Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey, Febrero - Junio 2025

TC3002B | Desarrollo de aplicaciones avanzadas de ciencias computacionales

Documentación final del compilador

Profesor:

Ing. Elda G. Quiroga

Alumno:

José Carlos Sánchez Gómez A01174050

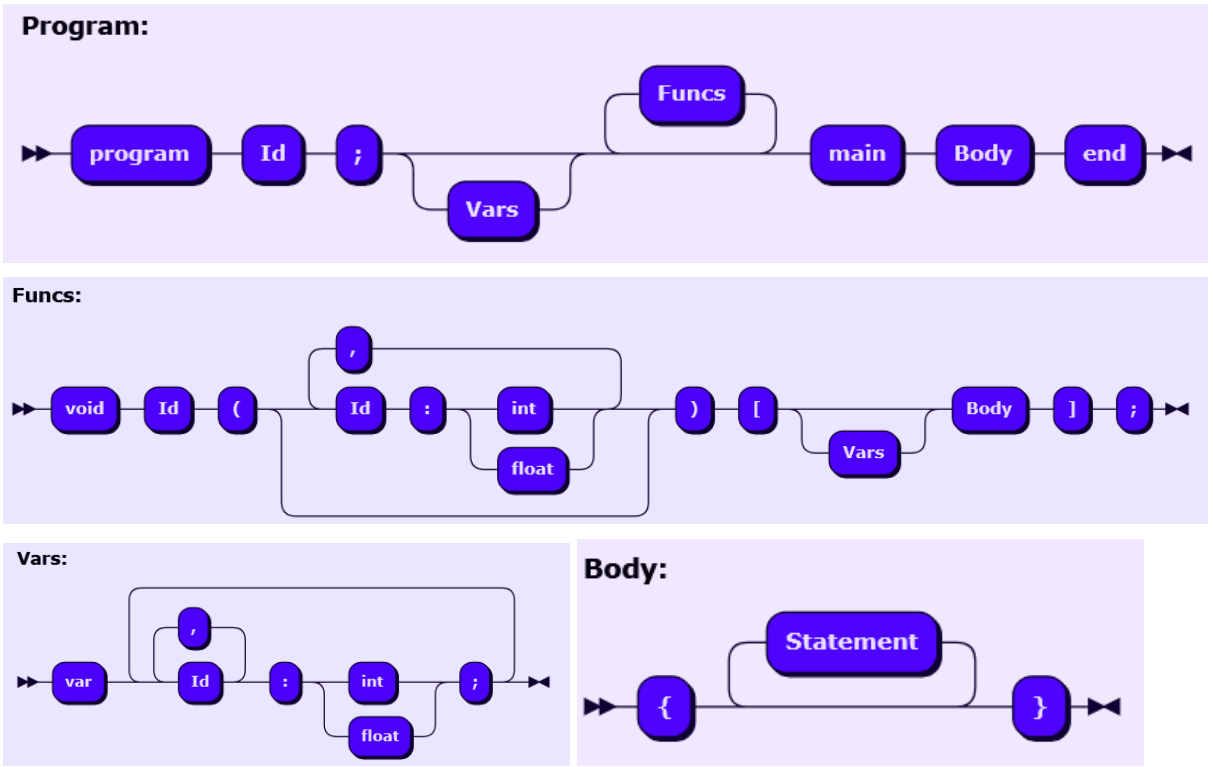
Lunes 02 de junio del 2025

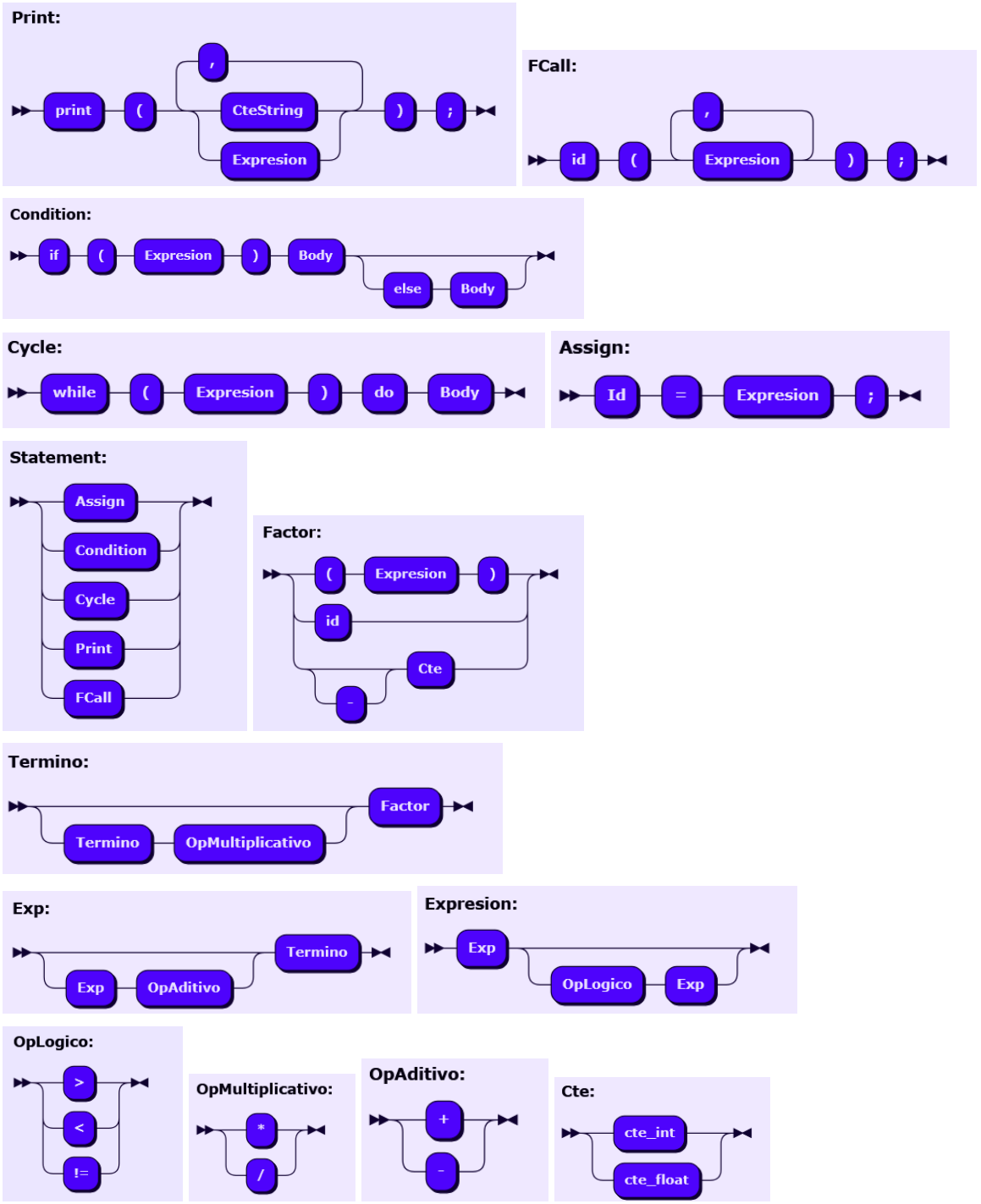
Definición del lenguaje Baby Duck

Tokens

- Identificación
  - id: [a-zA-Z_][a-zA-Z0-9_]*
- Constantes
  - cte_string: "[^"\n]*"
  - cte_int: [0-9]+
  - cte_float: [0-9]+\ .[0.9]+
- Operadores
  - + - * / > < = !=
- Símbolos
  - ; , ( ) [ ] { } .
- Palabras reservadas
  - PROGRAM, MAIN, END, VAR, INT_TYPE, FLOAT_TYPE, VOID, PRINT, WHILE, DO, IF, ELSE
- Tipos de dato
  - int
  - float

Reglas gramaticales





Herramientas usadas

Para realizar el lexer, opte por utilizar la herramienta Logos, ya que la implementación es sencilla. Sólo se definen las reglas de los tokens en un enum, y con eso ya puede tokenizar un texto. Otra razón por la cual escogí esta herramienta, es por su integración sencilla con la herramienta de mi parser, LALRPOP. Esta herramienta tiene documentación de cómo utilizarla con otros crates de rust, además de cómo funciona por sí sola la herramienta. LALRPOP ofrece un lexer, sin embargo, me pareció más sencillo de utilizar el de Logos. LALRPOP es parser bottom to top, y utiliza un lenguaje LR (Left to Right), permite usar signos de expresiones regulares, como *, +, ?, para simplificar ciertas reglas gramaticales,

y evitar ambigüedades. Además ofrece una manera de especificar precedencia y asociatividad usando macros de rust.

Definición de clases usadas

Program Manager

```rust
pub struct ProgramManager {
	pub cubo: CuboSemantico,
	pub tabla_funciones: FunctionTable,
	pub quadruplets: QuadrupletList,
	pub value_table: ValueTable,
	pub operand_stack: Vec<i32>,
	pub operator_stack: Vec<Operator>,
	pub polish_vector: Vec<String>,
	pub instruction_pointer: i32,
	pub jumps_stack: Vec<i32>,
	pub curr_function: Stack<String>,
	pub memory_stack: Stack<Memory>,
	pub function_ids: HashMap<i32, String>,
	position_before_fcall: Stack<i32>,
	upcoming_function: Option<Memory>,
}
```

Decidí crear una clase que guardara todo lo relacionado con el proceso de compilación. Esto debido a que lalrpop permite añadir código propio mientras está parseando el lenguaje. Program Manager es el que se encarga de guardar las funciones, los valores, y las variables generadas en el programa; de crear lo cuadruplos para cada instrucción, y también de ejecutarlos.

- operand_stack: Este vector es el encargado de guardar las direcciones de memoria de una variable o constante usada en alguna expresión. Es el responsable de que las operaciones puedan ser realizadas.
- operator_stack: Similar al operand_stack, sin embargo, este guarda operadores (+ - * / > < !=). Crucial para realizar las operaciones, ya que con este se garantiza que la expresión a realizar fuera válida.
- polish_vector: Fue usado para validar que las operaciones siguieran el orden adecuado de ejecución. No es necesario para el funcionamiento del compilador, sin embargo se usó para validar el correcto funcionamiento.
- instruction_pointer: Apuntador hacia el cuádruplo ejecutándose actualmente. Es fundamental en la etapa de ejecución (máquina virtual), ya que este es el responsable de que un cuádruplo se ejecute. Necesario para cuadruplos como GOTO y GOTOF.
- jumps_stack: Pila usada para guardar la posición de un cuádruplo que se necesitará completar en un futuro. Por ejemplo, cuándo hay un while o un if, y no se sabe

cuándo se terminarán los cuádruplos dentro del brazo verdadero del if. Necesitamos la posición completar el cuádruplo de GOTOF que se encarga de saltar este pedazo en caso de que la expresión sea negativa.

- curr_function: Esta pila es utilizada en los puntos neurálgicos, para determinar en qué función se encuentra actualmente. Es necesaria, ya que podemos tener variables en funciones o en el main, entonces necesitamos una forma de saber en dónde estamos y así buscarlas en la tabla de variables de esa función.
- function_ids: Diccionario utilizado para guardar el nombre de una función, con respecto a su identificador. Cada función tiene un id secuencial asignado conforme vayan apareciendo en el programa.
- position_before_fcall: Esta pila es crucial para el correcto funcionamiento del programa, ya que es la que se encarga de regresar al instruction_pointer a la posición que se encontraba antes de entrar a una función. Es necesario ya que al entrar a una función, el instruction_pointer se mueve a la posición dónde la función inicia, perdiendo así el cuadruplo que sigue después de la función.

Los campos restantes no fueron mencionados, ya que se hablará más a fondo de cada uno a continuación.

ProgramManager es el que se encarga (junto con las estructuras asociadas a él) de crear las direcciones de memoria para valores temporales, crear y rellenar cuadruplos, y de ejecutar los cuádruplos generados.

Cubo Semántico

```rust
pub enum Type {
	Int,
	Float,
	Bool,
	String,
	Error,
}

#[derive(Debug, Clone, PartialEq, Eq, Hash, Copy)]
pub enum Operator {
	Add,
	Subtract,
	Multiply,
	Divide,
	LessThan,
	GreaterThan,
	Equal,
	NotEqual,
}
```

```rust
pub struct CuboSemantico {
	pub res_operaciones: HashMap<(Type, Operator, Type), Type>,
}
```

Esta estructura fue implementada usando un diccionario, ya que proporciona una alta velocidad para buscar e ingresar valores. Es usada para comprobar que una operación sea válida. Es decir, que los tipos de datos de las variables o valores temporales sean adecuados para la operación que se necesita, además de saber que tipo de dato regresa dicha operación.

Tabla de variables

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum VarValue {
	Int(i64),
	Float(f64),
	Bool(bool),
}

#[derive(Debug, Clone, PartialEq)]
pub enum VarType {
	Int,
	Float,
}

#[derive(Debug, Clone, PartialEq)]
pub struct VariableInfo {
	pub name: String,
	pub value: VarValue,
	pub var_type: VarType,
	pub address: i32,
}


pub type VariableTable = HashMap<String, VariableInfo>;
```

La tabla de variables fue implementada, igual que el cubo semántico, con un diccionario por las mismas razones que la estructura anterior. Este diccionario se encarga de guardar los valores importantes de una variable (nombre, valor, tipo de dato, y dirección).

Tabla de funciones

```rust
#[derive(Debug, Clone, PartialEq)]
pub struct FunctionParam {
	pub var_type: VarType,
	pub name: String,
}

pub struct FunctionInfo {
	pub name: String,
	pub params: Vec<FunctionParam>,
	pub vars: VariableTable,
	pub vars_amount: Vec<Vec<i32>>,
	pub start_address: i32,
}

pub type FunctionTable = HashMap<String, FunctionInfo>;
```

La tabla de funciones es un diccionario que guarda los datos importantes de una función, y lo relaciona con el nombre de esta. Los datos guardados son:

- Nombre
- Parámetros de la función
    - Vector que guarda el nombre del parámetro y el tipo de dato de este.
- Tabla de variables
- Cantidad de variables
    - Matriz que guarda la cantidad de direcciones de memoria de variables y valores temporales de tipo entero, flotante y booleano necesarios para la ejecución de la función.
- Posición del cuádruplo dónde inicia la función

Memoria

```rust
pub enum Value {
	Int(i64),
	Float(f64),
	Bool(bool),
}

pub struct Memory {
	pub values: Vec<Vec<Vec<Value>>>,
}
```

Memory es el encargado de guardar los valores, ya sea de variables o temporales, de una función. Esta estructura es usada en dos campos de Program Manager; memory_stack y upcoming_function.

- memory_stack: Se usa una pila para guardar el bloque de memoria de una función por el comportamiento que tiene esta. Cada que se ingresa a una función se hace un push a esta pila con la memoria correspondiente a esa función, y en el momento que se acaba la función, se elimina la memoria del tope de la pila. Garantizando así que no se reserva más memoria que la necesaria para la ejecución de todo el programa.
- upcoming_function: Éste Memory se genera y se va completando conforme van apareciendo cuadruplos relacionados a una función (MEMORY, PARAM, GOSUB). Cuando es completado, se añade al memory_stack y se le vuelve a asignar un valor de None.

Ejecución del programa

Cuádruplos

```rust
#[derive(Debug, Clone)]
pub struct Quadruplet {
	pub operator: i32,
	pub arg1: i32,
	pub arg2: Option<i32>,
	pub result: Option<i32>,
}

pub struct QuadrupletList {
	quadruplets: Queue<Quadruplet>,
}
```

Esta estructura es la que se encarga de guardar las instrucciones que el programa tiene que ejecutar. Se guardan únicamente números, sin embargo, cada uno significa algo diferente dependiendo de la posición en la que se encuentre. El primer número es el operador. Cada operador (explicación después) tiene asignado un número que permite identificar cuál es. El segundo y tercer número son direcciones de memoria usadas para la operación a ejecutar. El cuarto número es la dirección de memoria de dónde se va a guardar el resultado de la operación. El tercer y cuarto número son de tipo Option, ya que hay algunos operadores que solo requieren de una dirección de memoria o de dos, y al asignarles este tipo permite tener esa flexibilidad sin tener cuádruplos de diferentes tamaños.

Operadores de Cuádruplos

```rust
1 => QuadOperator::Goto,
2 => QuadOperator::GotoV,
```

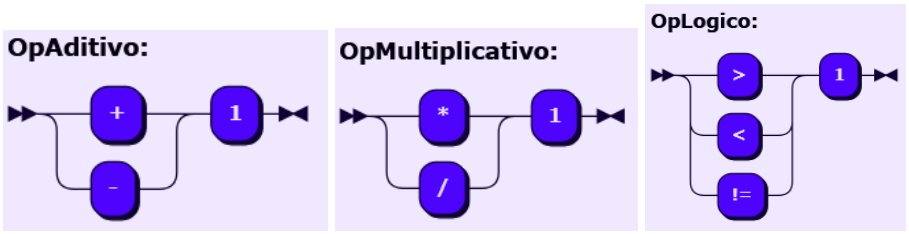
```
3 => QuadOperator::GotoF,
4 => QuadOperator::Assign,
5 => QuadOperator::Add,
6 => QuadOperator::Subtract,
7 => QuadOperator::Multiply,
8 => QuadOperator::Divide,
9 => QuadOperator::GreaterThan,
10 => QuadOperator::LessThan,
11 => QuadOperator::NotEqual,
12 => QuadOperator::Print,
13 => QuadOperator::Memory,
14 => QuadOperator::Param,
15 => QuadOperator::GoSub,
16 => QuadOperator::EndFunc,
17 => QuadOperator::EndProgram,
```

En la máquina virtual, el número de operador es convertido al operador actual para tener más claro que operador tiene cada cuádruplo.

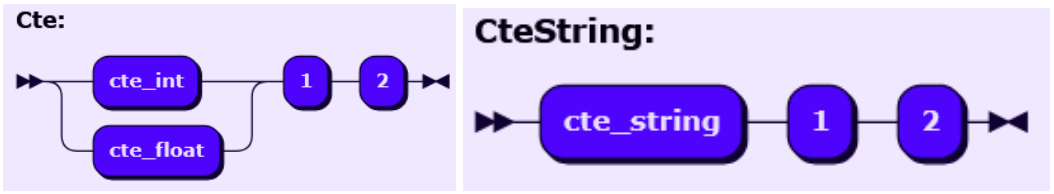
1. Este operador se encarga de mover el instruction_pointer a otra posición.
2. Este operador se encarga de mover el instruction_pointer a otra posición si el valor dentro de la dirección de memoria es verdadero. (Este operador no se usó en el lenguaje, pero lo implemente para futuras adiciones al lenguaje).
3. Este operador se encarga de mover el instruction_pointer a otra posición si el valor dentro de la dirección de memoria es falso. (Este operador no se usó en el lenguaje, pero lo implemente para futuras adiciones al lenguaje).
4. Este operador se encarga de poner un valor de una dirección de memoria a otra.
5. Este operador se encarga de agarrar el valor de las direcciones de memoria que están en el cuádruplo, y pone el resultado en la última dirección de memoria del cuádruplo.
6. Lo mismo que Add, pero este operador resta los valores.
7. Lo mismo que Add, pero este operador multiplica los valores.
8. Lo mismo que Add, pero este operador divide los valores, y checa que el dividendo no sea cero.
9. Lo mismo que Add, pero hace la comparación lógica mayor que (>).
10. Lo mismo que Add, pero hace la comparación lógica menor que (<).
11. Lo mismo que Add, pero hace la comparación lógica no es igual a (!=).
12. Este operador se encarga de imprimir en la terminal el valor que esté en la primera dirección de memoria del cuádruplo (tiene que ser un string, entero o flotante).
13. Este operando se encarga de crear una estructura Memory con los espacios de memoria necesarios para la ejecución de la función con el id que está en la segunda posición del cuádruplo.
14. Param se encarga de poner el valor del argumento en la segunda dirección de memoria del cuádruplo.
15. GoSub se encarga de encontrar la posición de inicio de la función cuyo id está en la segunda posición del cuádruplo, mover el instruction_pointer a esa posición y hacer push a memory_stack con la estructura creada en Memory.

16. EndFunc se encarga de hacer pop a memory_stack, y mover el instruction_pointer a la posición que estaba antes de entrar a la función.
17. EndProgram únicamente imprime que se acabó el programa, y unicamente está en el último cuádruplo del programa.

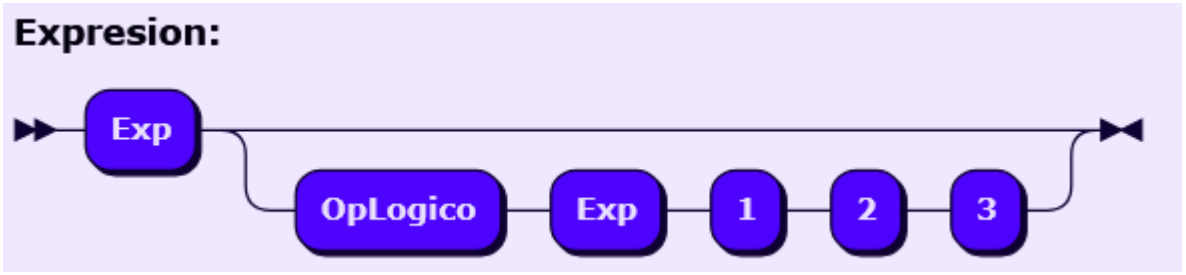
Puntos Neurálgicos



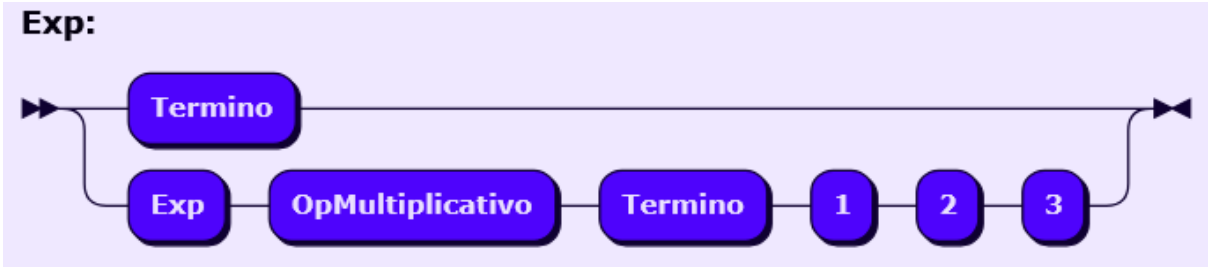
1 .- Push en el operator_stack con el operador encontrado.



1.- Guardar en la tabla de valores la constante, y obtener la dirección de memoria.
2.- Push en operand_stack la dirección de memoria.

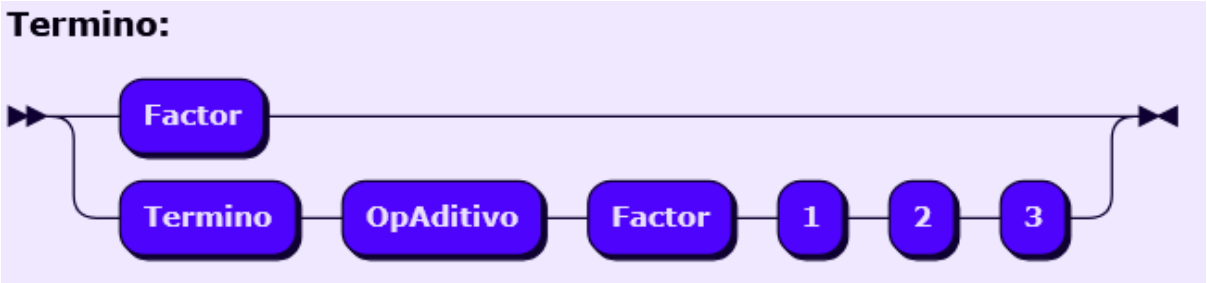


1.- Verificar que los últimos dos elementos en el operand_stack, con el OpLogico, sea una operación válida.
2.- Generar temporal booleano.
3.- Push a operand_stack del temporal.

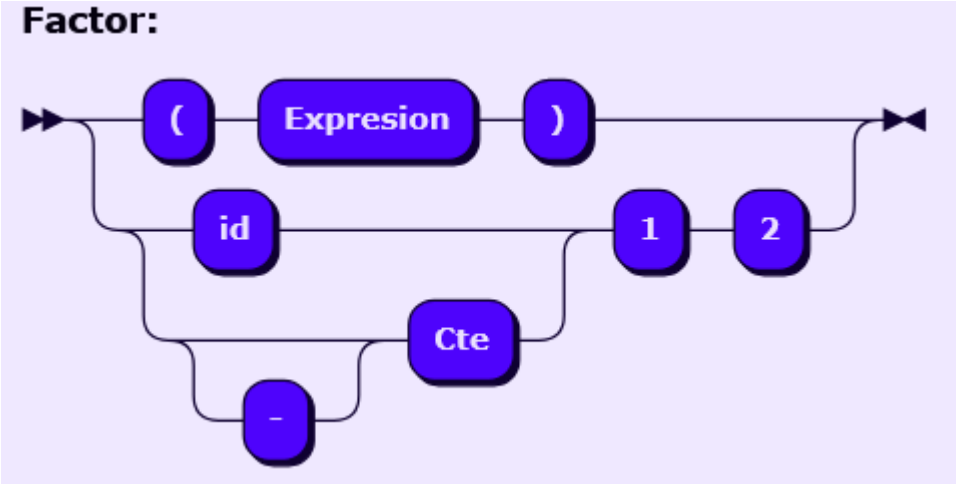


1.- Verificar que los últimos dos elementos en el operand_stack, con el OpMultiplicativo, sea una operación válida.

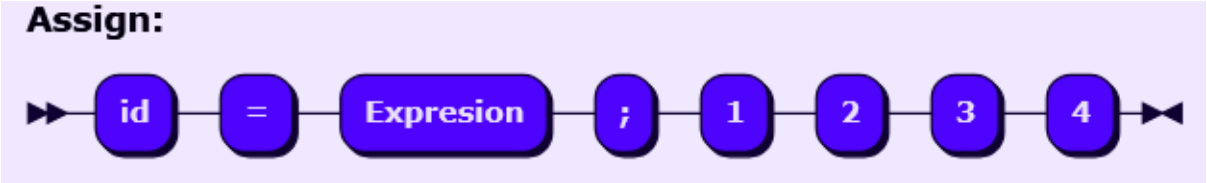
2.- Generar temporal del tipo de operación realizada (int o float).
3.- Push a operand_stack del temporal.



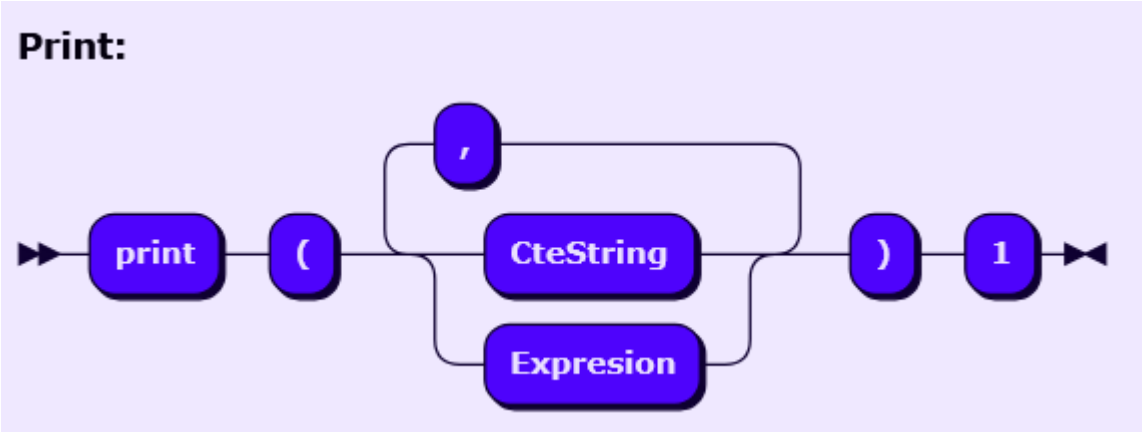
1.- Verificar que los últimos dos elementos en el operand_stack, con el OpAditivo, sea una operación válida.
2.- Generar temporal del tipo de operación realizada (int o float).
3.- Push a operand_stack del temporal.



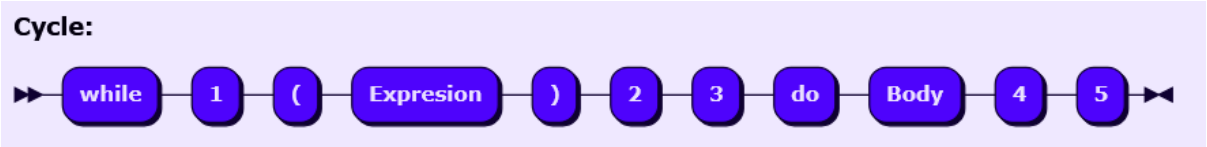
1.- Obtener dirección de memoria de variable o constante.
2.- Push en operand_stack de la dirección de memoria



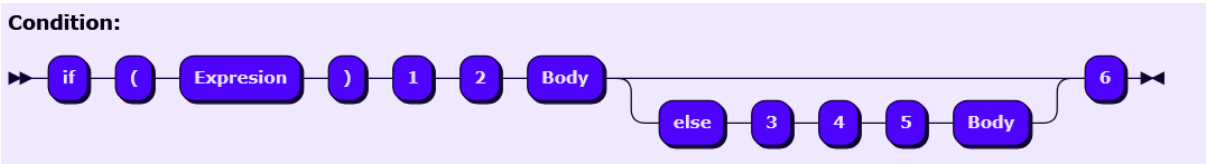
1.- Obtener dirección de memoria de la variable.
2.- Pop del operand_stack para obtener la dirección de memoria a asignar.
3.- Comprobar que el tipo de dato de la variable y el valor a asignar son iguales.
4.- Crear cuádruplo para asignar valores.



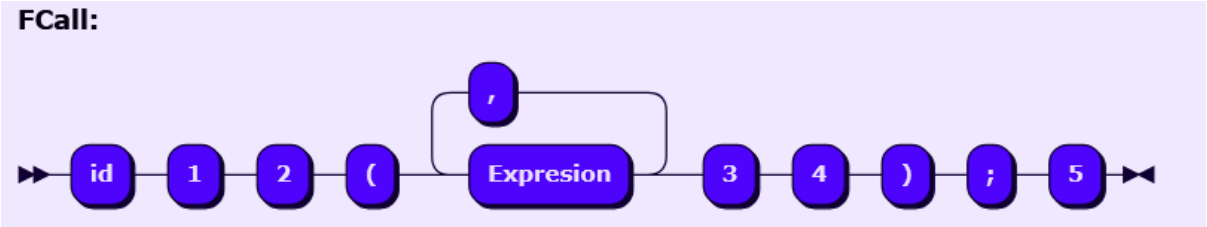
1.- Crear cuadruplo de print con el valor del tope de operand_stack, hasta que ya no hayan elementos en la pila.



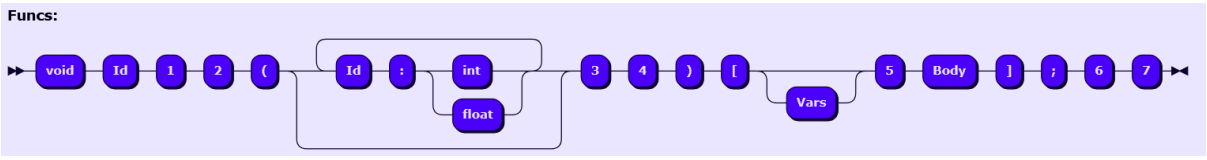
1.- Push a jumps_stack con la posición actual del instruction_pointer.
2.- Crear cuádruplo GOTOF con el tope de operand_stack, dejando el tercer valor del cuádruplo vacío (se llenará después)
3.- Push a jumps_stack de instruction_pointer - 1
4.- Pop de jumps_stack, y crear cuádruplo GOTO con el valor de jumps_stack recién popeado.
5.- Pop de jumps_stack, y rellenar el cuádruplo GOTOF con el valor de jumps_stack recién conseguido.



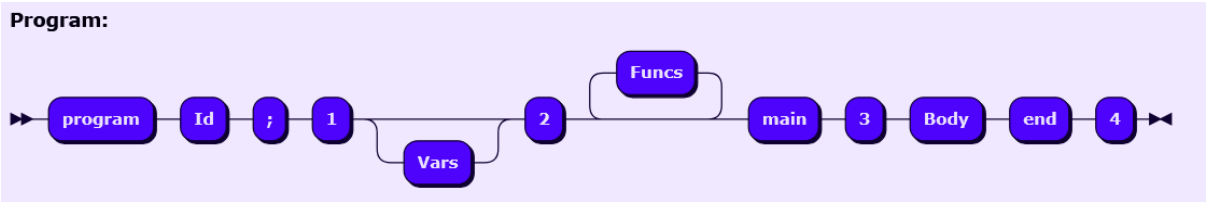
1.- Crear cuádruplo GOTOF con el tope de operand_stack, dejando el tercer espacio del cuádruplo vacío.
2.- Push en jumps_stack con el valor de instruction_pointer - 1.
3.- Crear cuadruplo GOTO con el segundo valor vacio.
4.- Pop de jumps_stack y rellenar el cuádruplo de esa posición con el valor de instruction_pointer.
5.- Push en jumps_stack con el valor de instruction_pointer - 1.
6.- Pop de jumps_stack, y rellenar el cuádruplo de esa posición con el valor de instruction_pointer.



1.- Crear cuádruplo de Memory con el id de la función.
2.- Push a curr_function con el id de la función.
3.- Pop a operand_stack hasta que esté vacío y comprobar que cada valor sea del mismo tipo que espera el parámetro.
4.- Crear cuádruplo de Param con la dirección de memoria del argumento y parámetro.
5.- Pop a curr_function, y crear cuádruplo de GOSUB con el id de la función.



1.- Push a curr_function el id de la función.
2.- Push a jumps_stack el valor actual de instruction_pointer.
3.- Crear una tabla de variables temporal para la funcion e insertar los parámetros en esta.
4.- Insertar la tabla de variables en la información de la función, en la tabla de funciones.
5.- Insertar las variables con su información en la tabla de variables de la función actual.
6.- Actualizar la cantidad de variables de la función.
7.- Crear cuádruplo de EndFunc, y pop a curr_function.



1.- Crear cuádruplo de GOTO.
2.- Insertar las variables definidas en el directorio de variables de main.
3.- Llenar cuádruplo de la posición 0 con el valor de instruction_pointer.
4.- Crear cuádruplo de EndProgram.