



# Tecnológico de Monterrey

**Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)**

**Entrega 03 - Código de Expresiones y Estatutos**

**Alumno:**  
Miguel Pedraza A01284469

6 de mayo de 2024

## Entrega 0

### Análisis:

Al investigar las opciones de herramientas para la generación automática del compilador utilizando Python como mi lenguaje para crearlo mis opciones se redujeron a las siguientes:

#### 1. PLY (Python Lex-Yacc):

- **Pros:**

- Implementación en Python, lo que facilita su integración con proyectos de Python.
- Basado en la popular herramienta Lex y Yacc, que es bastante familiar para quienes tienen experiencia en compiladores.
- Ofrece bastante flexibilidad en términos de definición de gramáticas y semántica.

- **Contras:**

- La sintaxis puede ser un poco complicada para usuarios nuevos en compiladores.
- Aunque es potente, puede ser menos eficiente en términos de rendimiento en comparación con herramientas más específicas y optimizadas ya que este fue creado principalmente para fines académicos.

- **Funcionamiento en cada etapa:**

- **Análisis Léxico:** Se puede definir tokens usando expresiones regulares, lo que facilita el análisis léxico. Además de la posibilidad de definir patrones para reconocer tokens como identificadores, números, cadenas, etc.
- **Análisis Sintáctico:** Se define la gramática usando reglas de producción, similar a Yacc. Esto permite definir la estructura sintáctica del lenguaje. Las acciones semánticas se pueden asociar con estas reglas para construir el árbol de análisis sintáctico o realizar otras operaciones semánticas.
- **Análisis Semántico:** Las acciones semánticas definidas en las reglas de producción pueden ser utilizadas para realizar verificaciones semánticas, como comprobaciones de tipos, manejo de ámbitos, etc.

#### 2. Antlr (Another Tool for Language Recognition):

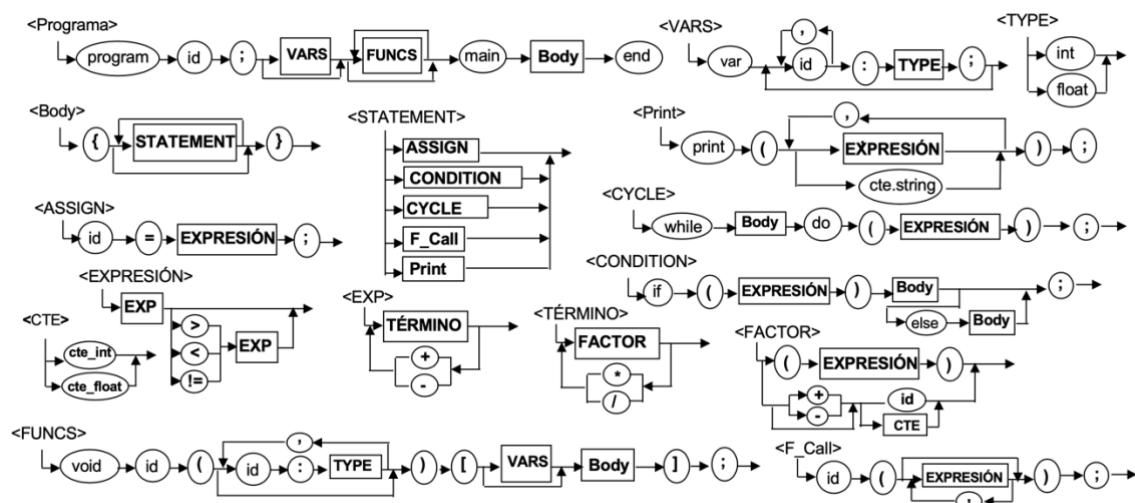
- **Pros:**

- Es muy poderoso y ampliamente utilizado en la industria.
- Tiene soporte para múltiples lenguajes de programación.

- Ofrece herramientas gráficas para ayudar en el desarrollo y depuración de gramáticas.
  - **Contras:**
    - No está escrito en Python, por lo que la integración con proyectos de Python puede ser un poco más complicada.
    - La curva de aprendizaje puede ser compleja, especialmente para los nuevos en la creación de compiladores.
  - **Funcionamiento en cada etapa:**
    - **Análisis Léxico:** Se puede generar analizadores léxicos basados en reglas que se definen en la gramática. Estas reglas pueden especificar cómo reconocer tokens en el código fuente.
    - **Análisis Sintáctico:** Con ese mismo archivo se pueden crear analizadores sintácticos que construyen árboles de análisis sintáctico de la gramática definida. Estos árboles pueden ser utilizados para realizar análisis semántico y generar código de salida.
    - **Análisis Semántico:** Se puede integrar acciones semánticas directamente en la gramática, lo que permite realizar verificaciones semánticas y otras operaciones durante el análisis sintáctico.

Por mi parte me fui más por la opción de utilizar PLY ya que considero es más adecuado para proyectos Python puros debido a su implementación en Python. Además de que me gusta la flexibilidad que ofrece PLY a la hora de definir los tokens, palabras reservadas, expresiones, reglas, etc.

## Diagrama de lenguaje:



## Proceso:

Para la parte del léxico fue bastante sencillo de realizar ya que solamente debíamos de crear nuestra tupla con todos los tokens que analizamos en nuestro diagrama y posteriormente otra tupla en donde guardábamos las palabras reservadas para finalmente unirlas en una sola tupla.

```
1  # Imports
2  import lex
3  import yacc
4
5  # Define tokens
6  tokens = (
7      "ID",
8      "CTESTRING",
9      "CTEINT",
10     "CTEFLOAT",
11     "PLUS",
12     "MINUS",
13     "TIMES",
14     "DIVIDE",
15     "GREATERTHAN",
16     "LESSTHAN",
17     "NOTEQUAL",
18     "ASSIGN",
19     'LPAREN',
20     'RPAREN',
21     "LBRACKET",
22     "RBRACKET",
23     "LBRACE",
24     "RBRACE",
25     "COMMA",
26     "COLON",
27     "ENDINSTRUC"
28 )
49  # Define reserved keywords
50  reserved = {
51      'if' : 'IF',
52      'else' : 'ELSE',
53      'while' : 'WHILE',
54      "program": "PROGRAM",
55      "main": "MAIN",
56      "end": "END",
57      "var": "VAR",
58      "print": "PRINT",
59      "void": "VOID",
60      "while": "WHILE",
61      "do": "DO",
62      "int": "INT",
63      "float": "FLOAT"
64 }
```

Posteriormente se crearon tanto expresiones regulares tanto para algunos tokens individuales como funciones donde identificábamos tokens más complejos como IDs, números enteros, flotantes, etc.

```

30  # Define token regular expressions
31  t_PLUS = r'\+'
32  t_MINUS = r'-'
33  t_TIMES = r'\*'
34  t_DIVIDE = r'/'
35  t_GREATERTHAN = r'\>'
36  t_LESSTHAN = r'\<'
37  t_NOTEQUAL = r'\!='
38  t_ASSIGN = r'\='
39  t_LPAREN = r'\('
40  t_RPAREN = r'\)'
41  t_LBRACKET = r'\['
42  t_RBRACKET = r'\]'
43  t_LBRACE = r'\{'
44  t_RBRACE = r'\}'
45  t_COMMA = r'\,'
46  t_COLON = r'\:'
47  t_ENDINSTRUC = r'\;'

48

```

```

69  # Define a rule for IDs
70  def t_ID(t):
71      r'[a-zA-Z_][a-zA-Z_0-9]*'
72      t.type = reserved.get(t.value, 'ID') # Check for reserved words
73      return t
74
75  # Define a rule for floating numbers
76  def t_CTEFLOAT(t):
77      r'[-+]?[0-9]*\.[0-9]+'
78      t.value = float(t.value)
79      return t
80
81  # Define a rule for integer numbers
82  def t_CTEINT(t):
83      r'[0-9]+'
84      t.value = int(t.value)
85      return t
86
87  # Define a rule for strings
88  def t_CTESTRING(t):
89      r'"[a-zA-Z_][a-zA-Z_0-9]*"'
90      t.value = str(t.value)
91      return t
92
93  # Define how to handle whitespace
94  t_ignore = ' \t'
95
96  # Define a rule so we can track line numbers
97  def t_newline(t):
98      r'\n'
99      t.lexer.lineno += len(t.value)
100

```

Eso sería todo por parte del léxico, por parte de la sintaxis para crear el parser tenemos que agregar todas nuestras reglas gramaticales como funciones en donde cada función representa cada regla. Dentro de ellas se tiene una convención de utilizar los tokens en mayúsculas y cualquier otra variable o regla en minúsculas.

```

109  # ----- Define grammar rules (Syntax Parser) -----
110
111 # Define parser precedence
112 precedence = (
113     ('left', 'PLUS', 'MINUS'),
114     ('left', 'TIMES', 'DIVIDE'),
115 )
116
117 def p_prog(p):
118     "prog : PROGRAM ID ENDINSTRUCCION vars funcs MAIN body END"
119
120 def p_vars(p):
121     """vars : VAR variables
122     | empty"""
123
124 def p_variables(p):
125     "variables : list_ids COLON type ENDINSTRUCCION mas_vars"
126
127 def p_list_ids(p):
128     "list_ids : ID mas_ids"
129
130 def p_mas_ids(p):
131     """mas_ids : COMMA list_ids
132     | empty"""
133
134 def p_mas_vars(p):
135     """mas_vars : variables
136     | empty"""
137
138 def p_type(p):
139     """type : INT
140     | FLOAT"""

```

Finalmente, al terminar esto podemos correr un input test donde probamos crear un programa usando las reglas gramaticales e imprimir la lista de tokens generada por el scanner y el parser donde nos muestra si nuestra sintaxis ha sido válida.

```

# Test
basic_program_data = """
program test;

var i, j: int;

void max(i: int, j: int) [
    {
        if (i > j) {
            print(i);
        } else {
            print(j);
        };
    }
]

main
{
    i = 5;
    j = 10;
    max(i, j);

    while { check = i < 10; } do ( i + 1 );
}
end
"""

```

```
----- Scanner -----  
LexToken(PROGRAM,'program',2,1)  
LexToken(ID,'test',2,9)  
LexToken(ENDINSTRUC,';',2,13)  
LexToken(VAR,'var',4,16)  
LexToken(ID,'i',4,20)  
LexToken(COMMA,',',4,21)  
LexToken(ID,'j',4,23)  
LexToken(COLON,':',4,24)  
LexToken(INT,'int',4,26)  
LexToken(ENDINSTRUC,';',4,29)  
LexToken(VOID,'void',6,32)  
LexToken(ID,'max',6,37)  
LexToken(LPAREN,'(',6,40)  
LexToken(ID,'i',6,41)  
LexToken(COLON,':',6,42)  
LexToken(INT,'int',6,44)  
LexToken(COMMA,',',6,47)  
LexToken(ID,'j',6,49)  
LexToken(COLON,':',6,50)  
LexToken(INT,'int',6,52)  
LexToken(RPAREN,')',6,55)  
LexToken(LBRACKET,'[',6,57)  
LexToken(LBRACE,'{',7,63)  
LexToken(IF,'if',8,73)  
LexToken(LPAREN,'(',8,76)  
LexToken(ID,'i',8,77)  
LexToken(GREATERTHAN,'>',8,79)  
LexToken(ID,'j',8,81)  
LexToken(RPAREN,')',8,82)  
LexToken(LBRACE,'{',8,84)  
LexToken(PRINT,'print',9,98)  
LexToken(LPAREN,'(',9,103)  
LexToken(ID,'i',9,104)
```

```
----- Parser -----  
  
program test;  
  
var i, j: int;  
  
void max(i: int, j: int) [  
{  
    if (i > j) {  
        print(i);  
    } else {  
        print(j);  
    };  
}  
]  
  
main  
{  
    i = 5;  
    j = 10;  
    max(i, j);  
  
    while { check = i < 10; } do ( i + 1 );  
}  
end
```

## Actualizaciones entrega 1

Al ir revisando todo lo que se necesitaba para el léxico y la sintaxis me di cuenta de que de tenía algunos errores dentro de mi gramática y expresiones regulares. Dentro de los cambios realizados por parte de las expresiones regulares decidí ser más específico y declarar cada uno de los tokens de manera individual ya sea usando expresiones regulares sencillas o agregando las funciones para definir que son los números y ids validos junto con palabras reservadas:

```
# Define token regular expressions
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_GREATERTHAN = r'\>'
t_LESSTHAN = r'\<'
t_NOTEQUAL = r'\!='
t_ASSIGN = r'\='
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_COMMA = r'\,'
t_COLON = r'\:'
t_ENDINSTRUC = r'\;'

# Define a rule for IDs
def t_ID(t):
    r'[a-zA-Z][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID') # Check for reserved words
    return t

# Define a rule for floating numbers
def t_CTEFLOAT(t):
    r'[-+]?[0-9]*\.[0-9]+'
    t.value = float(t.value)
    return t

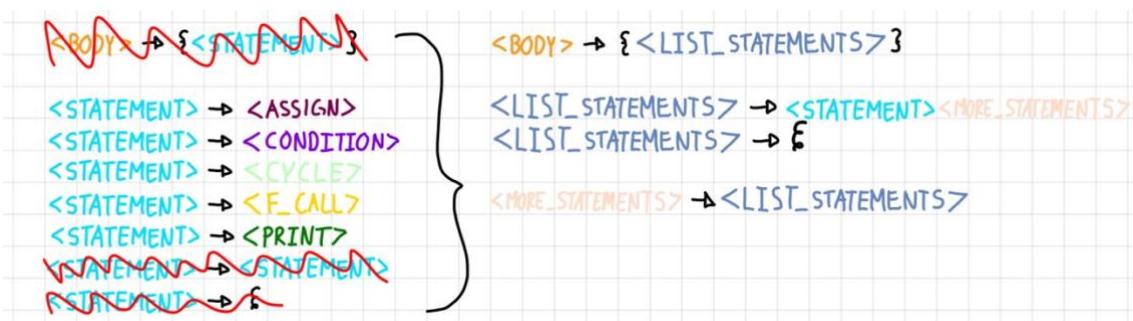
# Define a rule for integer numbers
def t_CTEINT(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

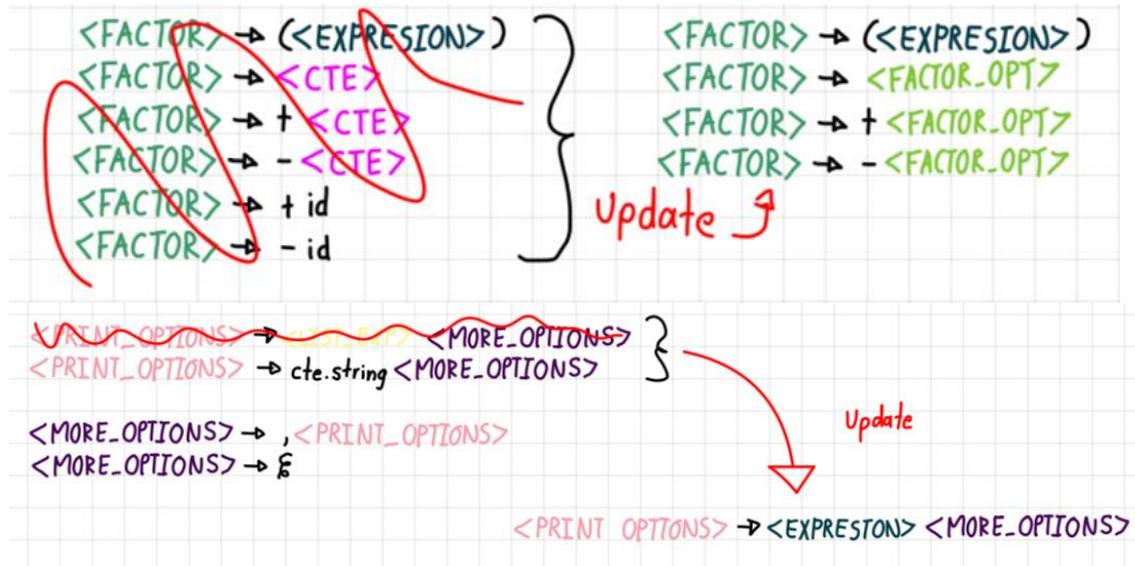
# Define a rule for strings
def t_CTESTRING(t):
    r'\"([^\"]*)\"'
    t.value = str(t.value)
    return t

# Define how to handle whitespace
t_ignore = ' \t'

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

Y por parte de la gramática tuve que actualizar las siguientes reglas por errores o para mejorar la lógica:



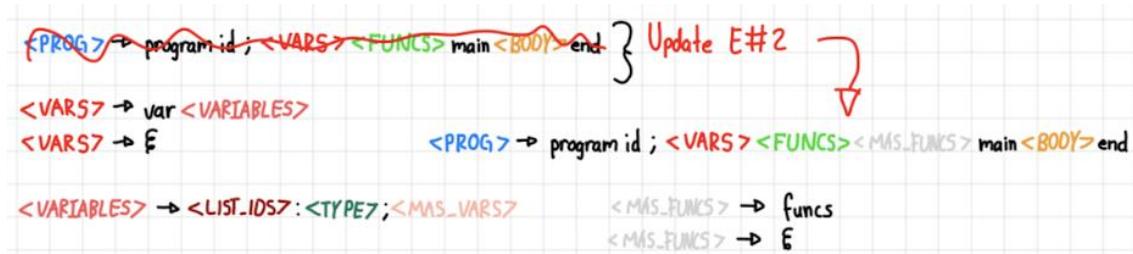


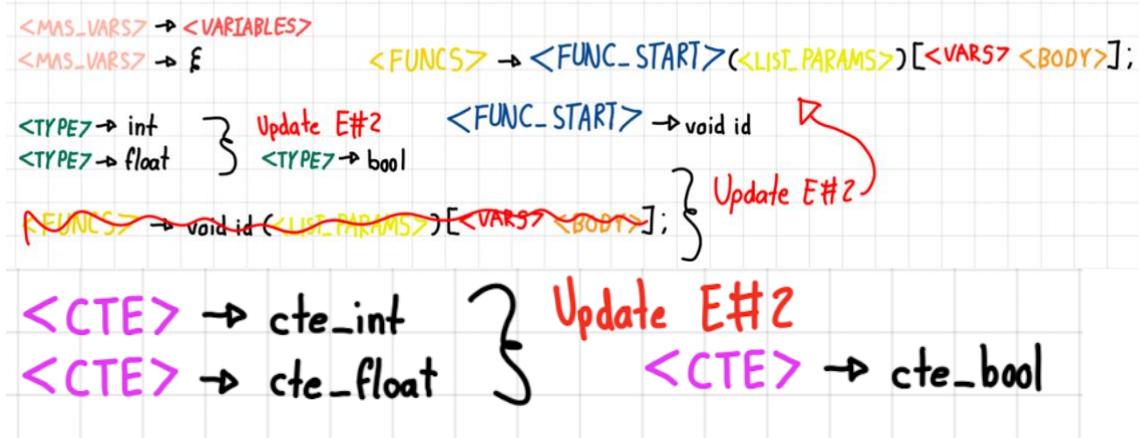
## Actualizaciones entrega 2

Dentro de la siguiente iteración del reto ahora finalmente nos adentramos a revisar la semántica de nuestro proyecto. Para esto trabajamos en conjunto con el proceso de parseo sintáctico agregando puntos neurológicos en donde agregamos funciones extras para manejar principalmente 3 cosas: el diseñar la tabla de consideraciones semántica (cubo semántico), la creación del directorio de funciones con sus respectivas tablas de variables y el manejo de errores de asignación, variables/funciones ya declaradas y errores en los tipos al combinarlos unos con otros.

Para esto primero que nada tuvimos que dar un retoque a nuestras reglas gramaticales. En donde principalmente nos encargamos de agregar un nuevo tipo, el cual es el `bool`. Esto para facilitar unas nuestras expresiones en condiciones o al realizar comparativas con operadores relacionales.

Por otra parte, un cambio que fue muy importante fue el que teníamos un error en donde solo permitíamos la declaración de una sola función, lo que hizo que se tuviera que agregar una regla más para encargarse de la repetición, además de partir la regla inicial de funciones para poder detectar que encontramos un `void`, que según nuestro diagrama de lenguaje indica que viene una función, y dentro de esa nueva regla cambiamos scope para agregar variables locales a la función.





### Directorio de funciones y variables:

El punto crucial de la entrega estaba en lograr almacenar todas las variables y funciones considerando su scope. Para esto se tenían que considerar principalmente la necesidad de diccionarios en Python para simular un hash table y stacks para ir guardando nombres de variables, sus tipos y tipos de operandos en ciertas partes del sistema.

Primero desde un inicio tenemos instanciado el directorio de funciones, que esta creado con un diccionario de Python para simular un hash table, en donde por default ya estamos en el scope global y lo almacenamos como si fuera una función que dentro tendrá otro diccionario que de variables donde se almacenará los nombres y tipos de variables.

```
funcs_dir = {
    'global': {
        'vars': {}
    }
}

current_scope = 'global'
```

Algo muy importante a la hora de realizar todo este proceso es que PLY realiza su parsing utilizando LR Bottom Up. Esto agrego una dificultad un poco más grande a la hora de entender que gramáticas pasan primero y como llevar a cabo la creación y estructuración de las instrucciones. Por ejemplo, tomando en cuenta la regla gramatical que tenemos debajo. La manera en la que se hace el parse es que se va hasta algún lead node por decirlo de una forma y se empieza a resolver desde ahí hacia atrás. Entonces en el caso de la regla de abajo es que entrará dentro de la instrucción de vars y se irá hasta la regla más baja posible, en este caso es cuando lleguemos al tipo de dato ya que tiene solamente 3 opciones entonces de ahí se ira regresando lo almacenado hacia instrucciones pasadas hasta regresar y terminar con la regla de vars y seguir con lo de funcs.

```

def p_prog(p):
    "prog : PROGRAM ID ENDINSTRUC vars funcs mas_funcs MAIN body END"
    p[0] = ('prog', p[2], p[4], p[5], p[7])

```

Con esto en mente al ya tener el scope en global por default y como lo primero que vamos a realizar son todas las reglas de variables y desde aquí podíamos realizar el proceso de almacenado de variables a su respectivo directorio. Para esto necesitamos 2 stacks, uno para los nombres de las variables y otro para el tipo. Como mencionamos al entrar y pasar por todas las reglas de variables llegaremos a la regla de tipo y es donde guardaremos el tipo que estamos teniendo.

```

def p_type(p):
    """type : INT
            | FLOAT
            | BOOL"""
    p[0] = p[1]
    current_type_stack.append(p[1])

```

Con esto se ira pasando a través de todas las reglas y llegaremos a la parte de variables donde ya tendremos la lista de ids, es decir los nombres de variables y debido a que nuestro diagrama nos permite agregar n variables con el mismo tipo y después agregar otras n variables con otro tipo esto hace que podamos guardar el tipo que será usado para las n variables que vienen de list ids y en caso de que vengan más se actualice al recibir otro tipo. Finalmente debemos revisar si la var que estamos revisando ya existía, si es así lanzamos error, de lo contrario almacenamos la variable en las variables de la función con el current scope.

```

def p_variables(p):
    "variables : list_ids COLON type ENDINSTRUC mas_vars"

    # Get every var name from list ids
    for var_name in p[1]:
        current_var_stack.append(var_name)

    # Get the type of the n coming variables
    vars_type = current_type_stack.pop()

    # Pop the vars and types and add them to the directories
    while current_var_stack:
        var_name = current_var_stack.pop()

        # Check scope
        if current_scope in funcs_dir:
            if var_name in funcs_dir[current_scope]['vars']:
                raise ReferenceError(f"'{var_name}' variable has already been declared")
            else:
                funcs_dir[current_scope]['vars'][var_name] = vars_type
        else:
            funcs_dir[current_scope] = {'vars': {var_name: vars_type}}

```

Con esto ya tenemos también las bases para realizar el almacenado de variables funciones tan sencillo como al detectar en nuestra nueva regla gramatical que viene una función poder cambiar el current scope.

```
def p_funcs(p):
    """ funcs : func_start LPAREN list_params RPAREN LBRACKET vars body RBRACKET ENDINSTRUC
    | empty"""

def p_func_start(p):
    """func_start : VOID ID"""

    # Set the global variable current_scope to be the new function
    global current_scope
    current_scope = p[2]

    print("----- Current scope: ", current_scope)

    # Check if we already have the function declared in out func dir
    if current_scope in funcs_dir:
        raise ReferenceError(f"'{current_scope}' function has already been declared")

    p[0] = p[2]
```

Así podemos nombrar el scope igual que nuestras funciones y revisar si estas ya existían o no dentro del directorio de funciones, en caso de que no se encuentren en el directorio vamos a continuar. Y según nuestra regla lo que viene primero son la lista de parámetros en donde podemos obtener el nombre y tipo del parámetro y revisar si existen o no en nuestra función, sino lo agregamos.

```
def p_list_params(p):
    """list_params : ID COLON type mas_params
    | empty"""

    # Get current scope global variable
    global current_scope

    if len(p) == 5:
        # Get name and type of param
        param_name = p[1]
        param_type = p[3]

        # Check scope
        if current_scope in funcs_dir:
            # Check if we have one already declared within the scope
            if param_name in funcs_dir[current_scope]['vars']:
                raise ReferenceError(f"'{param_name}' variable has already been declared in this function '{current_scope}'")
            else:
                funcs_dir[current_scope]['vars'][param_name] = param_type
        else:
            funcs_dir[current_scope] = {'vars': {param_name: param_type}}

        p[0] = [(p[1], p[3])] + p[4]
    else:
        p[0] = []
```

Con esto nos encargamos de todos los parámetros. Y como nuestra regla de variables que vimos anteriormente ya se encarga de revisar el scope para ver si existe la variable de forma local o global sino la agrega o marca el error.

Gracias a eso podemos almacenar nuestras variables y funciones sin ningún problema. Y para manejar los errores por variables no declaradas o malas asignaciones debido al tipo de dato se creo la tabla de consideraciones semánticas o el cubo semántico en donde se especifica dependiendo del tipo del operando

izquierdo, derecho y el operador el tipo que se espera de resultado. Con esto podemos validar que se lleven a cabo operaciones válidas de acuerdo con el tipo de dato que se va a utilizar.

```
# Definir los tipos y operadores
tipos = ['int', 'float', 'bool']
operadores = ['+', '-', '*', '/', '!=', '<', '>']

# Crear un cubo semántico vacío
cubo_semántico = {}

# Inicializar el cubo semántico
for tipo1 in tipos:
    cubo_semántico[tipo1] = {}
    for tipo2 in tipos:
        cubo_semántico[tipo1][tipo2] = {}
        for operador in operadores:
            cubo_semántico[tipo1][tipo2][operador] = None

# Ejemplo de llenado del cubo semántico para algunos operadores
cubo_semántico['int']['int']['+'] = 'int'
cubo_semántico['int']['int']['-'] = 'int'
cubo_semántico['int']['int']['*'] = 'int'
cubo_semántico['int']['int'][ '/') = 'float'

cubo_semántico['float']['float']['+'] = 'float'
cubo_semántico['float']['float']['-'] = 'float'
cubo_semántico['float']['float']['*'] = 'float'
cubo_semántico['float']['float'][ '/') = 'float'

cubo_semántico['int']['float']['+'] = 'float'
cubo_semántico['float']['int']['+'] = 'float'
cubo_semántico['int']['float']['-'] = 'float'
cubo_semántico['float']['int']['-'] = 'float'
cubo_semántico['int']['float']['*'] = 'float'
cubo_semántico['float']['int']['*'] = 'float'
cubo_semántico['int']['float'][ '/') = 'float'
cubo_semántico['float']['int'][ '/') = 'float'

# Ejemplo para operadores relacionales
cubo_semántico['int']['!='] = 'bool'
cubo_semántico['int']['<'] = 'bool'
```

```
# Function that returns the expected type of an operation
def get_expected_type(left_operand, right_operand, operator):
    if cubo_semántico[left_operand][right_operand][operator]:
        return cubo_semántico[left_operand][right_operand][operator]
    else:
        raise TypeError(f"'{operator}' is not supported between instances of '{left_operand}' and '{right_operand}'")
```

Posteriormente con esto creado se necesitaba tomar en cuenta las operaciones que se llevan a cabo e ir almacenando los tipos de dato para determinar si la asignación es correcta o si existen las variables que se están utilizando. Para eso tenemos que ir dentro del body y su operación más baja es llegar a los opciones de factor que son o un ID que representa una variable o una constante que según nuestro diagrama solo puede ser entero, flotando y ahora bool con lo que agregamos en esta entrega.

```

def p_factor_opt(p):
    """factor_opt : cte
    | ID"""

    # Get operand type if it exist in our vars table
    operand_type = get_operand_type(p[1])

    # Add it to the operand stack to keep track
    operand_stack.append(operand_type)

    p[0] = p[1]

```

La función es muy importante ya que gracias a esta función es donde podemos identificar dependiendo de si estamos en global o no si debemos de realizar la búsqueda de las variables primero en nuestro scope local y si no se encuentra buscar en el global antes de decidir si hay error o no. Con esto podemos darnos cuenta si existen o no variables que ya fueron declaradas en el mismo scope, de lo contrario mandamos su tipo correspondiente.

```

# Function that checks the type of a given operand
def get_operand_type(operand):
    # Initialize the return var
    operand_type = ""

    # First check if we are not in global in order to also look into those variables
    if current_scope != "global":
        # Check the types of the operands
        if operand in funcs_dir[current_scope]["vars"]:
            operand_type = funcs_dir[current_scope]["vars"][operand]
        elif operand in funcs_dir["global"]["vars"]:
            operand_type = funcs_dir["global"]["vars"][operand]
        else:
            if isinstance(operand, bool):
                operand_type = "bool"
            elif isinstance(operand, int):
                operand_type = "int"
            elif isinstance(operand, float):
                operand_type = "float"
            else:
                raise ReferenceError(f"'{operand}' is not defined")
    else:
        if operand in funcs_dir[current_scope]["vars"]:
            operand_type = funcs_dir[current_scope]["vars"][operand]
        else:
            if isinstance(operand, bool):
                operand_type = "bool"
            elif isinstance(operand, int):
                operand_type = "int"
            elif isinstance(operand, float):
                operand_type = "float"
            else:
                raise ReferenceError(f"'{operand}' is not defined")

    return operand_type

```

La manera en que se utiliza la función como mencione anteriormente, debido al bottom up, dentro de los body y si llegamos al final de las expresiones iremos tomando los valores y sus tipos y conforme encontremos operaciones iremos utilizando nuestro cubo semántico y nuestro stack de operandos para ir viendo si nuestro resultado va cumpliendo con los tipos que deben de ser.

```
def p_termino(p):
    "termino : factor mas_terminos"
    if p[2] is None:
        p[0] = p[1]
    else:
        left_termino = p[1]
        operator, right_termino = p[2]

        # Pop the last types
        right_operand_type = operand_stack.pop()
        left_operand_type = operand_stack.pop()

        # Check expected type
        result_type = get_expected_type(left_operand_type, right_operand_type, operator)

        # Push the new result type to the stack
        operand_stack.append(result_type)

    p[0] = [operator, left_termino, right_termino]
```

Finalmente, así se ve una prueba con el programa funcional y regresa el directorio de funciones con sus variables:

<pre>basic_program_data = """ program BasicProgram;  var i, j: int;  void max(i: int, j: int) [     {         if (i &gt; j) {             print(i);         } else {             print(j);         };     } ];  main {     i = 5;     j = 10;     print("The max value between i=", i, " and j is=", j, " is: ");     max(i, j);      do {         i = i + 1;          if (i &lt; j) {             print("La i sigue siendo menor. i = ", i);         } else {             print("La j es ahora mayor. j = ", j);         };     } while ( i &lt; j ); } end """  </pre>	<pre>Function and Vars Directory: global: {'vars': {'j': 'int', 'i': 'int'}}  max: {'vars': {'j': 'int', 'i': 'int'}}</pre>
--	---

En caso de recibir variables no declaradas, malas asignaciones de tipos o operaciones no válidas el compilador ya se encarga de detectarlo:

```
basic_program_data = """
program BasicProgram;

var i, j: int;

void max(i: int, j: int) [
    {
        if (i > a) {
            print(i);
        } else {
            print(j);
        };
    };
]

main
{
    i = 5;
    j = 10;
    print("The max value between i=", i, " and j is=", j, " is: ");
    max(i, j);

    do {
        i = i + 1;

        if (i < j) {
            print("La i sigue siendo menor. i = ", i);
        } else {
            print("La j es ahora mayor. j = ", j);
        };
    } while ( i < j );
}
end
"""

```

```
raise ReferenceError(f"'{operand}' is not defined")
ReferenceError: 'a' is not defined
```

```
basic_program_data = """
program BasicProgram;

var i, j: int;

void max(i: int, j: int) [
    var b: bool;
    {
        if (i > b) {
            print(i);
        } else {
            print(j);
        };
    };
]

main
{
    i = 5;
    j = 10;
    print("The max value between i=", i, " and j is=", j, " is: ");
    max(i, j);

    do {
        i = i + 1;

        if (i < j) {
            print("La i sigue siendo menor. i = ", i);
        } else {
            print("La j es ahora mayor. j = ", j);
        };
    } while ( i < j );
}
end
"""

```

```
File "/Users/miguelpedraza/Documents/Tec/Octavo Semestre/Aplicacion
raise TypeError(f"'{operator}' is not supported between instances
TypeError: '>' is not supported between instances of 'int' and 'bool'
miguelpedraza@MacBook-Pro-de-Miguel-3 ply_python_compiler %
```

```

basic_program_data = """
program BasicProgram;

var i, j: int;

void max(i: int, j: int) [
    var b: bool;
    {
        b = 5;

        if (i > j) {
            print(i);
        } else {
            print(j);
        };
    }
];

main
{
    i = 5;
    j = 10;
    print("The max value between i=", i, " and j is=", j, " is: ");
    max(i, j);

    do {
        i = i + 1;

        if (i < j) {
            print("La i sigue siendo menor. i = ", i);
        } else {
            print("La j es ahora mayor. j = ", j);
        };
    } while ( i < j );
}
end
"""

```

```

raise TypeError(f"Result type must be '{exp_type}'")
TypeError: Result type must be 'bool', not 'int'
miguelpedraza@MacBook-Pro-de-Miguel-3:~/ply_python

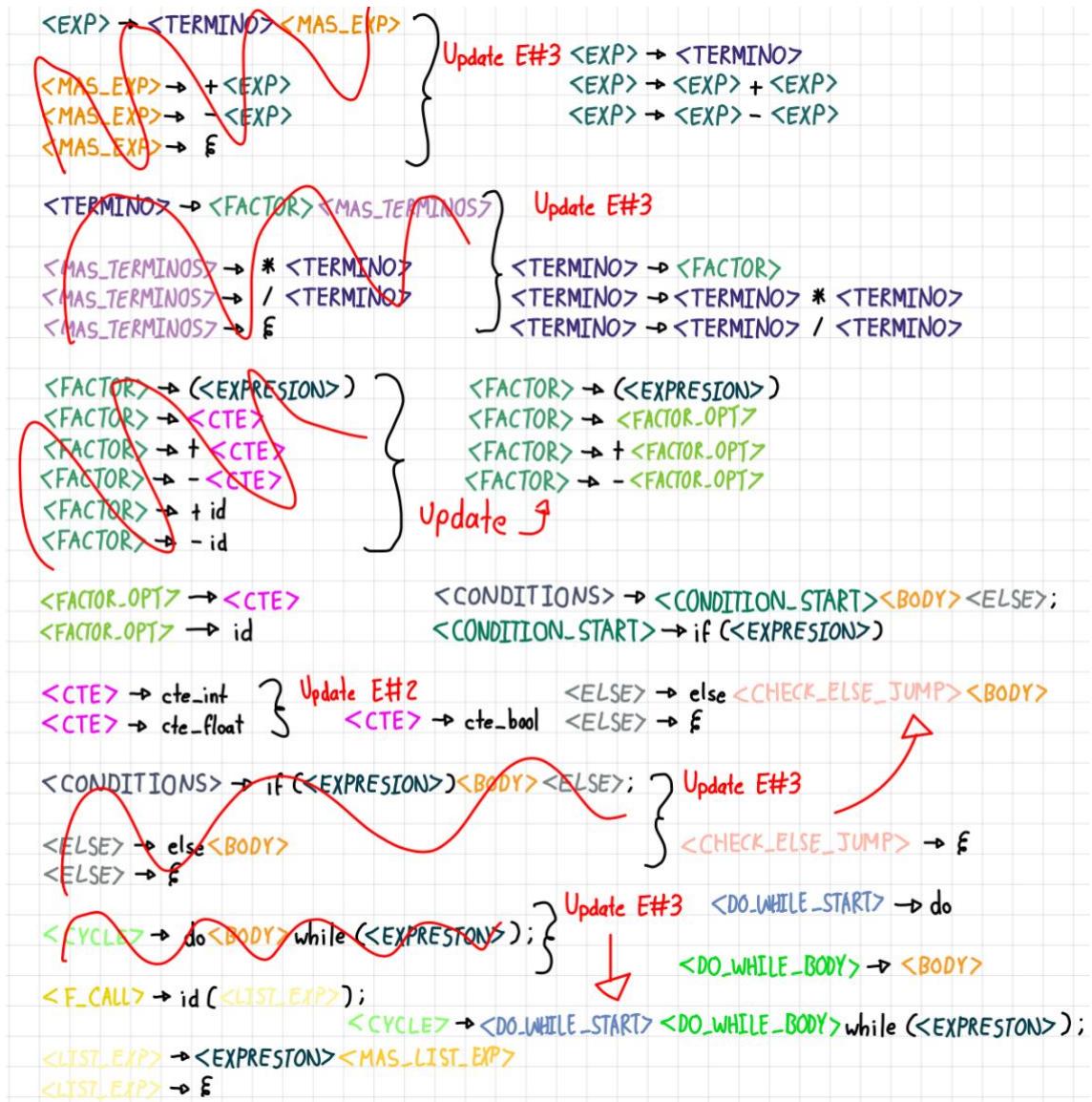
```

## Actualizaciones entrega 3

Ahora para nuestra nueva iteración del reto nos pusimos a realizar la representación intermedia de nuestras expresiones y estatutos de nuestro proyecto. Esto quiere decir que nos encargamos de generar la lista de cuaduplos que representan las instrucciones que lleva a cabo nuestro programa, es decir los pasos a realizar cuando tenemos asignaciones, condiciones, ciclos, funciones, prints y expresiones. El código intermedio que generamos será lo que al final debe de recibir la maquina virtual antes de generar los resultados.

Para esto primero que nada tuvimos que dar otro retoque grande a nuestras reglas gramaticales. En donde principalmente nos encargamos de agregar esos momentos en donde debemos de realizar saltos en nuestras instrucciones por ejemplo cuando tenemos condicionales o ciclos, y saber si en caso de que la expresión sea verdadera o false poder saltarse o no las siguientes instrucciones.

Un cambio que fue muy importante en nuestra gramática fue el hecho de que acortamos mucho las reglas para las expresiones, exp, termino y factor. A simple vista parece que agregamos ambigüedad en las nuevas reglas, pero esto se acomodó de esa forma ya que PLY cuenta con una manera de llevar a cabo tanto la precedencia como la asociatividad de las operaciones y gracias a eso logramos realizar cualquier expresión de manera muy sencilla.



## Expresiones

Al poner las reglas gramaticales de esa manera parecería que genera ambigüedad, pero gracias a la documentación de PLY me di cuenta de que al agregar un diccionario especificando la precedencia y asociatividad este se encargaría de evaluarlo sin necesidad de manejo extra de estructuras de datos. Con estas reglas como PLY usa un bottom up parsing automáticamente al ver que viene una expresión, pero no le sigue un operador relacional, esta se reduce a un solo exp lo que hace que entre a la regla de exp y de misma forma si no detecta que viene una suma automáticamente se reduce a término y sigue con las reglas de esa manera.

Ahora dentro del código solo tuvimos que preocuparnos de ir haciendo la revisión de tipos y agregando los cuádruples conforme entrabamos a las reglas en donde si existieran operaciones como se muestra a continuación:

```

def p_expresion(p):
    """expresion : exp
        | exp GREATERTHAN exp
        | exp LESSTHAN exp
        | exp NOTEQUAL exp"""

    if len(p) == 2:
        p[0] = p[1]
    else:
        # Pop the last types
        right_operand_type = operand_type_stack.pop()
        left_operand_type = operand_type_stack.pop()

        # Check expected type
        result_type = get_expected_type(left_operand_type, right_operand_type, p[2])

        # Push the new result type to the stack
        operand_type_stack.append(result_type)

        temp_var = globals.quadruples_queue.new_temp()
        globals.quadruples_queue.add_quadruple(p[2], p[1], p[3], temp_var)
        p[0] = temp_var

def p_exp(p):
    """exp : exp PLUS exp
        | exp MINUS exp"""

    # Pop the last types
    right_operand_type = operand_type_stack.pop()
    left_operand_type = operand_type_stack.pop()

    # Check expected type
    result_type = get_expected_type(left_operand_type, right_operand_type, p[2])

    # Push the new result type to the stack
    operand_type_stack.append(result_type)

    temp_var = globals.quadruples_queue.new_temp()
    globals.quadruples_queue.add_quadruple(p[2], p[1], p[3], temp_var)
    p[0] = temp_var

def p_exp_factor(p):
    "exp : termino"

    p[0] = p[1]

def p_termino(p):
    """termino : termino TIMES termino
        | termino DIVIDE termino"""

    # Pop the last types
    right_operand_type = operand_type_stack.pop()
    left_operand_type = operand_type_stack.pop()

    # Check expected type
    result_type = get_expected_type(left_operand_type, right_operand_type, p[2])

    # Push the new result type to the stack
    operand_type_stack.append(result_type)

    temp_var = globals.quadruples_queue.new_temp()
    globals.quadruples_queue.add_quadruple(p[2], p[1], p[3], temp_var)
    p[0] = temp_var

def p_termino_factor(p):
    "termino : factor"
    p[0] = p[1]

```

## Asignaciones:

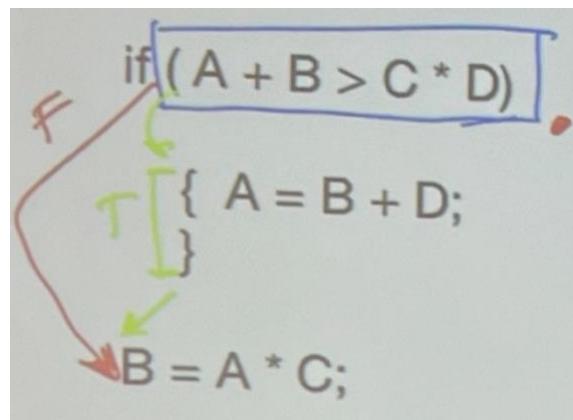
Para las asignaciones gracias a todo el trabajo hecho por PLY dentro de las reglas de expresiones solamente al entrar a la asignación se valida todo lo de tipos y que existan variables desde la entrega pasada y solamente necesitamos agregar la asignación a nuestra lista de cuádruplos con el valor que se fue asignando a lo largo de las reglas de expresiones.

```
def p_assign(p):
    "assign : ID ASSIGN expresion ENDINSTRUC"
    # Get the var name assigned value expresion and type of that value
    variable_name = p[1]
    assigned_value = p[3]
    assigned_type = operand_type_stack.pop()

    # Check first if the variable name was actually declared if so check if the types are correct
    if variable_name not in globals.funcs_dir[globals.current_scope]["vars"]:
        raise ReferenceError(f"Assignment to undeclared variable '{variable_name}'")
    elif globals.funcs_dir[globals.current_scope]["vars"][variable_name] != assigned_type:
        expected_type = globals.funcs_dir[globals.current_scope]["vars"][variable_name]
        raise TypeError(f"Result type must be '{expected_type}', not '{assigned_type}'")
    else:
        globals.quadruples_queue.add_quadruple('=', assigned_value, None, variable_name)
        p[0] = ['assign', variable_name, assigned_value]
```

## Condicionales:

Para llevar a cabo los estatutos condicionales tuvimos que implementar un stack extra para guardar el índice de la lista de cuádruplos en donde tenemos que volver para agregar donde se realizan los saltos. Esto porque al seguir el estatuto condicional tenemos que saltarnos las reglas de verdadero en caso de que sea falso, como se ve en el diagrama de abajo.



Dentro del código podemos ver como al partir la regla de condición para separar el `contion_start` podemos después de realizar los cuádruplos necesarios para las expresiones agregar el `goto` en caso de que sea falso y dejamos el lugar vacío para volver más adelante que sepamos donde debe de llegar. Además de agregar el índice actual menos 1 de la lista de cuádruplos para recordar a que cuádruplo vemosos volver para agregar el salto.

Después de eso vemos si tenemos un else y en caso de que si también tenemos que agregar la instrucción goto para saltarnos la regla de igual forma. Para poder regresar a la regla de condición inicial para poder cambiar y agregar el índice de la instrucción que debemos de acceder.

```
def p_condition(p):
    "condition : condition_start body else_block ENDINSTRU"
    # When we finish get the end jump index
    end_index = jump_stack.pop()

    # Add the pending index to the quadruple
    globals.quadruples_queue.edit_quadruple(end_index, None, None, None, globals.quadruples_queue.quadruples_len() + 1)

    p[0] = ['if', p[1], p[2], p[3]]

def p_condition_start(p):
    "condition_start : IF LPAREN expresion RPAREN"
    # Check if the expresion is valid bool
    expression_type = operand_type_stack.pop()
    if expression_type != "bool":
        raise TypeError(f"Expression must be 'bool', not '{expression_type}'")

    # Add the goto f when we finish evaluating the expression and save the index to return later
    globals.quadruples_queue.add_quadruple("gotof", p[3], None, None)
    jump_stack.append(globals.quadruples_queue.quadruples_len() - 1)

    p[0] = [p[1], p[3]]

def p_else_block(p):
    """else_block : ELSE check_else_jump body
    | empty"""
    if len(p) == 4:
        p[0] = [p[1], p[3]]

def p_check_else_jump(p):
    "check_else_jump : empty"
    # Add the goto f when we finish evaluating the expression
    globals.quadruples_queue.add_quadruple("goto", None, None, None)

    # Get the index of goto we had pending
    pending_goto_index = jump_stack.pop()

    # Save the new index to return later
    jump_stack.append(globals.quadruples_queue.quadruples_len() - 1)

    # Update the pending quadruple
    globals.quadruples_queue.edit_quadruple(pending_goto_index, None, None, None, globals.quadruples_queue.quadruples_len() + 1)
```

## Ciclos:

Para los estatutos de ciclos es un poco más sencillo ya que solamente debemos de preocuparnos de hacer el gotot para regresar al inicio del do. Como se ve en el siguiente código tenemos que volver a partir la regla que teníamos para agregar un inicio en el do, aquí podemos agregar el índice actual de la lista de cuádruplos al stack de saltos para saber a dónde volver. Y una vez evaluemos la expresión podemos sacar el índice y agregar el gotot hacia ese índice más uno para poder regresar en caso de que sea verdadero.

```

def p_cycle(p):
    "cycle : do_while_start do_while_body WHILE LPAREN expresion RPAREN ENDINSTRU"
    # Check if the expresion is valid bool
    expression_type = operand_type_stack.pop()
    if expression_type != "bool":
        raise TypeError(f"Expression must be 'bool', not '{expression_type}'")

    # Get the index of the start of do while
    start_do_while_index = jump_stack.pop()

    # Add the goto t to go back in case we need to continue looping
    globals.quadruples_queue.add_quadruple("gotot", p[5], None, start_do_while_index + 1)

    p[0] = ('do_while', p[2], p[4])

def p_do_while_start(p):
    "do_while_start : DO"
    # Before doing the body save the index to return later when we arrive to the while
    jump_stack.append(globals.quadruples_queue.quadruples_len())

    p[0] = p[1]

def p_do_while_body(p):
    "do_while_body : body"
    p[0] = p[1]

```

## Print:

Para los estatutos de print realizamos solamente en la regla principal del print la revisión de las expresiones de print que vienen. Primero se realizan los procesos de las expresiones y al finalizar se pasa la lista de los print options y podemos sencillamente agregar el cuádruplo con las expresiones correspondientes. Con esto podemos llevar a cabo el proceso de manera sencilla y eficiente.

```

def p_print(p):
    "print : PRINT LPAREN print_opt RPAREN ENDINSTRU"
    for expr in p[3]:
        if isinstance(expr, tuple):
            expr_value = expr[0]
            globals.quadruples_queue.add_quadruple('print', None, None, expr_value)
        else:
            globals.quadruples_queue.add_quadruple('print', None, None, expr)

    p[0] = ('print', p[3])

def p_print_opt(p):
    """print_opt : expresion more_opt
    | CTESTRING more_opt"""

    p[0] = [p[1]] + p[2]

def p_more_opt(p):
    """more_opt : COMMA print_opt
    | empty"""
    if len(p) == 3:
        p[0] = p[2]
    else:
        p[0] = []

```

## Resultados:

Finalmente tenemos este resultado hasta este momento:

```
program QuadTest;

var a, b, c, d, e, f: int;

main
{
  do {
    if (a + b < c) {
      a = b + c;
      do {
        a = a - 1;
      } while (a > b + c);
    } else {
      do {
        a = b + c * d;
        b = a - d;
      } while (b > c + d);
    };
  } while (a > b * c);
  a = b * c;
  c = 0;
}
end
```

```
----- Function and vars directory -----
global:
{'vars': {'f': 'int', 'e': 'int', 'd': 'int', 'c': 'int', 'b': 'int', 'a': 'int'}}
```

```
----- Quadruplets queue -----
1: (+, a, b, t1)
2: (<, t1, c, t2)
3: (gotof, t2, None, 12)
4: (+, b, c, t3)
5: (=, t3, None, a)
6: (-, a, 1, t4)
7: (=, t4, None, a)
8: (+, b, c, t5)
9: (>, a, t5, t6)
10: (gotot, t6, None, 6)
11: (goto, None, None, 20)
12: (*, c, d, t7)
13: (+, b, t7, t8)
14: (=, t8, None, a)
15: (-, a, d, t9)
16: (=, t9, None, b)
17: (+, c, d, t10)
18: (>, b, t10, t11)
19: (gotot, t11, None, 12)
20: (*, b, c, t12)
21: (>, a, t12, t13)
22: (gotot, t13, None, 1)
23: (*, b, c, t14)
24: (=, t14, None, a)
25: (=, 0, None, c)
```

## Actualizaciones entrega 4

Finalmente hemos llegado a finalizar nuestro compilador, considerando solamente ejecución del código dentro de main sin funciones adicionales de momento. Para ello finalmente nos adentramos a la máquina virtual que se encarga de representar a una computadora recreando a grandes rasgos la arquitectura clásica de Von Neuman en donde tenemos nuestra unidad de procesamiento y nuestra memoria. Esto lo hacemos ya que estamos replicando la solución de Java llamada JVM que es una plataforma de software que ejecuta programas Java al transformar el código bytecode, generado por el compilador de Java, en instrucciones que el sistema operativo del equipo puede entender. Esto es porque cuando un programa Java se compila no se convierte directamente en código máquina específico para un procesador, sino en bytecode, que es independiente de la plataforma. Es por eso por lo que nosotros replicamos de cierta forma un JVM para poder ejecutar nuestro resultado de compilador (código intermedio representado por lista de cuádruplos) y finalmente ejecutar las instrucciones del programa.

### Dentro del compilador (parser):

Para lograr hacer nuestra máquina virtual primero podíamos apoyarnos de nuestro mismo compilador para que los cuádruplos generados fueran representados como “direcciones virtuales” simulando las direcciones de memoria en donde estas serían asignadas. Primero que nada, definimos dentro de nuestro compilador que rango de direcciones virtuales tendría cada variable global entera, variable global flotante, variable temporal entera, etc. Por mi parte comencé eligiendo los siguientes rangos:

### Particiones de memoria

Globales enteras - 1000	Temporales enteras - 8000
Globales flotante - 2000	Temporales flotante - 9000
Globales booleana - 3000	Temporales booleana - 10000
Constante entera - 4000	
Constante flotante - 5000	
Constante string - 6000	
Constante bool - 7000	

### Operators / Operations

1: +	6: >	11: <i>gotof</i>
2: -	7: <	12: <i>print</i>
3: *	8: !=	
4: /	9: <i>goto</i>	
5: =	10: <i>gotot</i>	

Dentro de nuestro código creamos una clase para manejar la asignación de dichas direcciones virtuales dependiente de los rangos designados. Y dentro del código necesitamos crear funciones ayuda para manejar y asignar o recuperar la información necesaria. Todo fue llevándose a cabo de la misma manera que guardábamos los cuádruplos antes, solamente ahora también utilizando nuestra nueva clase y funciones de apoyo como se ve en las siguientes imágenes:

```

def allocate_temp(self, temp, var_type):
    if var_type == 'int':
        # Set the var considering the base
        self.temp_int_list.append(temp)
    elif var_type == 'float':
        # Set the var considering the base
        self.temp_float_list.append(temp)
    elif var_type == 'bool':
        # Set the var considering the base
        self.temp_bool_list.append(temp)
    else:
        raise TypeError(f"Unknown variable type: {var_type}")

def get_operator_memory(self, operator):
    try:
        index = self.operators_operations_list.index(operator)
        return index + 1
    except ValueError:
        print(f"{operator} is not in the list")

def get_var_int(self, operand):
    try:
        index = self.vars_int_list.index(operand)
        return index + self.vars_int_base
    except ValueError:
        print(f"{operand} is not in the list")

def get_var_float(self, operand):
    try:
        index = self.vars_float_list.index(operand)
        return index + self.vars_float_base
    except ValueError:
        print(f"{operand} is not in the list")

```

```

class VirtualMemoryCompiler:
    def __init__(self):
        # Base addresses for different types and scopes
        self.vars_int_base = 1000
        self.vars_float_base = 2000
        self.vars_bool_base = 3000
        self.const_int_base = 4000
        self.const_float_base = 5000
        self.const_string_base = 6000
        self.const_bool_base = 7000
        self.temp_int_base = 8000
        self.temp_float_base = 9000
        self.temp_bool_base = 10000

        # Operators/operations list
        self.operators_operations_list = ["+", "-", "*", "/", "=", ">", "<", "!=",
                                         "goto", "gotof", "print"]

        # Lists to map variables
        self.vars_int_list = []
        self.vars_float_list = []
        self.vars_bool_list = []

        self.const_int_list = []
        self.const_float_list = []
        self.const_string_list = []
        self.const_bool_list = []

        self.temp_int_list = []
        self.temp_float_list = []
        self.temp_bool_list = []

```

Como mencionamos anteriormente dentro del parser solo necesitábamos ir agregando en otra lista los cuádruplos con memorias virtuales a la vez que creábamos los que ya teníamos, estos podrían dejar de crearse, pero los dejé para fines de debug y ver que todo esté en orden.

```
def p_expresion(p):
    """expresion : exp
                  | exp GREATERTHAN exp
                  | exp LESSTHAN exp
                  | exp NOTEQUAL exp"""

    if len(p) == 2:
        p[0] = p[1]
    else:
        # Pop the last types
        right_operand_type = operand_type_stack.pop()
        left_operand_type = operand_type_stack.pop()

        # Check expected type
        result_type = get_expected_type(left_operand_type, right_operand_type, p[2])

        # Push the new result type to the stack
        operand_type_stack.append(result_type)

        # Generate a new temp var
        temp_var = globals.quadruples_queue.new_temp()

        # Add the temp var to assign virtual memory num
        globals.global_memory.allocate_temp(temp_var, result_type)

        # Get memory address equivalents
        operator_memory, operand1_memory, operand2_memory, result_memory = translate_to_memory(p[2], p[1], left_operand_type)

        # Add to quadruples list
        globals.quadruples_queue.add_quadruple(p[2], p[1], p[3], temp_var)
        globals.quadruples_queue.add_memory_quadruple(operator_memory, operand1_memory, operand2_memory, result_memory)

    p[0] = temp_var
```

```
def p_assign(p):
    "assign : ID ASSIGN expresion ENDINSTRU"
    # Get the var name assigned value expresion and type of that value
    variable_name = p[1]
    assigned_value = p[3]
    assigned_type = operand_type_stack.pop()

    # Check first if the variable name was actually declared if so check if the types are correct
    if variable_name not in globals.funcs_dir[globals.current_scope]["vars"]:
        raise ReferenceError(f"Assignment to undeclared variable '{variable_name}'")
    elif globals.funcs_dir[globals.current_scope]["vars"][variable_name] != assigned_type:
        expected_type = globals.funcs_dir[globals.current_scope]["vars"][variable_name]
        raise TypeError(f"Result type must be '{expected_type}', not '{assigned_type}'")
    else:
        # Translate to memory
        operator_memory = translate_operator_to_memory("=")
        operand1_memory = translate_operand_to_memory(assigned_value, assigned_type)
        result_memory = translate_result_to_memory(variable_name, assigned_type)

        # Add quadruple
        globals.quadruples_queue.add_quadruple('=', assigned_value, None, variable_name)
        globals.quadruples_queue.add_memory_quadruple(operator_memory, operand1_memory, -1, result_memory)

    p[0] = ['assign', variable_name, assigned_value]
```

```

def translate_operator_to_memory(operator):
    translated_operator = globals.global_memory.get_operator_memory(operator)
    return translated_operator

def translate_operand_to_memory(operand, operand_type):
    translated_operand = -1

    if operand is not None:
        if operand in globals.funcs_dir[globals.current_scope]["vars"]:
            if operand_type == "int":
                translated_operand = globals.global_memory.get_var_int(operand)
            elif operand_type == "float":
                translated_operand = globals.global_memory.get_var_float(operand)
            elif operand_type == "bool":
                translated_operand = globals.global_memory.get_var_bool(operand)
            elif isinstance(operand, str) and operand[0] == "t":
                if operand_type == "int":
                    translated_operand = globals.global_memory.get_temp_int(operand)
                elif operand_type == "float":
                    translated_operand = globals.global_memory.get_temp_float(operand)
                elif operand_type == "bool":
                    translated_operand = globals.global_memory.get_temp_bool(operand)
                elif operand_type == "string":
                    translated_operand = globals.global_memory.get_const_string(operand)
            else:
                if isinstance(operand, bool):
                    translated_operand = globals.global_memory.get_const_bool(operand)
                elif isinstance(operand, int):
                    translated_operand = globals.global_memory.get_const_int(operand)
                elif isinstance(operand, float):
                    translated_operand = globals.global_memory.get_const_float(operand)
        else:
            if isinstance(operand, bool):
                translated_operand = globals.global_memory.get_const_bool(operand)
            elif isinstance(operand, int):
                translated_operand = globals.global_memory.get_const_int(operand)
            elif isinstance(operand, float):
                translated_operand = globals.global_memory.get_const_float(operand)

    return translated_operand

```

Esto debía de realizarse en cada parte que se tenga que generar o agregar cuádruplos. Y con eso podíamos generar las siguientes listas de cuádruplos:

1: (=, 5, None, i)	1: (5, 4000, -1, 1000)
2: (=, 10, None, j)	2: (5, 4001, -1, 1001)
3: (+, i, 1, t1)	3: (1, 1000, 4002, 8000)
4: (t1, None, i)	4: (5, 8000, -1, 1000)
5: (<, i, j, t2)	5: (7, 1000, 1001, 10000)
6: (gotof, t2, None, 11)	6: (11, 10000, -1, 11)
7: (print, None, None, "La i sigue siendo menor: i = ")	7: (12, -1, -1, 6000)
8: (print, None, None, i)	8: (12, -1, -1, 1000)
9: (print, None, None, "")	9: (12, -1, -1, 6001)
10: (goto, None, None, 16)	10: (9, -1, -1, 16)
11: (print, None, None, "La i es ahora igual que j")	11: (12, -1, -1, 6002)
12: (print, None, None, "i = ")	12: (12, -1, -1, 6003)
13: (print, None, None, i)	13: (12, -1, -1, 1000)
14: (print, None, None, "j = ")	14: (12, -1, -1, 6004)
15: (print, None, None, j)	15: (12, -1, -1, 1001)
16: (<, i, j, t3)	16: (7, 1000, 1001, 10001)
17: (gotot, t3, None, 3)	17: (10, 10001, -1, 3)

Gracias a esto pudimos generar nuestro archivo de texto (simulando un archivo obj que lee la máquina virtual) creamos este método en nuestra clase de cuádruplos donde lo más importante, considerando solamente que se ejecuta el código de main, es que se tiene que pasar todas las constantes, los rangos de las variables y muy importante el número de variables enteras, flotantes, booleanas ya sea globales o temporales. Esto porque dentro de la máquina virtual usaremos esa información para recrear una memoria en donde ir almacenando los cálculos reales, es por eso por lo que nos encargamos de simular memorias virtuales dentro

del compilador. A continuación, se muestra el método para generar dicho archivo y el archivo txt generado:

```
def generate_obj_file(self):
    with open("ovejota.txt", "w") as file:
        # Operations array
        file.write("Operators/Operations:\n")
        file.write(f'{globals.global_memory.operators_operations_list}\n')

        # Global vars count and base
        file.write("Global vars:\n")
        file.write(f"Global int vars = {len(globals.global_memory.vars_int_list)}\n")
        file.write(f"base = {globals.global_memory.vars_int_base}\n")

        file.write(f"Global float vars = {len(globals.global_memory.vars_float_list)}\n")
        file.write(f"base = {globals.global_memory.vars_float_base}\n")

        file.write(f"Global bool vars = {len(globals.global_memory.vars_bool_list)}\n")
        file.write(f"base = {globals.global_memory.vars_bool_base}\n")

        # Temp vars count and base
        file.write("Temp vars:\n")
        file.write(f"Temp int vars = {len(globals.global_memory.temp_int_list)}\n")
        file.write(f"base = {globals.global_memory.temp_int_base}\n")

        file.write(f"Temp float vars = {len(globals.global_memory.temp_float_list)}\n")
        file.write(f"base = {globals.global_memory.temp_float_base}\n")

        file.write(f"Temp bool vars = {len(globals.global_memory.temp_bool_list)}\n")
        file.write(f"base = {globals.global_memory.temp_bool_base}\n")

        # Constants
        file.write("Constant integers:\n")
        file.write(f"base = {globals.global_memory.const_int_base}\n")
        file.write(f'{globals.global_memory.const_int_list}\n')

        file.write("Constant floats:\n")
        file.write(f"base = {globals.global_memory.const_float_base}\n")
        file.write(f'{globals.global_memory.const_float_list}\n')

        file.write("Constant strings:\n")
        file.write(f"base = {globals.global_memory.const_string_base}\n")
        file.write(f'{globals.global_memory.const_string_list}\n')

    Operators/Operations:
    ['+', '-', '*', '/', '=', '>', '<', '!=', 'goto', 'gotot', 'gotof', 'print']
    Global vars:
    Global int vars = 2
    base = 1000
    Global float vars = 0
    base = 2000
    Global bool vars = 0
    base = 3000
    Temp vars:
    Temp int vars = 1
    base = 8000
    Temp float vars = 0
    base = 9000
    Temp bool vars = 2
    base = 10000
    Constant integers:
    base = 4000
    [5, 10, 1]
    Constant floats:
    base = 5000
    []
    Constant strings:
    base = 6000
    ['"La i sigue siendo menor: i = "', '""', '"La i es ahora igual que j"', '"i = "', '"j = "']
    Constant bools:
    base = 7000
    []
    Quadruples list:
    (5, 4000, -1, 1000)
    (5, 4001, -1, 1001)
    (1, 1000, 4002, 8000)
    (5, 8000, -1, 1000)
    (7, 1000, 1001, 10000)
    (11, 10000, -1, 11)
    (12, -1, -1, 6000)
    (12, -1, -1, 1000)
    (12, -1, -1, 6001)
    (9, -1, -1, 16)
    (12, -1, -1, 6002)
    (12, -1, -1, 1000)
    (12, -1, -1, 6003)
    (12, -1, -1, 5004)
```

## Máquina virtual:

Finalmente, para crear esta máquina virtual nos encargamos de crear dos clases, la clase de Máquina virtual, que es como nuestro CPU encargado de realizar las operaciones Y nuestra clase de Memoria que se encarga de manejar el almacenamiento de valores globales o temporales conforme se van realizando las operaciones.

En este caso la clase de memoria está pensada para poder ser instanciada y utilizada tanto para variables globales o temporales. La idea es que al instanciar la clase dentro de sus atributos tiene 3 listas la de enteros, flotantes y booleanos que manejamos en nuestro lenguaje. Al leer el archivo que viene del compilador podemos instanciar dichas listas solamente con el número de variables que tendrá cada lista. De esta manera logramos un manejo eficaz y sin gastar memoria de más. Con esta solución podemos tener una misma clase para instanciar la memoria global y memoria temporal (en un futuro también memoria local en caso de funciones).

```
class Memory:
    def __init__(self, int_space, float_space, bool_space, bases):
        # Bases for memory space
        self.int_base, self.float_base, self.bool_base = bases

        # List of vars memory spaces
        self.integers = [None] * int_space
        self.floats = [None] * float_space
        self.bools = [None] * bool_space
```

Todo inicia dentro de la clase de Máquina Virtual, en donde leemos el archivo y dentro de nuestra máquina virtual lo más importante es tener la tabla de constantes y el segmento de código (lista de cuádruplos):

```
def load_obj_file(self, filename):
    with open(filename, 'r') as f:
        lines = f.readlines()

    # Constants
    const_integers_base = 0
    const_integers = []
    const_floats_base = 0
    const_floats = []
    const_strings_base = 0
    const_strings = []
    const_bools_base = 0
    const_bools = []

    # Quadruples
    quadruples = []

    # Global and Temp Variables
    global_int_vars = 0
    global_float_vars = 0
    global_bool_vars = 0
    temp_int_vars = 0
    temp_float_vars = 0
    temp_bool_vars = 0
```

```

        if line.startswith("base ="):
            const_bools_base = int(line.split('=')[1].strip())
        else:
            const_bools = eval(line)
    else:
        if line.startswith("("):
            quadruples.append(list(map(int, eval(line)))))

    previous_line = line

    # Parse constant strings to remove the ""
    formatted_const_strings = [None] * len(const_strings)
    for index, string in enumerate(const_strings):
        new_string = string[1:-1] if len(string) > 1 else ""
        formatted_const_strings[index] = new_string

    # Delete the prev string constants list
    del const_strings
    gc.collect()

    # Finally create the const table
    const_table = {const_integers_base + i: v for i, v in enumerate(const_integers)}
    const_table.update({const_floats_base + i: v for i, v in enumerate(const_floats)})
    const_table.update({const_strings_base + i: v for i, v in enumerate(formatted_const_strings)})
    const_table.update({const_bools_base + i: v for i, v in enumerate(const_bools)})

    # Create all memory needed
    self.global_memory = Memory(global_int_vars, global_float_vars, global_bool_vars, [global_int_base, global_float_base, global_bool_base])
    self.temp_memory = Memory(temp_int_vars, temp_float_vars, temp_bool_vars, [temp_int_base, temp_float_base, temp_bool_base])

    return quadruples, const_table

```

Finalmente, con todo esto podemos empezar la ejecución del código con el método execute que la idea del algoritmo es bastante sencilla. Tenemos un apuntador de instrucciones y vamos a estar dentro de un ciclo en donde no parará hasta que lleguemos al final de la lista de cuádruplos o ahora conocido como segmento de código. Y para cada cuádruplo vamos a tener nuestros condicionales buscando el operador u operación que queremos hacer, al encontrarla dependiendo de que sea debemos primero encontrar los valores de los operandos ya sea dentro de la tabla de constantes o ya en memoria y finalmente realizamos la operación que se pide y guardamos el resultado real en la dirección de memoria necesaria y continuamos hasta que termine el segmento de código. En caso de tener condiciones o ciclos el programa se encarga simplemente de mover el apuntador de instrucciones a donde tenga que moverse y seguir con el programa.

```

def execute(self):
    while self.instruction_pointer < len(self.code_segment):
        current_quad = self.code_segment[self.instruction_pointer]
        self.handle_instruction(current_quad)
        self.instruction_pointer += 1

```

```

def handle_instruction(self, instruction):
    # Get all components of the quadruple instruction
    operator, left_operand, right_operand, result = instruction

    # Check what operation are we doing and store results in memory
    if self.operators_operations_list[operator - 1] == "+":
        # Get values of operands
        left_val = None
        right_val = None
        if left_operand in self.const_table:
            left_val = self.const_table[left_operand]
        else:
            left_val = self.get_value(left_operand)

        if right_operand in self.const_table:
            right_val = self.const_table[right_operand]
        else:
            right_val = self.get_value(right_operand)

        # Do the operation
        res = left_val + right_val

        # Set the value in corresponding memory
        self.set_value(result, res)

```

Algo muy importante son nuestros métodos para obtener y almacenar datos en la memoria. Gracias a los rangos de las direcciones virtuales en la lista de cuádruplos podemos identificar si son variables enteras, flotantes o booleanas, así como si son globales o temporales. A continuación, se muestra cómo se realizaron dichas funciones:

```
def get_value(self, operand):
    if operand in self.const_table:
        return self.const_table[operand]
    else:
        if operand >= self.global_memory.int_base and operand <= self.global_memory.bool_base + 999:
            if operand >= self.global_memory.int_base and operand < self.global_memory.float_base:
                return self.global_memory.integers[operand - self.global_memory.int_base]
            elif operand >= self.global_memory.float_base and operand < self.global_memory.bool_base:
                return self.global_memory.floats[operand - self.global_memory.float_base]
            else:
                return self.global_memory.bools[operand - self.global_memory.bool_base]
        else:
            if operand >= self.temp_memory.int_base and operand < self.temp_memory.float_base:
                return self.temp_memory.integers[operand - self.temp_memory.int_base]
            elif operand >= self.temp_memory.float_base and operand < self.temp_memory.bool_base:
                return self.temp_memory.floats[operand - self.temp_memory.float_base]
            elif operand >= self.temp_memory.bool_base and operand < self.temp_memory.bool_base + 999:
                return self.temp_memory.bools[operand - self.temp_memory.bool_base]
            else:
                raise MemoryError(f"Address {operand} not available!")
```

```
def set_value(self, result_address, value):
    # Use the ranges of the bases to determine weather we're having a global or temp var and store the value
    if result_address >= self.global_memory.int_base and result_address <= self.global_memory.bool_base + 999:
        if result_address >= self.global_memory.int_base and result_address < self.global_memory.float_base:
            self.global_memory.integers[result_address - self.global_memory.int_base] = value
        elif result_address >= self.global_memory.float_base and result_address < self.global_memory.bool_base:
            self.global_memory.floats[result_address - self.global_memory.float_base] = value
        else:
            self.global_memory.bools[result_address - self.global_memory.bool_base] = value
    else:
        if result_address >= self.temp_memory.int_base and result_address < self.temp_memory.float_base:
            self.temp_memory.integers[result_address - self.temp_memory.int_base] = value
        elif result_address >= self.temp_memory.float_base and result_address < self.temp_memory.bool_base:
            self.temp_memory.floats[result_address - self.temp_memory.float_base] = value
        elif result_address >= self.temp_memory.bool_base and result_address < self.temp_memory.bool_base + 999:
            self.temp_memory.bools[result_address - self.temp_memory.bool_base] = value
        else:
            raise MemoryError(f"Address {result_address} not available!")
```

## Pruebas de ejecución:

Finalmente, con todo este proceso de la máquina virtual y memoria podemos ejecutar el código de nuestro lenguaje de programación Little Duck. A continuación, se muestra un programa de ejemplo:

```
program BasicProgram;

var i, j: int;

main
{
    i = 5;
    j = 10;

    do {
        i = i + 1;

        if (i < j) {
            print("La i sigue siendo menor: i = ", i);
        } else {
            print("La i es ahora igual que j");
            print("i = ", i);
            print("j = ", j);
        }
    } while ( i < j );
}
end
```

```
La i sigue siendo menor: i =
6
La i sigue siendo menor: i =
7
La i sigue siendo menor: i =
8
La i sigue siendo menor: i =
9
La i es ahora igual que j
i =
10
j =
10
```