**Monterrey Institute of Technology and Higher Education**


# Documentación Final

## Lenguaje “Zero Two”

**Compiler Design**

**Elda Quiroga**

**Ivan Eloy Bonilla Gameros**

**05 de junio de 2023**

# Contents

# Descripción del proyecto

## Propósito

Este proyecto tiene el propósito de cumplir con los requerimientos básicos de un lenguaje de programación estructurado y simple, para así lograr orientar un poco más a los jóvenes que quieran empezar a aprender a programar. El objetivo principal es que se tenga que seguir un orden para codificar y aprender las bases de programación. Este proyecto podría ser una herramienta demasiado útil, para ser un lenguaje base y aprender todas las bases.

## Alcance

Este lenguaje de programación se diseñó para abarcar una variedad amplia de tareas de programación básicas y fundamentales, con el fin de brindar a los principiantes una sólida base en los conceptos de programación estructurada. El alcance del lenguaje incluye la posibilidad de realizar operaciones aritméticas básicas como suma, resta, multiplicación y división, así como funciones matemáticas más complejas como cálculos exponenciales y factoriales.

Además, el lenguaje proporciona herramientas para controlar el flujo del programa a través de estructuras de control como bucles y condicionales. Los usuarios pueden utilizar estas herramientas para escribir programas que puedan tomar decisiones basadas en ciertas condiciones y repetir acciones, lo cual es un concepto crucial en cualquier lenguaje de programación.

Para facilitar la organización y el manejo de datos, el lenguaje también permite la declaración y manipulación de variables y arreglos. Los usuarios pueden almacenar información en estas estructuras y luego acceder y modificar esa información de manera eficiente.

Además, este lenguaje de programación incorpora un conjunto de funciones predefinidas que facilitan la realización de tareas comunes y a menudo complicadas, ahorrando así al programador el tiempo y el esfuerzo que supondría implementar estas funciones desde cero. Estas funciones incluyen, pero no se limitan a, tareas como la generación de números aleatorios, la manipulación de cadenas de texto y la entrada y salida de datos.

Finalmente, aunque el lenguaje está diseñado principalmente para principiantes, también es lo suficientemente flexible y potente como para ser utilizado en una variedad de aplicaciones más avanzadas. El objetivo es proporcionar una base sólida en programación sobre la cual los usuarios puedan construir a medida que amplían sus conocimientos y habilidades.

El alcance de este proyecto, sin embargo, va más allá de ser simplemente un lenguaje de programación. Se busca que sea una herramienta de aprendizaje, que facilite a los principiantes el camino hacia la adquisición de habilidades de programación y el entendimiento de los conceptos de programación de computadoras. Con una sintaxis simple y clara y una gama de funcionalidades básicas pero fundamentales, este proyecto es una plataforma perfecta para aquellos que desean aprender a programar.

## Análisis de Requerimientos

Para satisfacer las necesidades del proyecto y proporcionar una herramienta de aprendizaje de programación eficaz, se definieron varios requerimientos clave.

1. **Simplicidad**: Para ser accesible a los principiantes, el lenguaje debe ser simple y directo. Esto significa que la sintaxis debe ser clara y fácil de entender, y que los conceptos básicos de la programación deben ser fáciles de aplicar.
2. **Estructuras de Control**: Las estructuras de control básicas, como los bucles y las declaraciones condicionales, son esenciales en cualquier lenguaje de programación. Deben estar disponibles y ser fáciles de utilizar para que los usuarios puedan controlar el flujo de sus programas de manera eficaz.
3. **Manipulación de Datos**: Los usuarios deben ser capaces de declarar y manipular variables y estructuras de datos. Esto implica la capacidad de almacenar y recuperar datos, así como realizar operaciones básicas en estos datos.
4. **Operaciones Matemáticas**: Como mínimo, los usuarios deben ser capaces de realizar operaciones matemáticas básicas. Esto incluye la suma, resta, multiplicación y división, así como la posibilidad de realizar cálculos más complejos.
5. **Funciones Predefinidas**: Para ahorrar tiempo y esfuerzo, el lenguaje debe incluir una serie de funciones predefinidas que guiaran al usuario de como realizar operaciones simples, como la suma de dos variables, multiplicación, llamada de funciones, etc.
6. **Extensibilidad**: Aunque el lenguaje está diseñado para principiantes, también debe ser extensible para permitir a los usuarios expandir sus habilidades. Esto significa que debe ser capaz de manejar programas más complejos a medida que los usuarios ganen confianza y experiencia.
7. **Estabilidad y Fiabilidad**: El compilador del lenguaje debe ser robusto y capaz de manejar una variedad de programas sin errores ni fallos. También debe proporcionar mensajes de error útiles y específicos para ayudar a los usuarios a depurar sus programas.

Estos requerimientos proporcionan una base sólida sobre la cual se puede construir el lenguaje de programación, garantizando que sea tanto una herramienta de aprendizaje eficaz como un lenguaje de programación funcional y versátil.

## Test Cases

Para garantizar la funcionalidad correcta y la robustez del compilador del lenguaje de programación, se deben realizar pruebas exhaustivas. Aquí se presentan algunos casos de prueba clave que se recomienda cubrir:

1. **Hello World**: Un programa simple que imprime "Hello, World!" en la salida. Este caso de prueba verifica que el compilador pueda manejar una estructura básica de programa y producir la salida esperada.
2. **Operaciones aritméticas básicas**: Prueba de las operaciones aritméticas básicas, como suma, resta, multiplicación y división, utilizando números enteros y de punto flotante. Esto asegura que el compilador pueda realizar cálculos matemáticos correctamente.
3. **Estructuras de control**: Prueba de las estructuras de control, como bucles (while) y declaraciones condicionales (if, else). Esto verifica que el compilador pueda controlar el flujo del programa y ejecutar las instrucciones según las condiciones especificadas.
4. **Manipulación de variables**: Prueba de la declaración y asignación de variables, así como de la recuperación y manipulación de sus valores. Esto garantiza que el compilador pueda manejar correctamente las variables y los datos almacenados en ellas.
5. **Funciones y llamadas a funciones**: Prueba de la definición de funciones, así como de las llamadas a funciones y la correcta transmisión de argumentos y retorno de valores. Esto asegura que el compilador pueda gestionar las funciones correctamente y permita la modularidad del código.
6. **Manipulación de estructuras de datos**: Prueba de la declaración y manipulación de estructuras de datos más complejas, como arreglos. Esto verifica que el compilador pueda manejar estructuras de datos más allá de las variables simples.
7. **Gestión de errores**: Prueba de la capacidad del compilador para detectar y reportar errores de sintaxis, errores semánticos y otros problemas durante el proceso de compilación. Esto garantiza que el compilador pueda proporcionar mensajes de error útiles y ayudar a los programadores a depurar sus programas.

Es importante cubrir una amplia gama de casos de prueba para asegurar que el compilador del lenguaje funcione correctamente en diferentes escenarios y cumpla con los estándares y requisitos comunes de los lenguajes de programación.

## Descripción del Proceso

El proceso de desarrollo de este proyecto fue uno de los mas complicados que he realizado, pero se tenían que realizar avances semanales para mantener un orden de desarrollo. Fue un reto demasiado grande personalmente, era muy complicado seguir el orden de desarrollo, pero se logró desarrollar al final.

### Avance 0: Tokens, Diagramas de Sintaxis y Gramática

Desde el primer avance estaba teniendo complicaciones con la gramática y como funcionaban y se leían los diagramas de sintaxis, por lo que la mayoría de este tiempo se dedico realizar al menos uno bien y funcional.

Avance 1: Léxico Sintaxis

En este avance ya teníamos que estar tener terminado el programa lexer, este fue uno de los puntos “simples” del proyecto, ya que tienen demasiada importancia, pero no era complicado definir los tokens que iba a utilizar ya que mi objetivo era un programa sencillo. Me gusto que pudiera dar de alta el token que yo quisiera y darle un significado.

Avance 2: Semántica Básica de Variables y Cubo Semántico

Al no contar con unos diagramas de sintaxis básicos bien estructurados, era complicado continuar y empezar la semántica de declaración y asignación de variables.

Avance 3: Semántica, Generación de Código de Expresiones y Estatutos Lineales

Para este avance, ya iba muy atrás ya que estaba demasiado enfocado en que las reglas de gramática quedaran bien estructuradas.

Avance 4: Generación de Código de Estatutos Condicionales y Ciclos

Sigo en las reglas gramaticales, tenia muchos errores, por lo que decidí empezar de nuevo desde 0, volviendo a hacer el lexer y el parser, ya con un mejor entendimiento de lo que estaba haciendo.

Avance 5: Generación de Codigo de Funciones

Finalmente tenía un parser competitivo que lograr el mínimo funcionamiento requerido.

Avance 6: Mapa de memoria de Ejecución para la Máquina Virtual y Ejecución de Expresiones

Continue con la semántica, declarando las funciones principales para el funcionamiento, seguía haciendo modificaciones en la gramática. Pero estatutos lineales y de ciclos casi terminados.

Avance 7: Codigo de Arreglos y Ejecución de Estatutos

La generación de código intermedio, se desarrollo de manera mas rápida, logrando mucho avance y llegue a funciones.

Avance 8:Codigo y Ejecución de Aplicación Particular

Llegando a funciones, fue demasiado difícil lograr los cambios de contexto, llevo el noventa por ciento del tiempo lograr un cambio de contexto para las funciones.

## Commits

Por falta de organización, no subí mi proyecto desde un inicio a un repositorio, todo el proyecto lo trabajé localmente. Después ya lo subí a un repositorio, y genere commits donde mejore el proyecto exponencialmente.

“Commits on Jun 5, 2023

Cambio para menjo de Goto main correcto, y len(quadruples) correcto.

@Bonilla7784

Bonilla7784 committed 2 days ago

Commits on Jun 7, 2023

Recursive done.

@Bonilla7784

Bonilla7784 committed 14 hours ago

Funciones recursivas y manejo de arreglos. CHECK

@Bonilla7784

Bonilla7784 committed 28 minutes ago”

## Aprendizajes logrados

Fue reto muy difícil, nunca había hecho algo parecido. Sin embargo, creo que es un proyecto indispensable en nuestro forjamiento como profesionales, ya que después de un proyecto de este calibre, es posible tener un conocimiento base pero de alto nivel, sobre la programación y todos los lenguajes en general. Nunca había utilizado Python, ni una herramienta como PLY, por lo que si fue una curva de aprendizaje demasiado larga y grande saber como manejar las estructuras, funciones y todo en general. Al final pasaba mas tiempo investigando como hacer cosas, que codificando, entonces si codifique mas de 500 líneas, pase el triple investigando como escribir esas líneas.

Es una experiencia única que recordare por siempre.

# Descripción del Lenguaje

### Características del lenguaje

Nombre del Lenguaje: “Zero Two”

El lenguaje de programación "Zero Two" fue diseñado específicamente para ser un lenguaje de programación orientado a principiantes, con el objetivo de enseñar los fundamentos de la programación de manera accesible. Se tiene un bloque llamado “Variables” en donde tenemos que definir todas las variables globales a declarar. Y después un bloque “main()” que es donde podemos escribir nuestras lineas de código, entre estos bloques se pueden definir funciones. Es demasiado importante el orden en este lenguaje de programación.

### Listado de Errores

Compilación

1. raise *Exception*("Variable " + p[1] + " not declared")
2. raise Exception("Type mismatch in assignment")
3. raise Exception("Type mismatch in expression")
4. raise Exception("Type mismatch in term")
5. raise Exception("Type mismatch in comparison expression")
6. raise Exception(f"Function {func_name} already declared.")
7. raise Exception(f"Return variable {p[2]} not found in function {current_function}.")
8. raise Exception("No current scope to declare variable in.")
9. raise Exception("Parameters cannot be declared in the global scope.")
10. raise Exception(f"Function {func_name} does not exist.")
11. raise Exception(f"Function {func_name} not declared.")
12. raise Exception(f"Function {func_name} called with incorrect number of arguments.")
13. raise Exception(f"Type mismatch in argument {i+1} of function {func_name} call.")
14. raise Exception(f"Unknown token type: {type(token)}")
15. raise Exception(f"Type mismatch in expression: {expression}")

Ejecución

1. raise Exception(f"Unknown built-in function '{function_name}'")

# Descripción del compilador

Equipo de computo

- Ordenador local con Sistema operativo Windows

Lenguaje

- Python

Librerías especiales

- PLY
- JSON

Descripción del Análisis Léxico

| Token | Expresión regular |
|---|---|
| ID | r'[a-zA-Z]([a-zA-Z]\|[0-9])*' |
| CTEI | r'\d+' |
| FLOAT_NUMBER | r'\d+\.\d+' |
| STRING_LITERAL | r'\"([^\\\n]\|(\\.))*?\"' |
| TRUE | r'true' |
| FALSE | r'false' |
| PLUS | r'\+' |
| MINUS | r'-' |
| MULTIPLY | r'*' |
| DIVIDE | r'/' |
| EQUALS | r'=' |
| LPAREN | r'\(' |
| RPAREN | r'\)' |
| LBRACKET | r'\[' |
| RBRACKET | r'\]' |
| LBRACE | r'\{' |
| RBRACE | r'\}' |
| SEMICOLON | r';' |
| COMMA | r',' |
| COLON | r':' |
| LESS_THAN | r'<' |
| GREATER_THAN | r'>' |
| LESS_EQUAL | r'<=' |
| GREATER_EQUAL | r'>=' |
| EQUAL_EQUAL | r'==' |
| NOT_EQUAL | r'!=' |

Palabras claves

| KEYWORDS |
|---|
| PROGAMA |
| VARIABLES |
| INT |
| FLOAT |
| STRING |
| BOOL |
| PRINT |
| IF |

| |
|---|
| ELSE |
| WHILE |
| RETURN |
| VOID |
| MAIN |
| TEACH_SUM |
| TEACH_SUBSTRACTION |
| TEACH_MULTIPLICATION |
| TEACH_DIVISION |
| TEACH_IF |
| TEACH_WHILE |
| TEACH_FUNCTION_DECLARATION |
| TEACH_FUNCTION_CALL |
| AYUDA |

### Descripción de Generación de Código Intermedio y Análisis Semántico

Para comenzar, he definido varios espacios de memoria para diferentes tipos de variables y para diferentes ámbitos, como 'global', 'local' y 'temporal'. Estos espacios de memoria almacenan las direcciones de memoria virtuales que se asignarán a las variables durante la ejecución del código.

Luego, he creado lo que se conoce como un "cubo semántico". Esta estructura es muy útil en el análisis semántico para definir las reglas de cómo interactúan diferentes tipos de datos entre sí con diferentes operadores. Esto asegura que las operaciones se realicen entre los tipos de datos correctos y también ayuda a definir qué tipo de dato será el resultado de dicha operación.

Además, he definido múltiples estructuras de datos, como pilas y tablas de símbolos, para ayudar a rastrear las variables y operadores a medida que el código se compila y se generan los cuádruplos.

Los cuádruplos son una forma de representar el código intermedio. Cada cuádruplo consta de un operador y dos operandos (izquierdo y derecho), y un resultado. Estos cuádruplos son una forma compacta de representar instrucciones de máquina y facilitan la optimización posterior del código.

### Direcciones virtuales asociadas a los elementos del código

En este compilador, defini varios espacios de memoria según el tipo de datos y su alcance. Los espacios de memoria global, local y temporal se refieren a la memoria reservada para variables globales, locales y temporales respectivamente. La memoria global es accesible para todo el código, mientras que la memoria local está disponible solo dentro de un contexto o alcance específico, generalmente dentro de una función o método. La memoria temporal es utilizada para almacenar valores intermedios generados durante la ejecución del código.

El espacio de memoria global está subdividido de la siguiente manera: las variables de tipo 'int' se almacenan a partir de la dirección 1000, las de tipo 'float' a partir de 2000, las de tipo 'bool' a partir de 3000, las de tipo 'string' a partir de 4000, los valores temporales a partir de 5000 y las constantes a partir de 6000.

De manera similar, para el espacio de memoria local, hemos asignado las direcciones a partir de 7000 para 'int', 8000 para 'float', 9000 para 'bool', 10000 para 'string' y 11000 para los valores temporales.

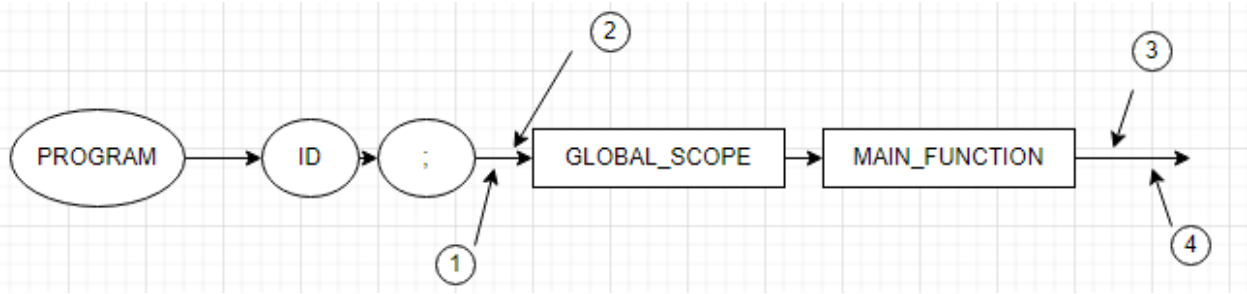
En el caso de la memoria temporal, los espacios comienzan desde 12000 para 'int', 13000 para 'float', 14000 para 'bool' y 15000 para 'string'.

Este diseño de espacios de memoria es vital para la asignación y gestión eficiente de memoria durante la ejecución del código. Los espacios reservados para cada tipo de datos garantizan que no haya conflictos de tipos de datos o pérdida de datos debido a la sobrescritura. Además, proporciona una forma eficiente de acceder y manipular las variables en función de su tipo y alcance. La elección de las direcciones de inicio para cada tipo de dato también permite una fácil expansión o contracción del espacio de memoria en caso de necesidad.

Es importante mencionar que, aunque estas direcciones parecen estar fijas, en realidad estas representan direcciones virtuales. La asignación exacta de la memoria física puede variar dependiendo de la implementación del sistema operativo y la gestión de memoria. El compilador y el sistema operativo trabajan juntos para mapear estas direcciones virtuales a las direcciones físicas durante la ejecución del código.

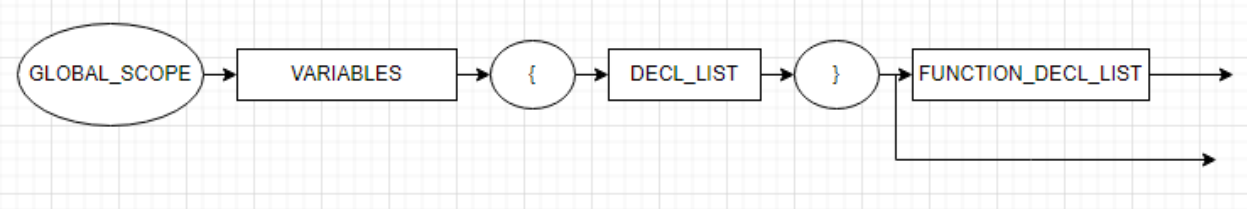
## Diagramas de Sintaxis

- P_PROGRAM

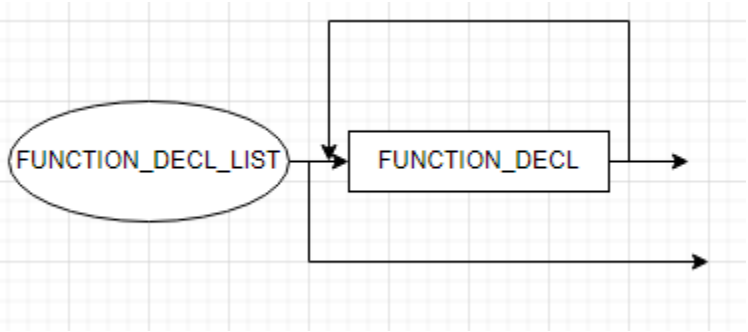



1) Placeholder_goto_main
2) Create_scopes
3) End_scopes
4) Fill_goto_main

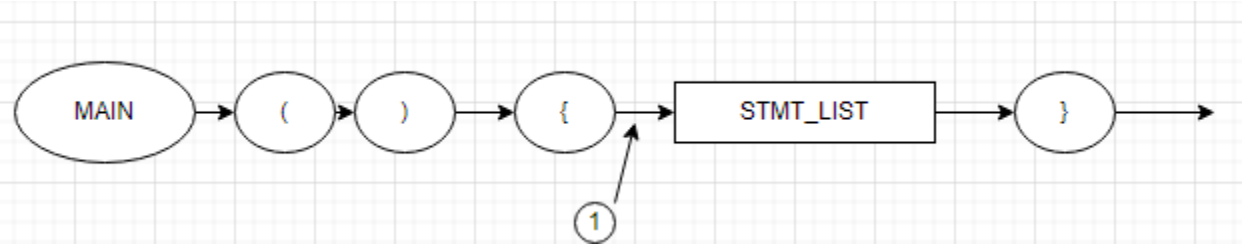
- P_GLOBAL_SCOPE




- P_FUNCTION_DECL_LIST

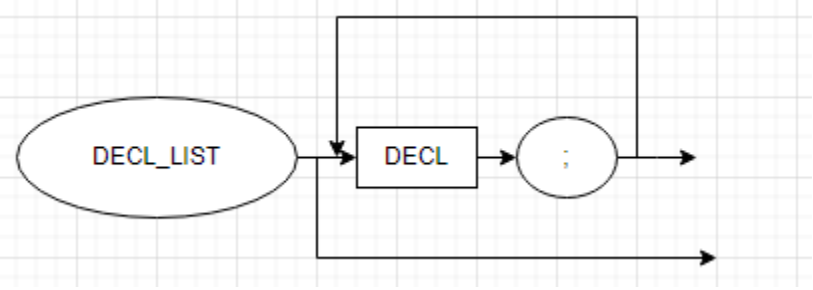



- P_MAIN_FUNCTION

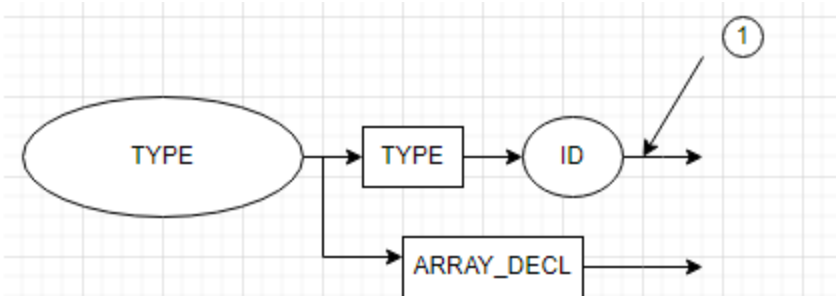



1) START_MAIN

- P_DECL_LIST

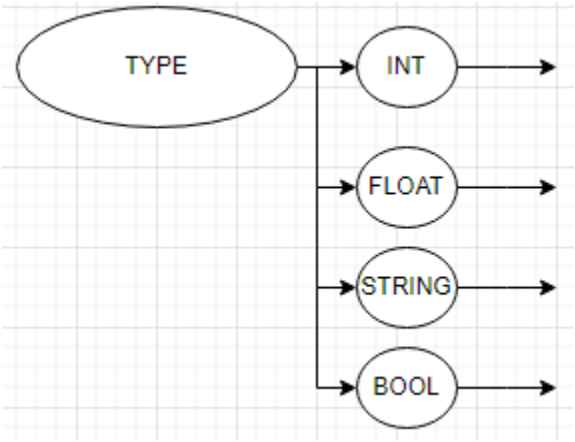



1. P_DECL

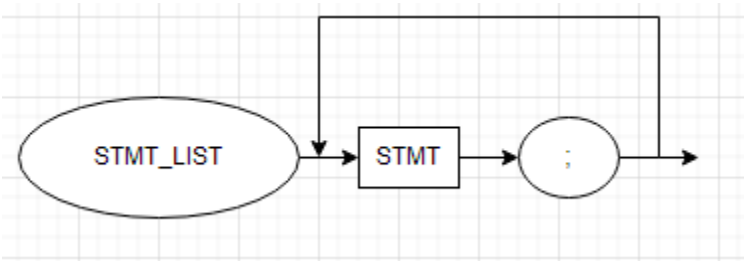



1. Add_to_current_scope

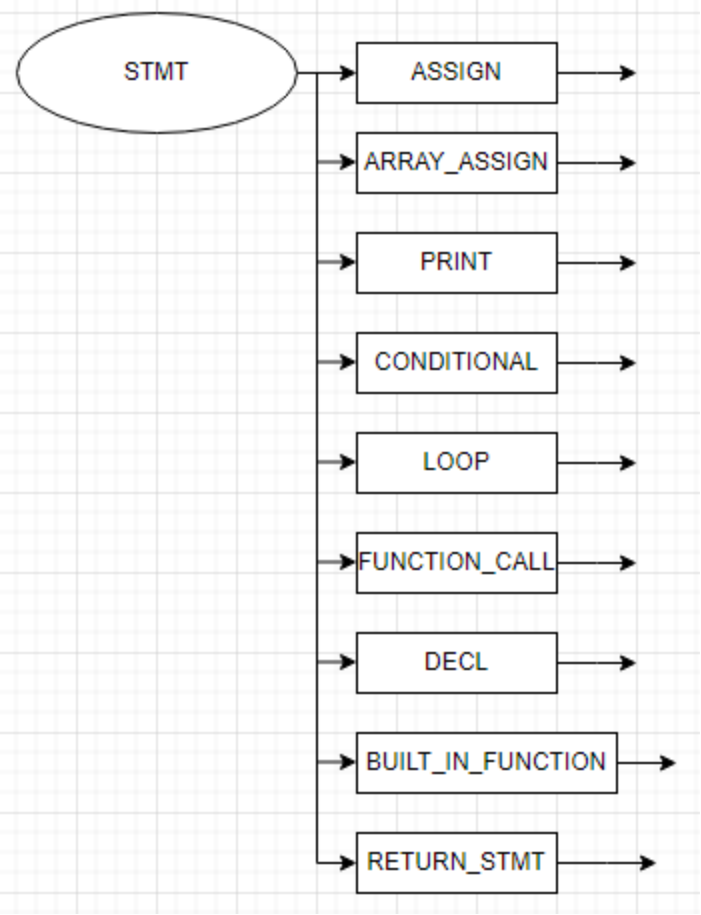
2. P_TYPE



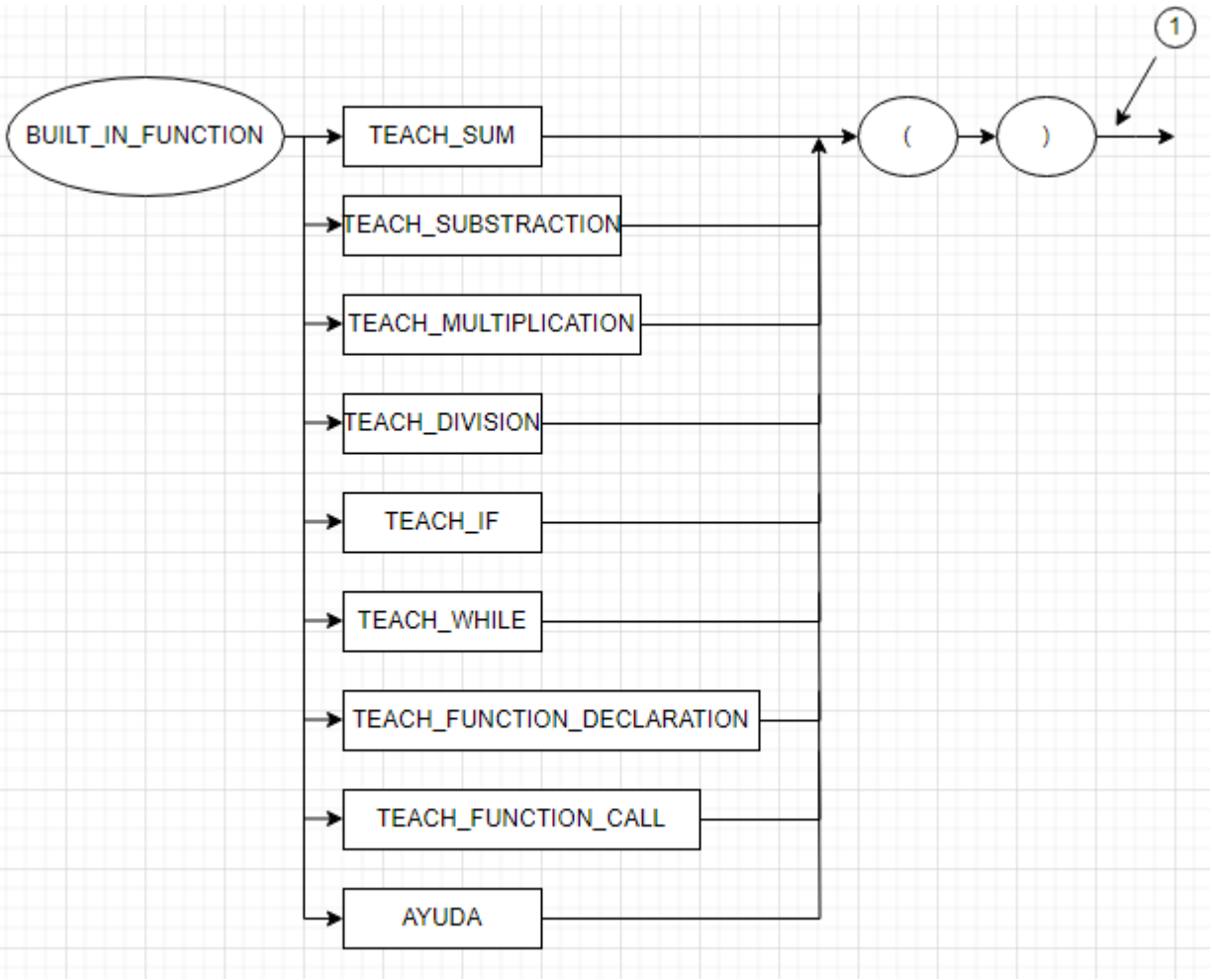

3. P_STMT_LIST




4. P_STMT

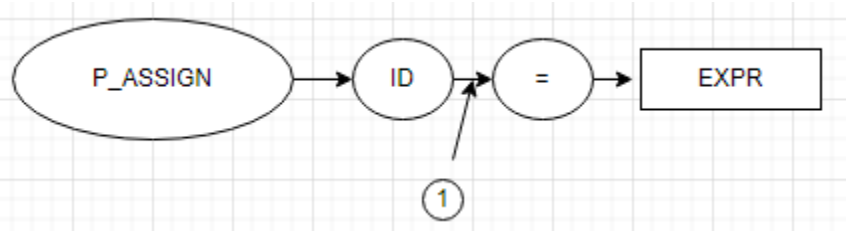



5. P_BUILT_IN_FUNCTION



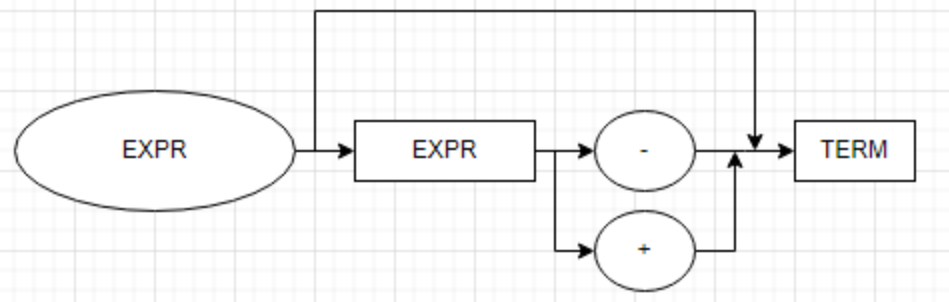

1. Generate_built_in_function_call

- P_ASSIGN

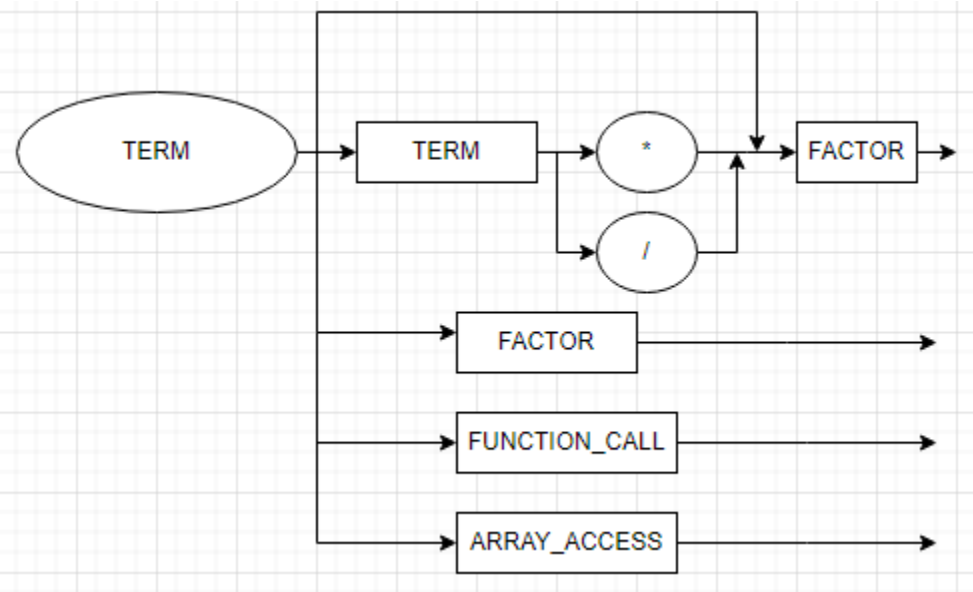



2. Get_variable_type

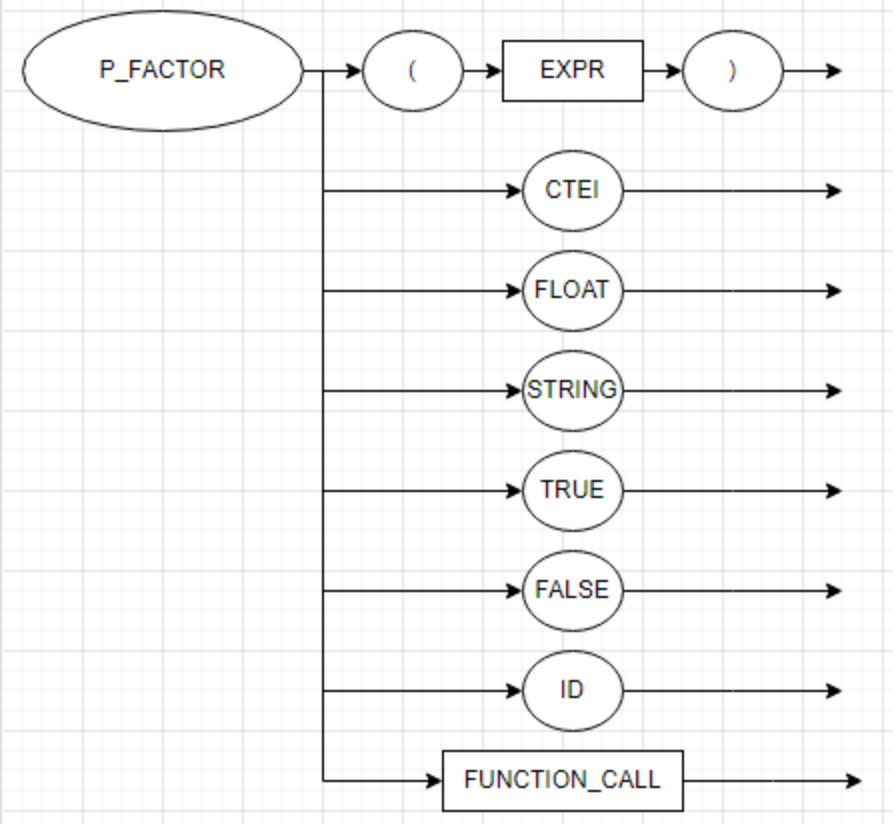
- P_EXPR



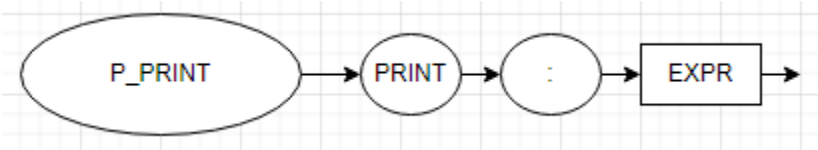

- P_TERM



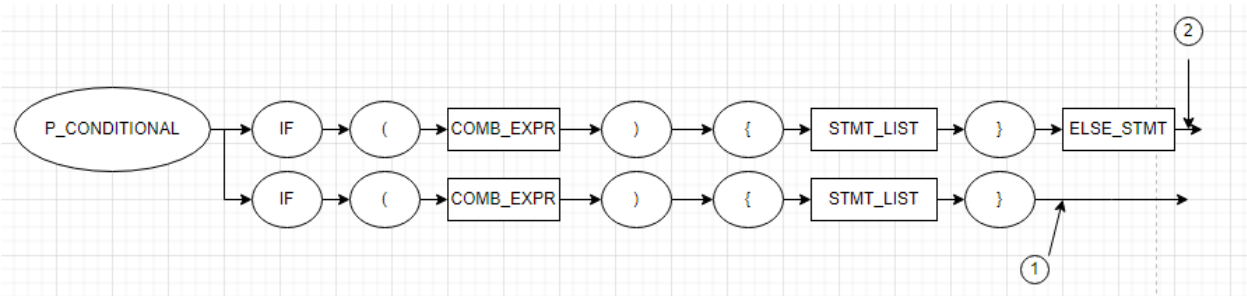

- P_FACTOR




- P_PRINT

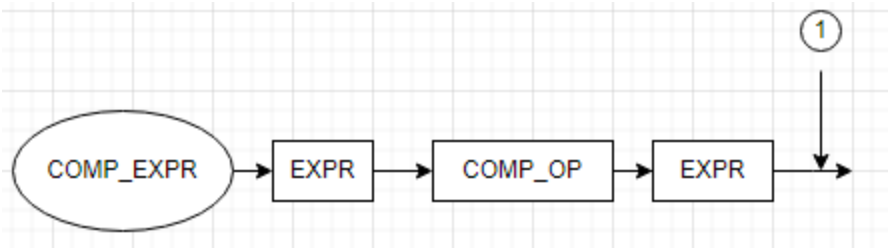



- P_CONDITIONAL



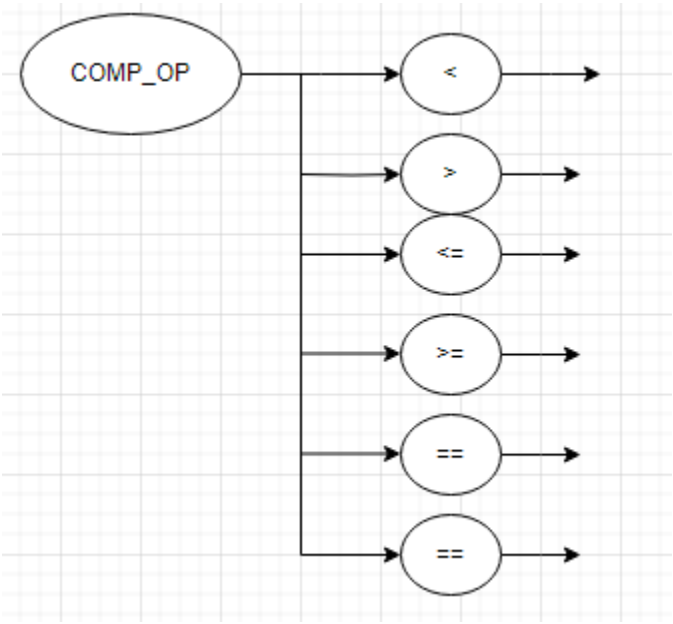

1. End_if()
2. End_else()

- P_COMP_EXPR

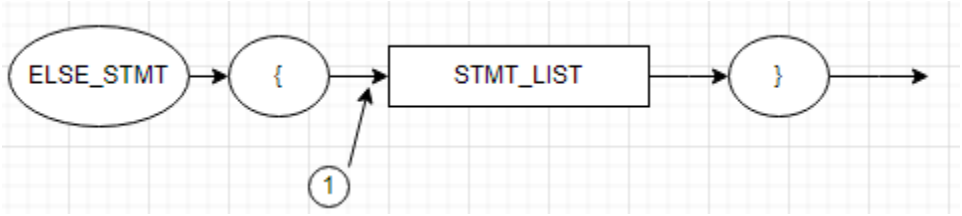



1. Start_if

- P_COMP_OP

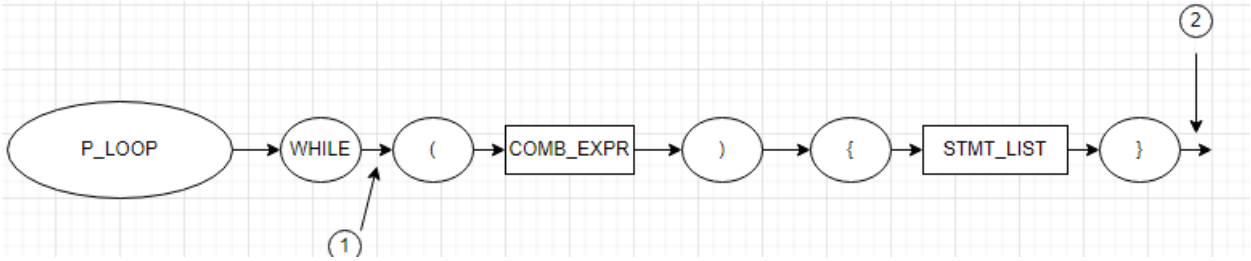



- P_ELSE_STMT



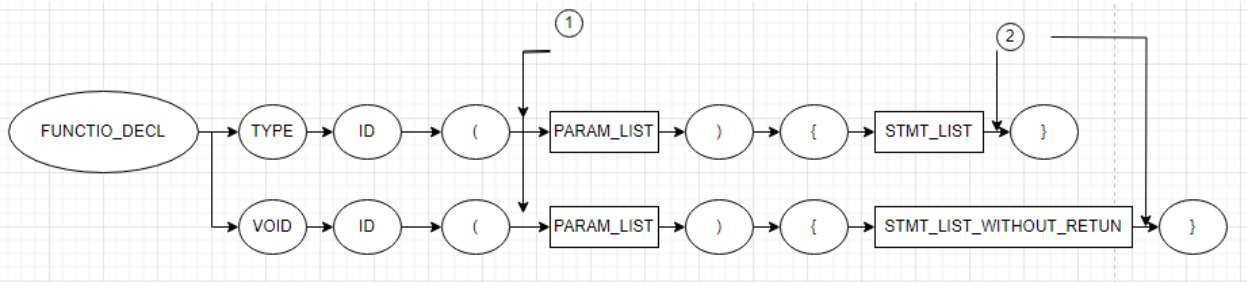

1. Else_part

- P_LOOP



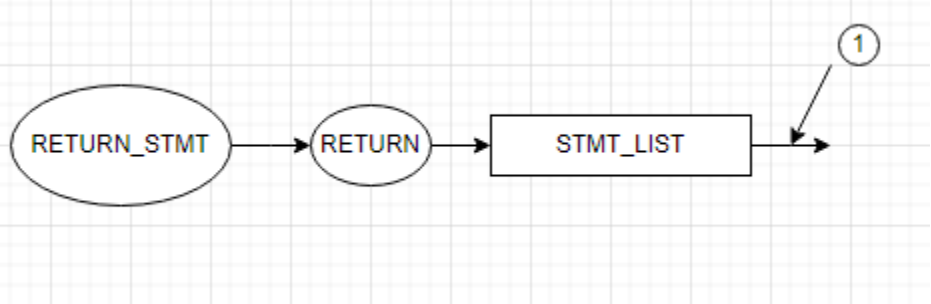
1. Start_while
2. End_while

- P_FUNCTION_DECL



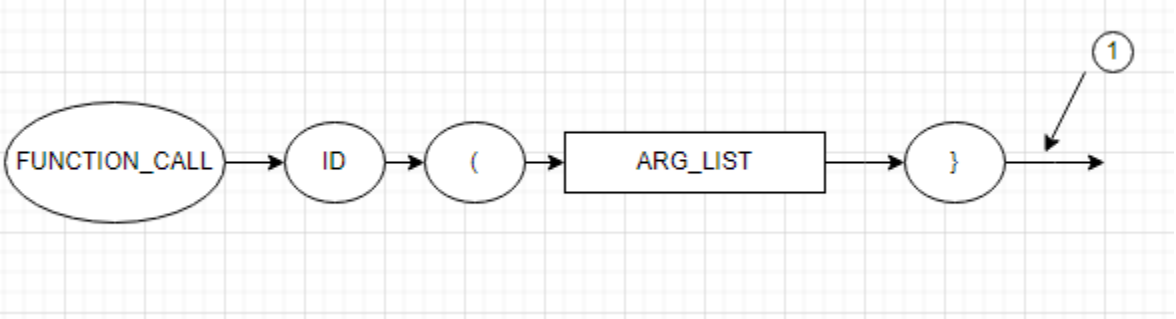

1. Create_function_scope
2. End_scopes

- P_RETURN_STMT



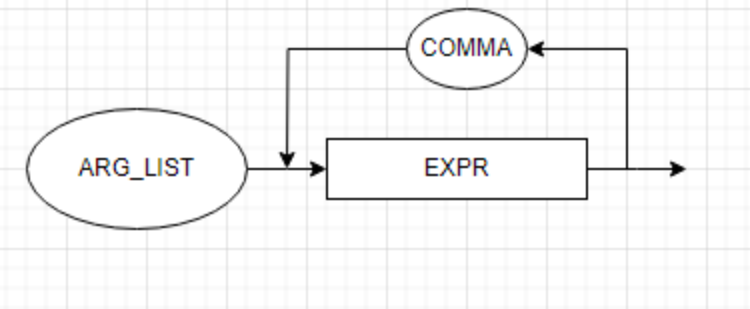

1. Handle_function_return

- P_FUNCTION_CALL

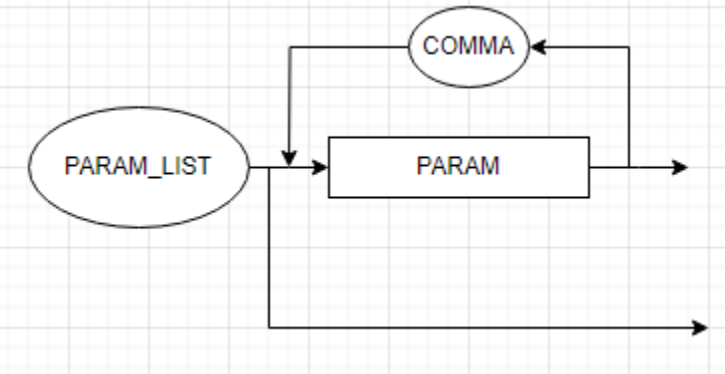



1. Handle_function_call

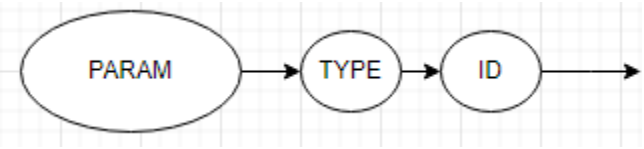
- P_ARG_LIST



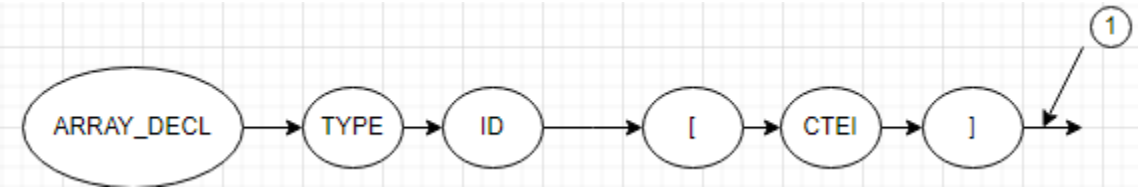

- P_PARAM_LIST




- P_PARAM

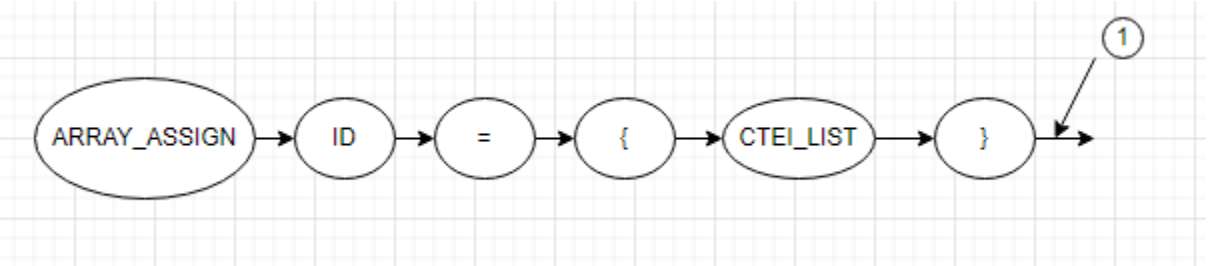



- P_ARRAY_DECL



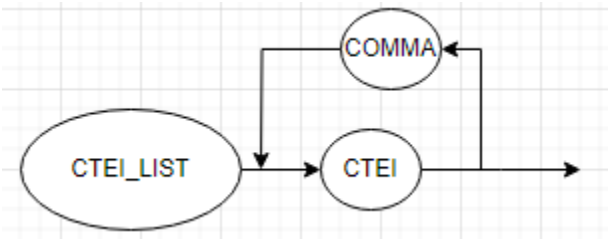

1. Declare_array

- P_ARRAY_ASSIGN

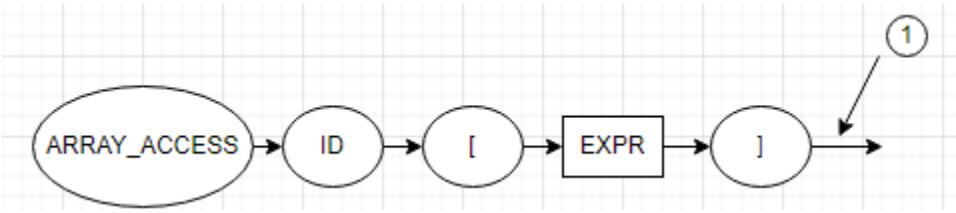



1. Handle_array_initialization

- P_CTEI_LIST




- P_ARRAY_ACCESS




1. Handle_array_access

Tabla de consideraciones semánticas

| **+ / -** | **int** | **float** |
|---|---|---|
| int | int | float |
| float | float | float |

| **‘*’ / ‘/’** | **int** | **float** |
|---|---|---|
| int | int | float |
| float | float | float |

| **‘=’** | **int** | **float** | **bool** | **string** |
|---|---|---|---|---|
| int | int | int | bool | N/A |
| float | float | float | N/A | N/A |
| bool | N/A | N/A | bool | N/A |
| string | N/A | N/A | N/A | string |

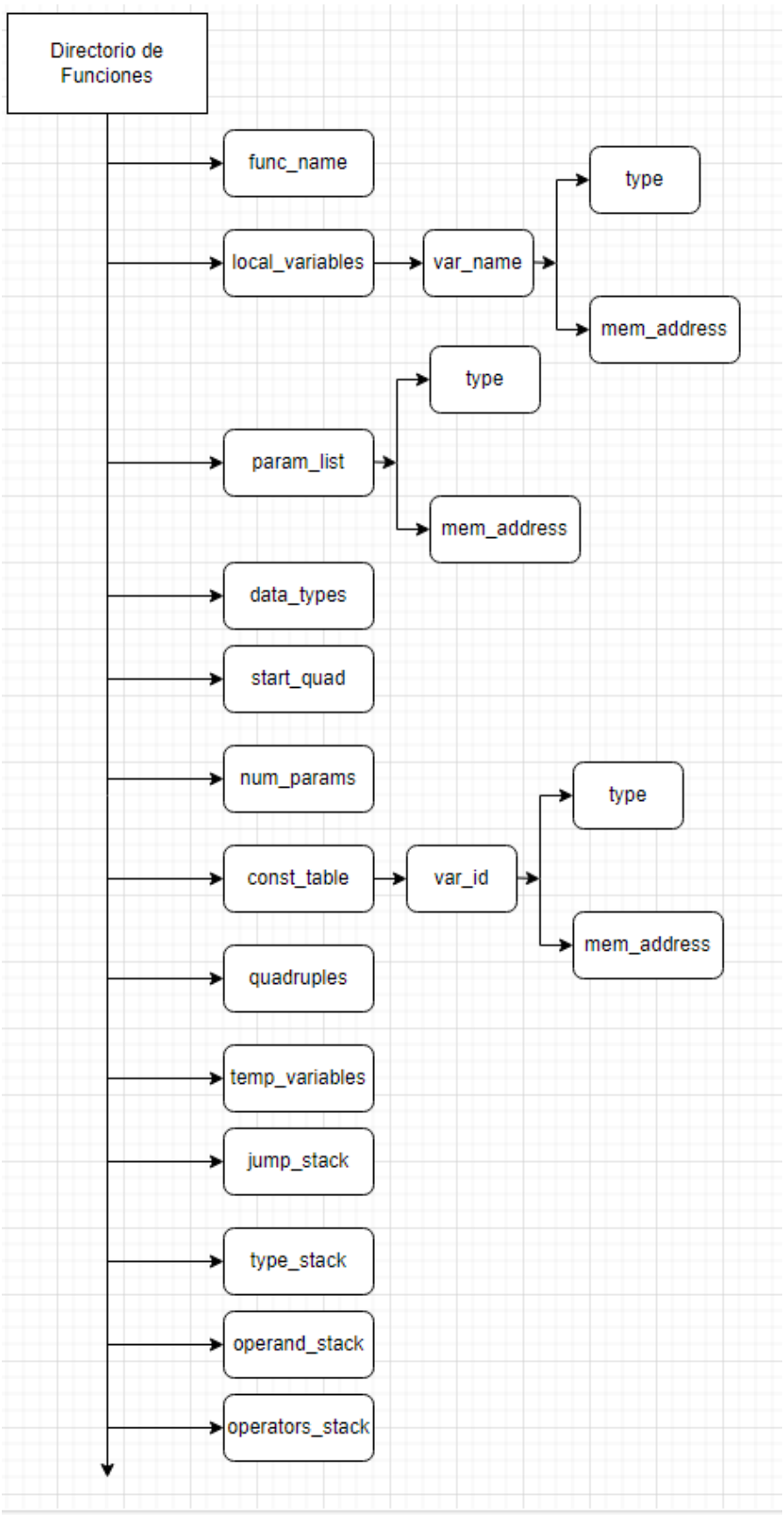
| **‘==’ / ‘!=’** | **int** | **float** | **bool** | **string** |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| Int | bool | N/A | N/A | N/A |
| Float | bool | bool | N/A | N/A |
| Bool | N/A | N/A | bool | N/A |
| String | N/A | N/A | N/A | bool |

| **‘<’ / ‘>’ / ‘<=’ / ‘>=’** | **int** | **float** |
|---|---|---|
| int | bool | bool |
| float | bool | bool |

| **‘&&’ / ‘||’** | **bool** |
|---|---|
| bool | bool |

Descripción del proceso de Administración de Memoria usado en compilación


Directorio de Funciones
func_name
local_variables
var_name
type
mem_address
param_list
type
mem_address
data_types
start_quad
num_params
const_table
var_id
type
mem_address
quadruples
temp_variables
jump_stack
type_stack
operand_stack
operators_stack

**Justificación**

- **current_scope:** Se utilizó generalmente en el análisis semántico y etapas posteriores, para determinar qué variables y funciones son accesibles en el punto actual del código.
- **data_types**: Esta lista es simplemente una enumeración de los posibles tipos de datos que el lenguaje puede manejar.
- **temp_variable_counter**: Este contador se utilizó generalmente para crear nombres únicos para las variables temporales que el compilador introduce durante varias optimizaciones.
- **quadruples**: Los cuádruplos son la forma común de representar instrucciones intermedias en la generación de código. Cada cuádruplo contiene una operación y tres argumentos.
- **operators_stack, operands_stack, types_stack**: Estos stacks se utilizaron durante el análisis semántico y la generación de código para manejar las operaciones y los operandos en las expresiones.
- **jump_stack**: Esta pila se usa generalmente para manejar las operaciones de salto, como ciclos y condicionales.
- **const_table**: Esta tabla lleva un seguimiento de todas las constantes utilizadas en el programa.
- **dir_func**: El directorio de funciones almacena información sobre cada función definida en el programa, como el nombre y tipo de sus parámetros, su tipo de retorno, y el cuádruplo inicial donde empieza su ejecución.
- **temp_variables, scopes_stack**: Estos se utilizaron para manejar las variables temporales y los contextos respectivamente, a medida que el compilador realiza su análisis y transformación del código.

## Descripción de la máquina virtual

Mi maquina virtual, fue desarrollada con la misma tecnología que el compilador, se utilizo el mismo equipo de computo y lenguaje, pero se agregaron utilerías especiales como JSON, y por supuesto PLY.

Se hizo uso de diferentes estructuras para la administración de memoria:

- **GlobalMemory, TempMemory, LocalMemory y Memory**: Estas clases manejan la asignación de direcciones de memoria para las variables. La clase Memory define las operaciones básicas para obtener y establecer valores en ciertas direcciones de memoria,

y para asignar nuevas direcciones de memoria a ciertos tipos de variables (int, float, bool, string). Las clases GlobalMemory, TempMemory y LocalMemory heredan de Memory y sirven para mantener separadas las variables globales, locales y temporales.

- **ExecutionStack**: Esta clase representa la pila de ejecución, que se utiliza para mantener el contexto de ejecución actual durante la ejecución del programa. El contexto de ejecución incluye la memoria local y la dirección de retorno cuando se hace una llamada a una función.

En el caso de la administración de memoria, use asignación de memoria estática. Durante la compilación, se asignan direcciones de memoria a las variables basadas en su tipo. Durante la ejecución, estas direcciones se utilizan para recuperar y establecer los valores de las variables. Por lo tanto, se establece una asociación directa entre las direcciones virtuales (compilación) y las reales (ejecución).

Por ejemplo, en compilación, cuando se declara una variable de tipo 'int', se le asigna una dirección de memoria del bloque de memoria reservado para enteros. Esta dirección de memoria se utiliza luego para almacenar y recuperar el valor de la variable durante la ejecución del programa.

Gracias a este enfoque de asignación de memoria, se logró una eficiencia en términos de velocidad, ya que no hay necesidad de buscar o liberar espacio durante la ejecución.

Las funciones **initialize_memory, get_local_address_for_param y execute_built_in_function(function_name)** juegan un papel importante en la ejecución, ya que **initialize_memory** inicializa los valores de la memoria constante e itera sobre la tabla de constantes, que es donde guardo las claves con los valores constantes y los valores son los atributos asociados a esas constantes (que incluyen las direcciones de memoria de las constantes). Después, convierte las constantes numéricas a sus respectivos tipos (float o int) y luego asigna estos valores a las direcciones correspondientes en la memoria constante usando el método **set().**

La función **get_local_address_for_param(),** esta función devuelve la dirección de memoria local para un parámetro de función específico. La dirección de memoria se obtiene del directorio de funciones (dir_func) usando el nombre de la función actual (current_function) y el número de parámetro (param_number). Y por último, **execute_built_in_function(function_name)** ejecuta una función incorporada con el nombre function_name. Verifica si el nombre de la función está en el diccionario de funciones incorporadas (built_in_functions). Si es así, llama a la función. Si no, lanza una excepción.

### Pruebas del funcionamiento del lenguaje

- **Expresiones regulares y aritméticas**

```
Program Hiro;

Variables {
  int x;
  int t;
  bool z;
}

main() {

  x = 5;
  t = 2;
  int r;
  r = (x * 2) + 10 * (t * 2);
  print: r;

}
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
PS C:\Users\gamer\Desktop\compi\compilador_final> python main.py input.txt
Compilacion terminada correctamente.
50
PS C:\Users\gamer\Desktop\compi\compilador_final>
```

- **Ciclos**

input.txt

```
Program Zorome;

Variables {
  int x;
  int t;
  bool z;
}

main() {

  x = 0;
  while (x <= 10) {
    if (x == 6) {
      print: "Llegamos a la mitad.";
    };
    print: x;
    x = x + 1;

  };

}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
PS C:\Users\gamer\Desktop\compi\compilador_final> python main.py input.txt
Compilacion terminada correctamente.
0
1
2
3
4
5
Llegamos a la mitad.
6
7
8
9
10
```

- **Funciones(con recursión)**

input.txt

```
Program Ichigo;

Variables {
  int x;
  int t;
  bool z;
}

int fact(int a) {
  int r;
  if (a > 1) {
    r = fact(a - 1) * a;
    return r;
  } else {
    r = 1;
    return r;
  };
}

main() {
  x = 5;
  int f;
  f = fact(x);
  print: f;
}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
PS C:\Users\gamer\Desktop\compi\compilador_final> python main.py input.txt
Compilacion terminada correctamente.
Quadruples:
0: ('Goto', None, None, 15)
1: ('>', 7000, 6000, 14000)
2: ('GotoF', 14000, None, 12)
3: ('-', 7000, 6000, 12000)
4: ('ERA', 'fact', None, None)
5: ('PARAMETER', 12000, None, 0)
6: ('GOSUB', 'fact', None, 1)
7: ('ASSIGN', 7001, None, 12001)
8: ('*', 12001, 7000, 12002)
9: ('=', 12002, None, 7001)
10: ('RETURN', None, None, 7001)
11: ('Goto', None, None, 14)
12: ('=', 6000, None, 7001)
13: ('RETURN', None, None, 7001)
14: ('ENDfunc', None, None, None)
15: ('=', 6001, None, 1000)
16: ('ERA', 'fact', None, None)
17: ('PARAMETER', 1000, None, 0)
18: ('GOSUB', 'fact', None, 1)
19: ('ASSIGN', 7001, None, 12003)
20: ('=', 12003, None, 1002)
21: ('print', None, None, 1002)
22: ('End', None, None, None)
120
```

input.txt

```
Program Miku;

Variables {
  int x;
  int t;
  bool z;
}

int pow(int base, int power) {
  int r;
  r = 0;
  if (power != 0 ) {
    r = pow(base, power - 1) * base;
    return r;
  } else {
    r = r + 1;
    return r;
  };
}

main() {
  x = 5;
  t = 7;
  int f;
  f = pow(x, t);
  print: f;
}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
PS C:\Users\gamer\Desktop\compi\compilador_final> python main.py input.txt
Compilacion terminada correctamente.
Quadruples:
0: ('Goto', None, None, 18)
1: ('=', 6000, None, 7002)
2: ('!=', 7001, 6000, 14000)
3: ('GotoF', 14000, None, 14)
4: ('-', 7001, 6001, 12000)
5: ('ERA', 'pow', None, None)
6: ('PARAMETER', 7000, None, 0)
7: ('PARAMETER', 12000, None, 1)
8: ('GOSUB', 'pow', None, 1)
9: ('ASSIGN', 7002, None, 12001)
10: ('*', 12001, 7000, 12002)
11: ('=', 12002, None, 7002)
12: ('RETURN', None, None, 7002)
13: ('Goto', None, None, 17)
14: ('+', 7002, 6001, 12003)
15: ('=', 12003, None, 7002)
16: ('RETURN', None, None, 7002)
17: ('ENDfunc', None, None, None)
18: ('=', 6002, None, 1000)
19: ('=', 6003, None, 1001)
20: ('ERA', 'pow', None, None)
21: ('PARAMETER', 1000, None, 0)
22: ('PARAMETER', 1001, None, 1)
23: ('GOSUB', 'pow', None, 1)
24: ('ASSIGN', 7002, None, 12004)
25: ('=', 12004, None, 1002)
26: ('print', None, None, 1002)
27: ('End', None, None, None)
78125
```

- **Arreglos**

input.txt

```
Program Naomi;

Variables {
  int x;
  int t;
  int arr[5];
}


main() {
  x = 5;
  t = 1;
  int r;
  arr = {5, 3, 6, 7, 8};

  while (t < 6) {
    r = arr[t] * 2;
    print: r;
    t = t + 1;
  };

}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
Compilacion terminada correctamente.
Quadruples:
0: ('Goto', None, None, 1)
1: ('=', 6000, None, 1000)
2: ('=', 6001, None, 1001)
3: ('=', 6000, None, 1002)
4: ('=', 6002, None, 1003)
5: ('=', 6003, None, 1004)
6: ('=', 6004, None, 1005)
7: ('=', 6005, None, 1006)
8: ('<', 1001, 6003, 14000)
9: ('GotoF', 14000, None, 20)
10: ('=', 1001, None, 12000)
11: ('VER', 12000, 6001, 6000)
12: ('+', 12000, 6007, 12001)
13: ('DEREF', 12001, None, 12002)
14: ('*', 12002, 6008, 12003)
15: ('=', 12003, None, 1007)
16: ('print', None, None, 1007)
17: ('+', 1001, 6001, 12004)
18: ('=', 12004, None, 1001)
19: ('Goto', None, None, 8)
20: ('End', None, None, None)
10
6
12
14
16
PS C:\Users\gamer\Desktop\compi\compilador_final>
```

## Documentación del código del proyecto

Todo el código de mi proyecto esta muy bien documentado, especificando que hace cada método y comentarios dentro de los métodos de puntos importantes, como por ejemplo estos métodos:

*# Function to create a new scope -> def* create_scope(*scope_name*):

# Function to end the current scope -> def end_scope():

# Function to create a new scope -> def create_scope(scope_name):

# Function to end the current scope -> def end_scope():

# Function to declare a variable in the symbol table -> def declare_variable(id, type):

# Function to declare a parameter variable in the symbol table -> def declare_param_variable(id, type):

# Function to declare a constant in the ConstTable -> def declare_constant(value, type):

# Function to generate a new quadruple -> def generate_quadruple(operator, left_operand, right_operand, result):

# Function to get the type of a variable -> def get_variable_type(variable_name):

# Function to get the memory address of a variable -> def get_memory_address(variable_name):

# Function to print quadruples -> def print_quadruples():

# Function to add a variable to the symbol table -> def add_variable_to_symbol_table(variable_name, variable_type, scope=None):

# Function to check if a variable has already been declared -> def check_variable_declared(variable_name, scope=None):

# Function to get the next available memory address -> def get_next_available_memory_address(data_type, is_temp=False):

# Function to add an operator to the operators stack -> def add_operator_to_stack(operator):

# Function to add an operand to the operands stack -> def add_operand_to_stack(operand, data_type):

# Function to pop an operator from the operators stack -> def pop_operator_from_stack():

# Function to pop an operand from the operands stack -> def pop_operand_from_stack():

# Function to generate a goto quadruple -> def
generate_goto_quadruple(jump_condition=None):

# Function to fill a jump in a goto quadruple -> def fill_goto_quadruple(quadruple_index,
jump_target):

# Function to declare a function in the symbol table -> def declare_function(function_name,
return_type):

# Function to check if a function has been declared -> def
check_function_declared(function_name):

# Function to start a new function scope -> def start_function_scope(function_name):

# Function to end a function scope -> def end_function_scope():

# Function to get the return type of a function -> def get_function_return_type(function_name):

# Function to declare a function parameter -> def declare_function_parameter(function_name,
parameter_type):

# Function to call a function -> def call_function(function_name):

# Function to return from a function -> def return_from_function(return_value):

# Function to handle array access -> def handle_array_access(array_id, index_expression):

# Function to handle array assignment -> def handle_array_assignment(array_id,
index_expression, assignment_expression):

# Function to handle the creation of a new array -> def handle_array_creation(array_id,
size_expression, element_type):

# Function to handle conditional statements -> def
handle_conditional_statement(condition_expression):

# Function to handle loop statements -> def handle_loop_statement(initial_condition_expression,
update_expression):