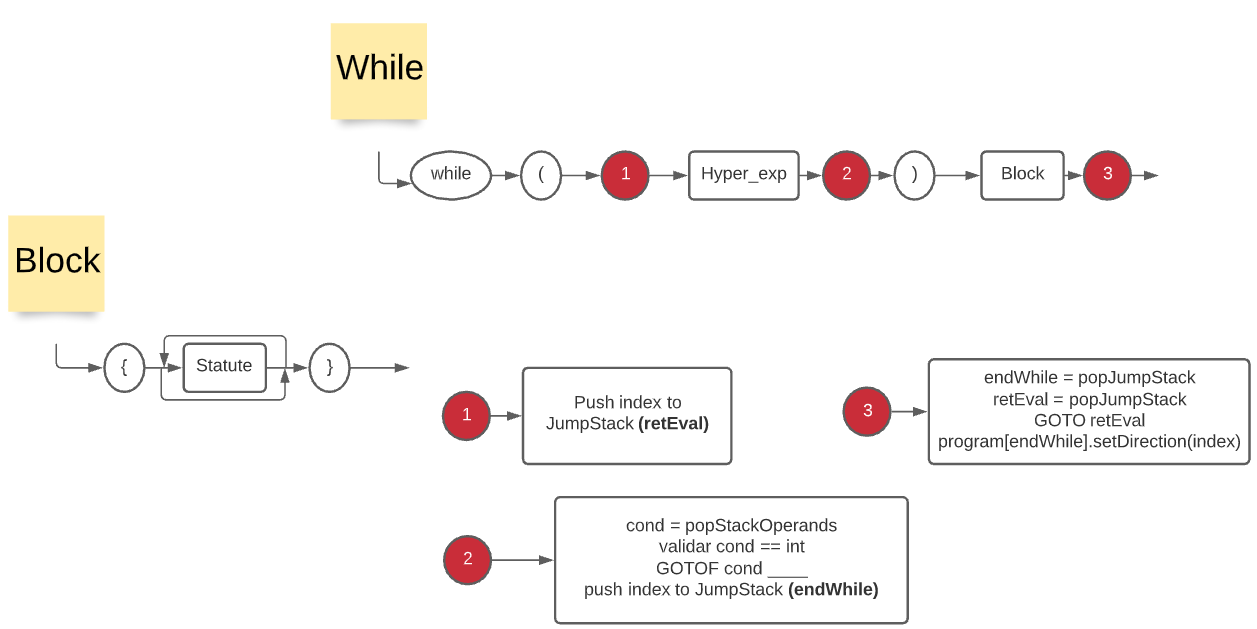


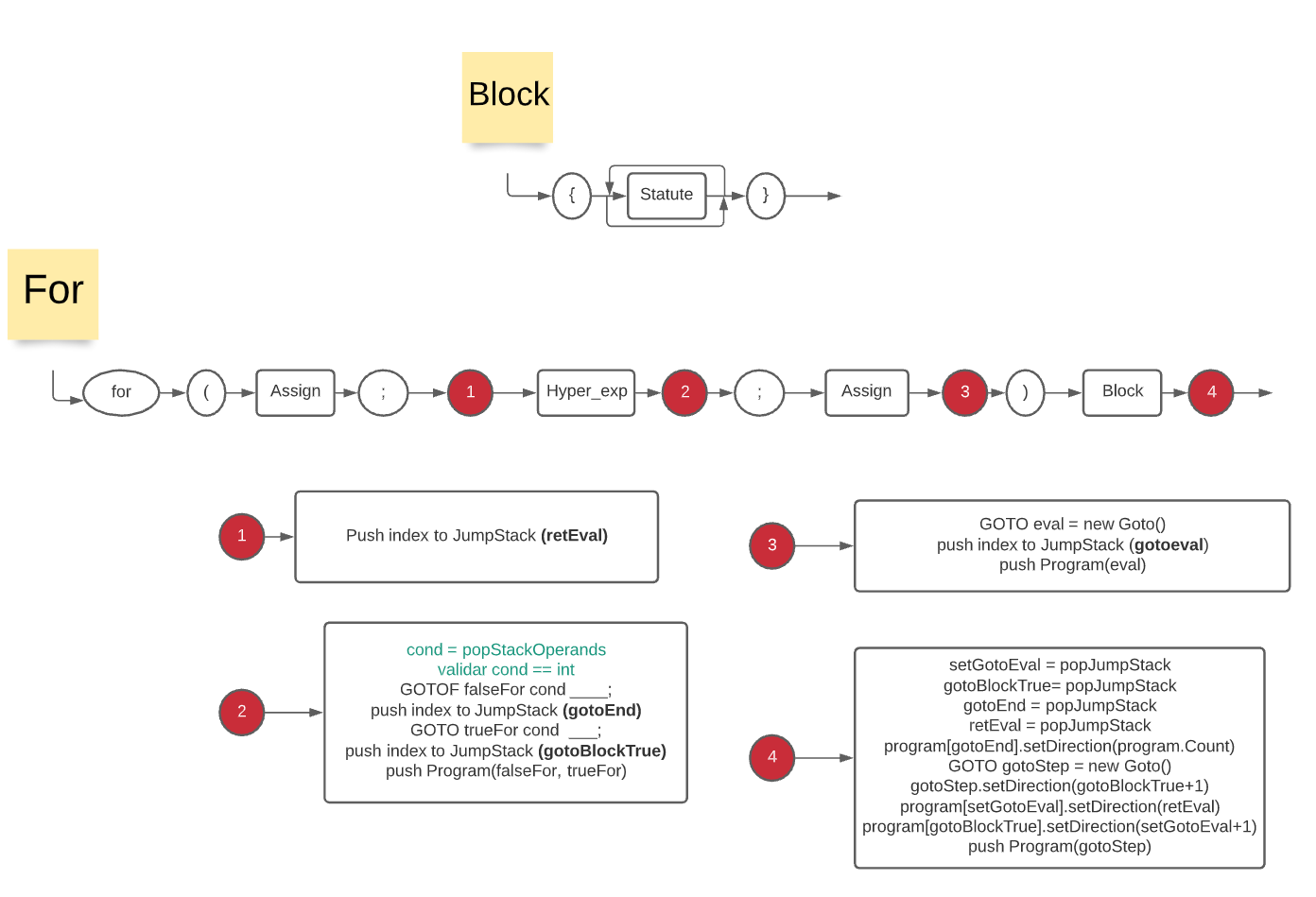


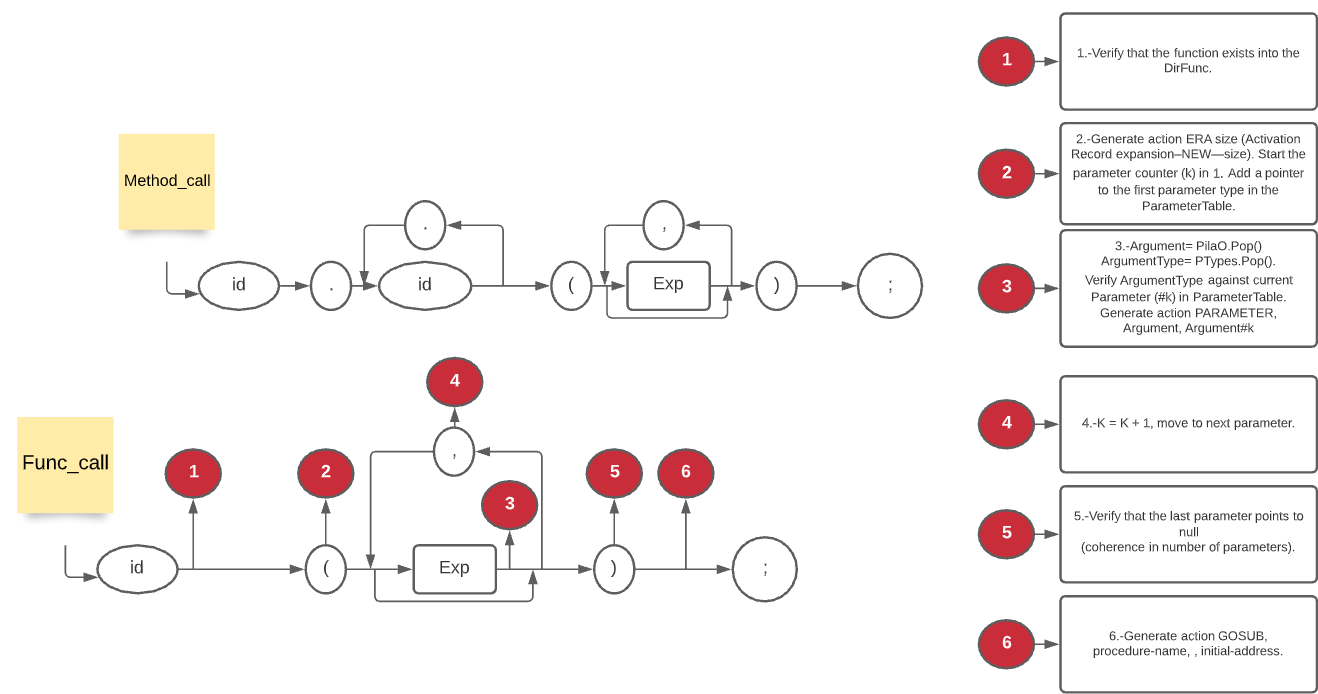












### Tabla de consideraciones semánticas (combinaciones factibles y errores de tipo)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **+, -, +=, -=** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | int | int |
| **Float** | float | float | Error |
| **Char** | char | Error | char |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **\*, /, \*=, /=** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | float | Error |
| **Float** | float | float | Error |
| **Char** | Error | Error | char |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **>, <, >=, <=, ==, ¡=** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | int | Error |
| **Float** | int | int | Error |
| **Char** | Error | Error | int |
| **and, or** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | Error | Error |
| **Float** | Error | Error | Error |
| **Char** | Error | Error | Error |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **\*\*** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | float | Error |
| **Float** | float | float | Error |
| **Char** | Error | Error | Error |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **//, %** | **RightOper** | | | |
| **LeftOper** |  | **Int** | **Float** | **Char** |
| **Int** | int | int | Error |
| **Float** | int | int | Error |
| **Char** | Error | Error | Error |

## Descripción detallada del proceso de Administración de memoria en la compilación

En nuestro compilador, usamos una administración de memoria con una arquitectura jerárquica, en la que iniciamos con una tabla de variables global y cada que entramos en un nuevo contexto (clase, función, main) creamos una nueva tabla de variables, que tiene un apuntador a la tabla anterior a la que se crea.

Las direcciones de variables, al ser parte de las tablas de variables, se van “limpiando” en cada contexto, siendo la única excepción las direcciones de constantes, las que nos aseguramos que sean insertadas en la tabla de variables global sin importar de en qué contexto se declaran.

Las direcciones de cada tabla se van modificando siempre que se use un nuevo símbolo de cualquier tipo en el contexto de esa tabla, de acuerdo al tamaño del símbolo que se agregue (en caso de arreglos o matrices).

Si se quiere usar un símbolo de la tabla de variables, se hace una búsqueda en la tabla del contexto actual, y si no se encuentra el símbolo, se hace una búsqueda en la tabla del contexto “padre”, siendo así posible usar variables globales en cualquier contexto.

Cuando se crea un objeto en un contexto, se consulta el directorio de clases, para crear todos los atributos de ese objeto en el contexto actual, pero considerando el acceso de los atributos.

En el caso de directorio de funciones y clases, solamente se tienen declarados de manera global, siendo así que, para la declaración de métodos de clases, estos se guardan en el mismo directorio de funciones global, solamente que el nombre se estructura como ***nombreClase+”.\_” +nombreMétodo***

### Especificación gráfica de CADA estructura de datos usada

|  |  |  |
| --- | --- | --- |
| Estructura | Estructura de datos |  |
| Tabla de variables | Int[] variableDirections  SymbolTable parentSymbolTable  Dictionary<string, int[]> symbols  Dictionary<string, Dictionary<string, int>> objects  Dictionary<string, string> objectClasses  Int id |  |
| Directorio funciones | Dictionary<string, Function> |  |
| Clase Function | Clase creada, con:  -int variableCount  -int tmpCount  -int quadIndex  -List<int> parameterTypes |
| Directorio de clases | Dictionary <string, Classes> |  |
| Clase Classes | Clase creada, con:  -int quadIndex  -int variableCount  -int methodCount  -Dictionary <string, int[]> |
| Cuádruplos | Clases:  Assign, Cuadruple, GoSub, Goto, InOut, Param, Return, Verify |  |
| Pilas (Operandos, Operadores, Tipos, Saltos) | *Stack<String>* **stackOperand**  *Stack<int>* **stackOperator**  *Stack<int>* **stackTypes**  *Stack<int>* **stackJumps** |  |
| Cubo Semántico | Dictionary<int, Dictionary<int, Dictionary<string, int>>> |  |
| Auxiliar trad operador | Dictionary<int, string> |  |
| Auxiliar trad tipos | Dictionary<int, string> |  |

# **Descripción de la máquina virtual**

## Equipo de cómputo, lenguaje y utilerías especiales usadas ( en caso de ser diferente que el compilador)

### Equipo de cómputo

Windows, Linux

### Lenguaje

Python 3

### Utilerías de Python 3 usadas

* re: Expresiones regulares
* operator: funciones con operadores disponibles en Python 3
* sys: biblioteca para leer parámetros al iniciar el programa en consola

## Descripción detallada del proceso de Administración de memoria en ejecución (arquitectura)

La memoria en la máquina virtual consta de múltiples contextos de memoria donde cada uno incluye sus propias estructuras para almacenar los tipos de variables más simples como int, float, char, string.

Los contextos disponibles en memoria son:

* global
* global temporales
* local
* local temporales
* constantes
* apuntadores

Cuando se llama un método se asigna el MemoryContext necesario según un directorio de funciones. Este MemoryContext se pone en la pila de localMemory y localTempMemory respectivamente. Al terminar la llamada de una función, se borran estos MemoryContext de la pila.

### Especificación gráfica de CADA estructura de datos usada para el manejo de scopes (Memoria local, global, etc.…)



### Asociación hecha entre las direcciones virtuales (compilación) y reales (ejecución)

Cada vez que se accede a una posición en memoria, se tienen definidos previamente los rangos numéricos en que una dirección pertenece a cada tipo de contexto en Memory y cada tipo de variable de MemoryContext (int, float, char, string).

En el caso de la memoria localMemory y localTempMemory, se accede al MemoryContext que esté hasta arriba de la pila de estas estructuras, ya que se refiere siempre al contexto local actual.

# **Pruebas del funcionamiento del lenguaje**

## Incluir pruebas que “comprueben” el funcionamiento del proyecto

### Codificación de la prueba (en AGRO)

### Resultados arrojados por la generación de código intermedio y por la ejecución.

# **Documentación del código del proyecto**