

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

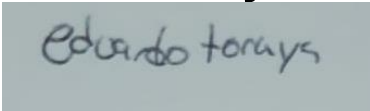


Diseño de Compiladores

**Profesores
Elda Quiroga González
Héctor Ceballos Cancino**

Documentación Final
Forever Alone

Eduardo Alejandro Toraya Solís A00819785

A rectangular box containing a handwritten signature in blue ink that reads "Eduardo toraya".

Eduardo toraya

Fecha de entrega: 2 de Junio del 2020.

Índice

- a) Descripción del proyecto
 - a. 1.- Propósito, Objetivos y alcance del proyecto. Página 3.
 - b. 2.- Análisis de Requerimientos y casos de Uso generales. Página 3.
 - c. 3.- Descripción de los principales Test Cases. Página 3.
 - d. 4.- Descripción del proceso de desarrollo del proyecto. Página 4.
- b) Descripción del lenguaje
 - a. 1.- Nombre del lenguaje. Página 8.
 - b. 2.- Descripción genérica de características del lenguaje. Página 8.
 - c. 3.- Listado de errores de compilación y ejecución. Página 8.
- c) Descripción del compilador
 - a. 1.- Equipo de cómputo, lenguaje y utilerías. Página 9.
 - b. 2.- Descripción del Análisis de Léxico. Página 9.
 - c. 3.- Descripción del Análisis de Sintaxis. Página 10.
 - d. 4.- Descripción de Generación de Código Intermedio y Análisis Semántico. Página 13.
 - e. 5.- Descripción detallada del proceso de Administración de Memoria usado en compilación. Página 17.
- d) Descripción de la Máquina Virtual
 - a. 1.- Equipo de Cómputo, lenguaje y utilerías especiales. Página 19.
 - b. 2.- Descripción detallada del proceso de administración de memoria. Página 19.
- e) Pruebas del funcionamiento del lenguaje.
 - a. Pruebas. Página 20.
- f) Listados perfectamente documentados del proyecto
 - a. Comentarios de Documentación y comentarios de Implementación. Página 23.

a) DESCRIPCIÓN DEL PROYECTO

a.1) Propósito, Objetivos y Alcance del Proyecto.

El Propósito de este proyecto es desarrollar un compilador en Python utilizando como herramienta lexx and yacc para parsear de tal forma que cumpla con los requerimientos funcionales y de diseño propuestos por el documento de ForeverAlone.

a.2) Análisis de Requerimientos y Casos de Uso generales.

Requerimientos:

- Debe manejar tipos de dato int, float y carácter.
- Debe contar con una función principal y funciones adicionales opcionales.
- Debe poder declarar variables, obtener variables del usuario y mostrar variables al usuario.
- Debe manejar arreglos de una dimensión en sus operaciones.
- Debe contar con operaciones aritméticas lineales.
- Debe contar con llamadas a funciones.
- Debe contar con manejo de estatutos cíclicos y condicionales.
- Maneja expresiones como +, -, *, /, &, |, <, <=, ==, <>, >=, y > para floats e integers y para caracteres sólo las lógicas.
- Las funciones podrán ser de tipo void (no regresan valor) o de tipos carácter, float o integer que regresarán valor de su mismo tipo.
- Permite recursividad

Casos de uso generales:

- Definir/Ejecutar programa.
- Declarar variables individuales o de arreglo.
- Llamar variables individuales o de arreglo
- Iniciar/terminar estatutos no secuenciales (while, for, if).
- Llamar a funciones para su ejecución.
- Asignar valores.
- Mostar/obtener valores para usuario
- Escribir comentarios.
- Escribir strings.
- Iniciar o detener recursión.

a.3) Descripción de los principales test cases.

T.C.1: En el testCase del file MVTestOperaciones, se prueba la consistencia de las operaciones de suma, resta, multiplicación, operadores de comparación, asignación, lectura de valor de usuario y escritura.

T.C.2: MVTestCiclos, se prueba la consistencia de operaciones no lineales. Se maneja el while, el if/else, el for y estos mismos anidados. Mientras se realizan operaciones aritméticas, escritura y asignación.

T.C.3: MVTestFunciones, se prueba la consistencia y el correcto funcionamiento de la memoria en funciones void, con retorno y recursivas, así

como llamadas recursivas y con llamada interna para confirmar el no colisionamiento de espacio de dirección. Este test case incluye la producción de un factorial y de la serie de Fibonacci hasta un valor dado por el usuario.

T.C.4: MVTestArrays, se prueba la consistencia de asignación de memoria de los arreglos, en el cual se prueba la asignación, lectura y operaciones aritméticas correctas de los arreglos. Se revisa el funcionamiento de arreglos anidados ejemplo: A[A[3]] y se prueba un sorteo de un arreglo definido en el mismo código combinando con características de ciclos.

a.4) Descripción del PROCESO.

El proceso para llevar a cabo este compilador fue el desarrollo por partes de acuerdo a los avances del mismo. Se comenzó con escoger una propuesta de proyecto de las ofrecidas por los profesores, en este caso se desarrolló el lexer, de ahí el código para operaciones lineales, seguido de operaciones no lineales, seguido de funciones y finalmente se desarrollaron los arreglos en el parser. Posteriormente, se desarrolló una máquina virtual que pudiese leer los cuádruplos generados en un texto objeto por el parser y los interpreta en Python.

Reflexión principal, Se tiene un mejor entendimiento del proceso interno que se puede llevar para poder traducir el código de un programador en instrucciones realizables por una computadora.

Reporte Entrega 13 abril
Entrega del analizador léxico y sintáctico

Issues a reparar para siguiente entrega:
I1.1 No identifica errores en específico.
I1.2 Faltaron pruebas extensivas.
I1.3 Se duda de que la gramática sea sólida en un 100%

Reporte Entrega 20 abril
Entrega del analizador léxico y sintáctico de cambio de proyecto a ForeverAlone

Diagramas de sintaxis: https://www.lucidchart.com/documents/edit/5059454c-35d8-4bb4-8807-630eae053432/0_0

Issues a reparar para siguiente entrega:
I1.1 No identifica errores en específico.
I1.2 Faltaron pruebas extensivas.
I1.3 Atraso total entrega 2, directorio de procedimientos y tabla de variables.

Reporte Entrega 27
Entrega del cubo semántico y otros.

Issues a reparar para siguiente entrega:
I1.1 No identifica errores en específico.
I1.2 Atraso parcial entrega 3 -> falto generar cuádruplos con estatutos lineales.

Progreso:
1.- tokens, diagramas sintaxis, cubo semántico, dir funciones, tabla de variables.

Reporte entrega 4 mayo
Entrega de atraso en estatutos lineales y entrega de estatutos no lineales.
I1.- Posible que se requiera agregar un dato más al cuádruplo con número de cuádruplo.
I2.- Falta manejar memoria.

Reporte entrega 17 mayo:
Faltan arreglos y máquina virtual.
Se me olvido subirlo.

Reporte 25 mayo:
Avance hasta funciones de la máquina virtual. incluye desarrollo funciones
Falta: desarrollar arreglos en máquina virtual, probar arreglos en parser,
documentación.

Reporte 31 mayo.
Compilador terminado. Con pruebas.

<https://github.com/EduardoToraya/Compiler-in-python>

Commits on May 28, 2020

1. [update before documentation](#)



EduardoToraya committed 1 minute ago

[f09b673](#)

Commits on May 27, 2020

1. [Funcionalidad completada.](#)



EduardoToraya committed 23 hours ago

[1c87540](#)

Commits on May 25, 2020

1. [Finished parser and functions working ...](#)



EduardoToraya committed 3 days ago

[0bd6bf2](#)

Commits on May 19, 2020

1. [Functions added and bug repair. ...](#)



EduardoToraya committed 9 days ago

[7b6a0d6](#)

Commits on May 17, 2020

1. [Update a funciones](#)



EduardoToraya committed 11 days ago

[7a7f7d3](#)

2. [changes to function](#)



EduardoToraya committed 11 days ago

[8337952](#)

Commits on May 13, 2020

1. [Update on memory management](#)



EduardoToraya committed 15 days ago

[b675be2](#)

Commits on May 3, 2020

1. [Update estatutos lineales y no lineales.](#)



EduardoToraya committed 25 days ago

[7392353](#)

Commits on Apr 27, 2020

1. [3rd update on compiler ...](#)



EduardoToraya committed on 27 Apr

[5a32f97](#)

Commits on Apr 20, 2020

1. [FirstUpload](#)



EduardoToraya committed on 20 Apr

[c069a18](#)

2. [Initial commit](#)



EduardoToraya committed on 20 Apr

[1c0ad64](#)

Firma:

b) DESCRIPCIÓN DEL LENGUAJE:

b.1) Nombre del Lenguaje: ForeverAlone.

b.2) Descripción genérica de las principales características del lenguaje.

Este lenguaje utiliza palabras en español para sus palabras reservadas de operaciones. Comienza definiendo el nombre del programa seguido de su declaración de variables globales, posteriormente es opcional crear las funciones las cuales deberán ser declaradas con la palabra clave de esta, seguida de su tipo y de los parámetros que tendrá la misma, similar a la instanciación de funciones en c++. Posteriormente se cuenta con un bloque Principal() de código y debe colocarse hasta el final del código para correcta ejecución, con las funciones auxiliares arriba del mismo.

b.3) Listado de errores que pueden ocurrir

Compilación:

- Variable ya instanciada
- Numero de variables en su límite. Se declararon mas de 2500 variables por tipo.
- Numero de constantes excedida.
- El índice de un arreglo debe ser una expresión íntegra.
- Variable por acceder requiere de dimensiones.
- Variable no encontrada en el ambiente.
- Función ya declarada.
- No se pueden declarar nombres de funciones igual que las variables globales.
- Parámetro ya declarado.
- Errores de tipo para asignación.
- Errores de coincidencia de parámetros.
- Número de parámetros de llamada a una función.
- La función que se está llamando no existe.
- Errores de tipo para operaciones OR, AND, + , -, Multiplicación, división y elementos comparativos.
- Errores de inconsistencia de paréntesis.
- Errores de validación de estatutos no lineales. For while if else.
- Error de tipo de retorno de una función.
- Errores sintácticos.
- Archivo no encontrado.

Ejecución:

- Error en acceso de arreglos fuera de dimensión.
- Error en archivo no existente o consistente.
- Error de dirección en ambiente.

c). DESCRIPCIÓN DEL COMPILADOR:

c.1) Equipo de cómputo, lenguaje y utilerías especiales usadas en desarrollo del proyecto.

Se utilizó Windows como SO para el desarrollo, pero se puede correr este compilador en cualquier equipo que pueda utilizar versiones de Python superiores a la 3.8. Se utilizo Python para el desarrollo con librerías como Python Lexx and Yacc y la terminal para su ejecución.

c.2) Descripción del análisis de Léxico.

```
##Declaration of tokens
tokens = [
    'ID',          ## variable
    'LPAREN',      ## (
    'RPAREN',      ## )
    'LBRACKET',    ## {
    'RBRACKET',    ## }
    'LSQUARE',     ## [
    'RSQUARE',     ## ]
    'COMMA',       ## ,
    'SEMICOLON',   ## ;
    'COMMENT',     ## #
    'CTE_I',       ## 123
    'CTE_F',       ## 123.123
    'CTE_C',       ## a
    'CTE_S',       ## palabra
    'EQUAL',       ## =
    'LESSTHAN',    ## <
    'LESSEQUAL',   ## <=
    'GREATERTHAN', ## >
    'GREATEREQUAL', ## >=
    'NOEQUAL',     ## <>
    'PLUS',        ## +
    'MINUS',       ## -
    'MULT',        ## *
    'DIV',         ## /
    'AND',         ## &
    'OR',          ## |
    'SAME',        ## ==
]

##Reserved words
reserverd = {
    'programa' : 'PROGRAMA',
    'var' : 'VAR',
    'int' : 'INT',
    'float' : 'FLOAT',
    'char' : 'CHAR',
    'void' : 'VOID',
    'principal' : 'PRINCIPAL',
    'funcion' : 'FUNCION',
    'regresa' : 'REGRESA',
    'lee' : 'LEE',
    'escribe' : 'ESCRIBE',
    'si' : 'SI',
    'entonces' : 'ENTONCES',
    'sino' : 'SINO',
    'mientras' : 'MIENTRAS',
    'haz' : 'HAZ',
    'desde' : 'DESDE',
    'hasta' : 'HASTA',
    'hacer' : 'HACER',
}

##tokens symbols
t_ignore = ' \t'

# Symbols
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_LSQUARE = r'\['
t_RSQUARE = r'\]'
t_COMMA = r','
t_SEMICOLON = r';'
t_ignore_COMMENT = r'\#.*'

# Comparison
t_EQUAL = r'='
t_LESSTHAN = r'<'
t_LESSEQUAL = r'<='
t_GREATERTHAN = r'>'
t_GREATEREQUAL = r'>='
t_NOEQUAL = r'<>'
t_SAME = r'=='

# Logic Operators
t_AND = r'&'
t_OR = r'|'

# Arithmetic Operators
t_PLUS = r'+'
t_MINUS = r'-'
t_MULT = r'*'
t_DIV = r'/'

# Constants
t_CTE_I = r'[0-9]+'
t_CTE_F = r'[0-9]+\.[0-9]+'
t_CTE_C = r'([^\s]*)'
t_CTE_S = r'([^\s])([^\s])?'"

tokens = tokens +
list(reserverd.values())

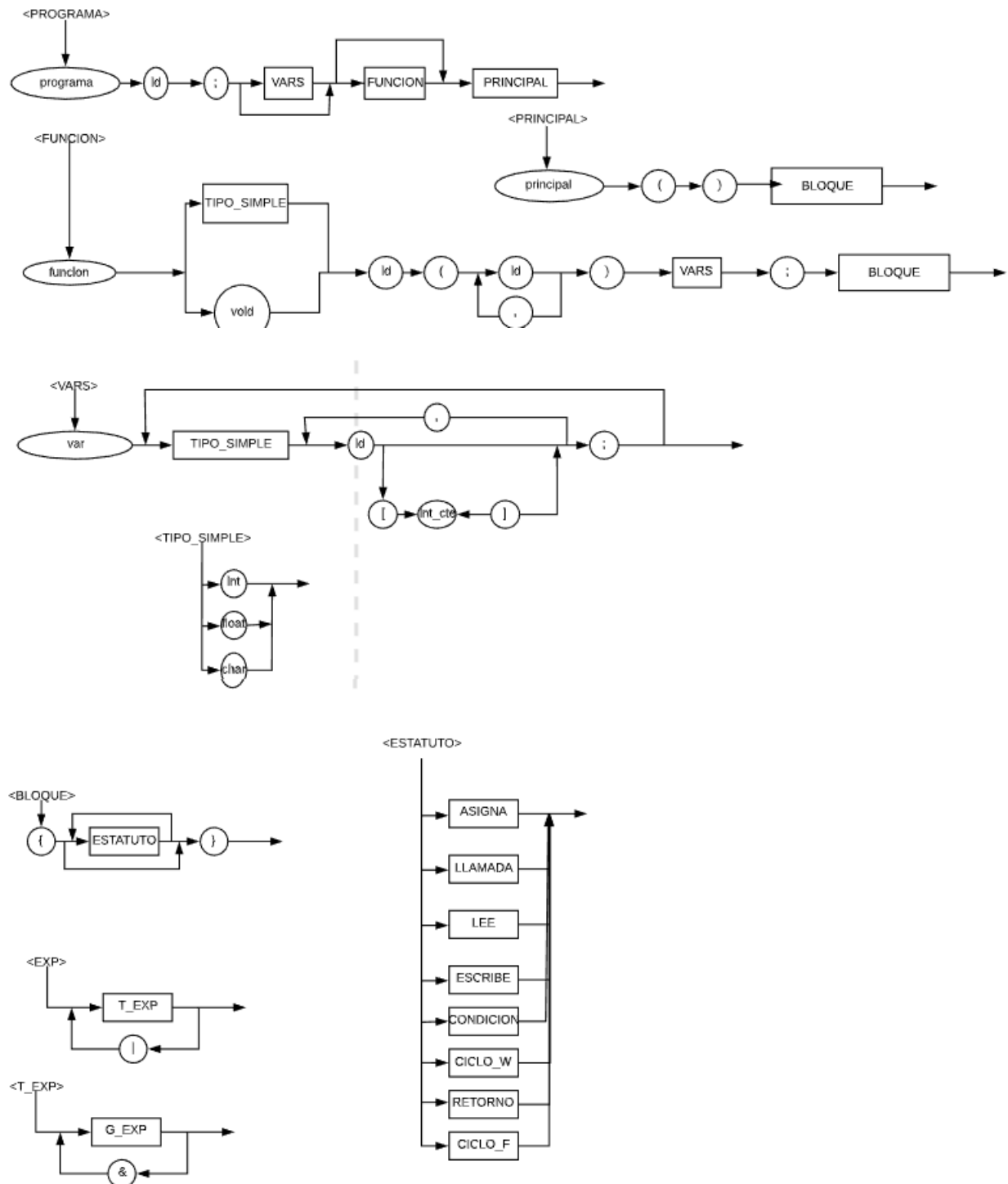
##declaration for letters with
words ex hola93
def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type =
reserverd.get(t.value,'ID')
    return t

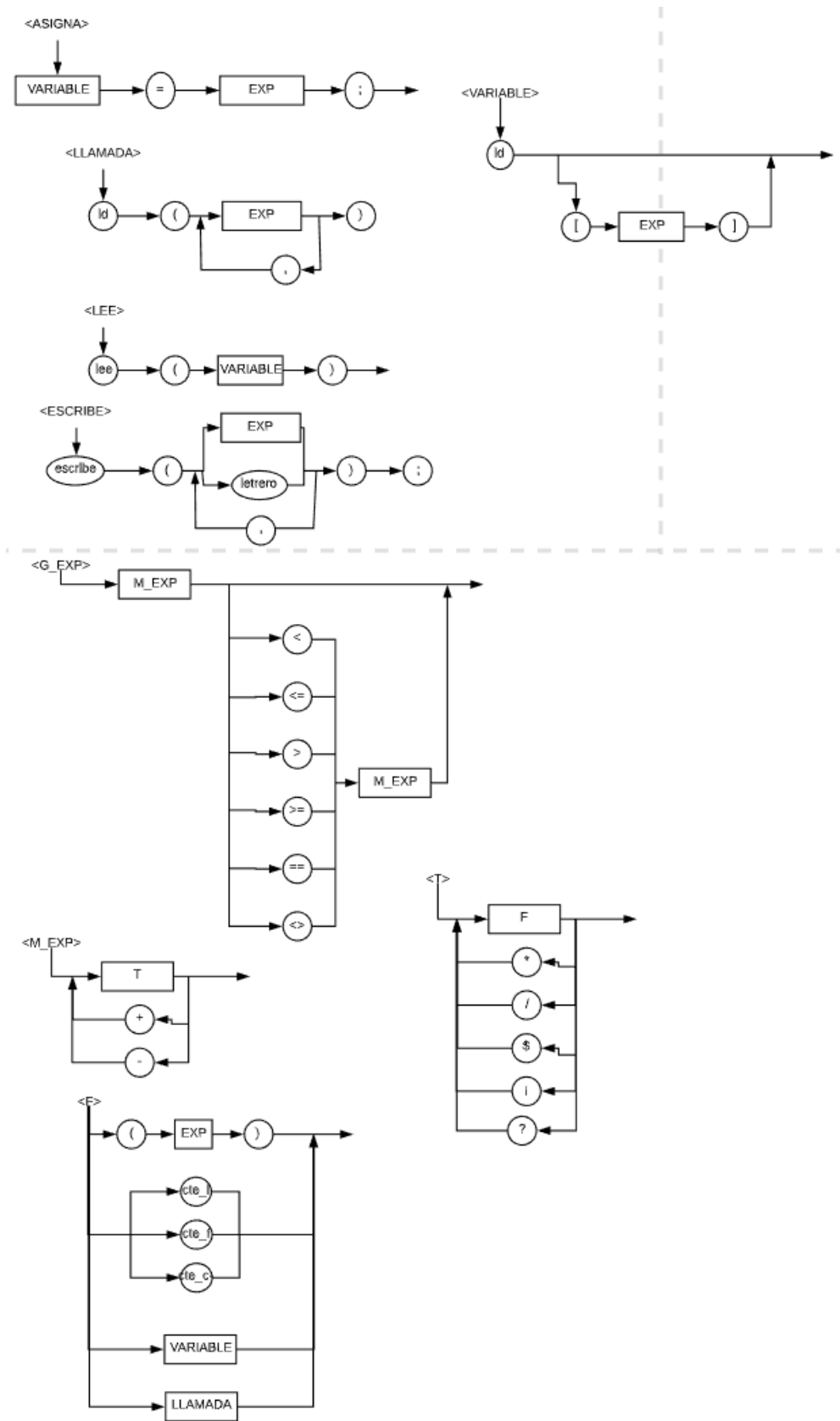
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

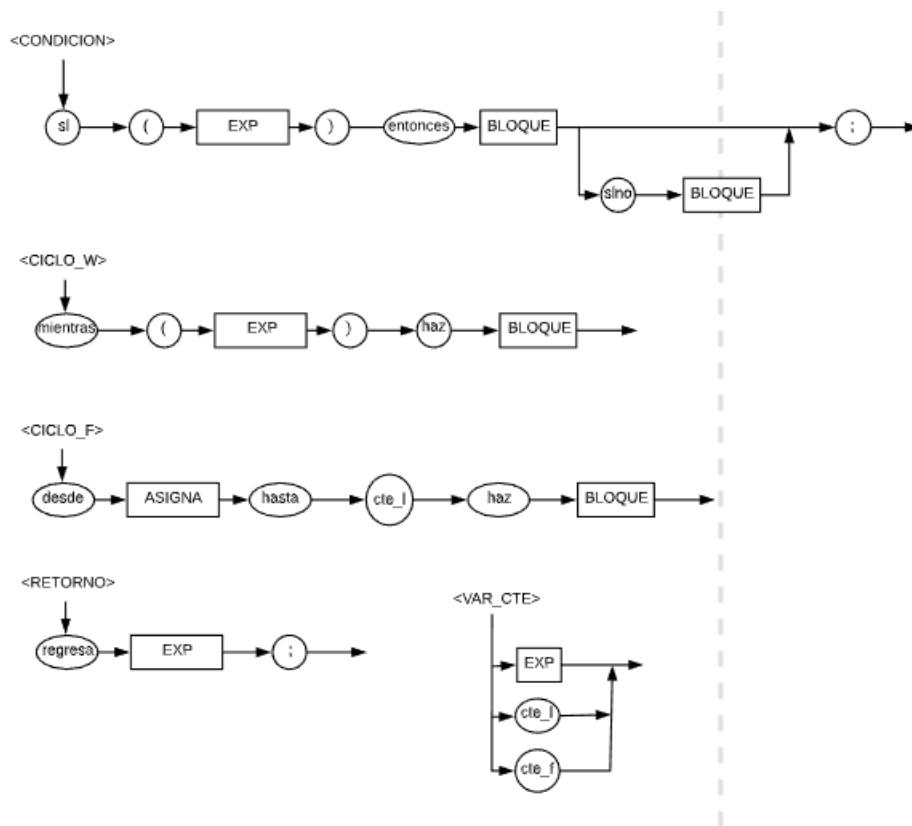
##Lexer error function
def t_error(t):
    global success
    success = False
    print ("Caracter no valido '%s'"
    % t.value[0])
    print("en la línea ",
    t.lexer.lineno)
    t.lexer.skip(1)
    sys.exit()
```

c.3) Descripción del análisis de sintaxis

Diagramas realizados en lucidchart.







c.4) Descripción de Generación de Código Intermedio y Análisis Semántico

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

Se definen los rangos de direcciones por tipo de dato.

```
DIR_BASE_INT = 0
DIR_BASE_FLOAT = 2500
DIR_BASE_CHAR = 5000
DIR_BASE_BOOL = 7500
```

Se define el largo por seccion y los rangos de direcciones base.

```
DIR_LENGTH = 2500
```

```
DIR_BASE_GLOBAL = 0
DIR_BASE_LOCAL = 10000
DIR_BASE_CTE = 20000
DIR_BASE_POINTER = 30000
```

```
dir_func = {
    'global': {
        'vars': {},
        'params': {},
        'next_int': DIR_BASE_GLOBAL + DIR_BASE_INT,
        'next_float': DIR_BASE_GLOBAL +
DIR_BASE_FLOAT,
        'next_char': DIR_BASE_GLOBAL +
DIR_BASE_CHAR,
        'next_bool': DIR_BASE_GLOBAL +
DIR_BASE_BOOL
    },
    'constants': {
        'next_int': DIR_BASE_CTE + DIR_BASE_INT,
        'next_float': DIR_BASE_CTE + DIR_BASE_FLOAT,
        'next_char': DIR_BASE_CTE + DIR_BASE_CHAR,
        'next_bool': DIR_BASE_CTE + DIR_BASE_BOOL,
        'cte': {}
    }
}
```

```
}
},
'array_pointers': {
    'next_pointer': DIR_BASE_POINTER
}
}
```

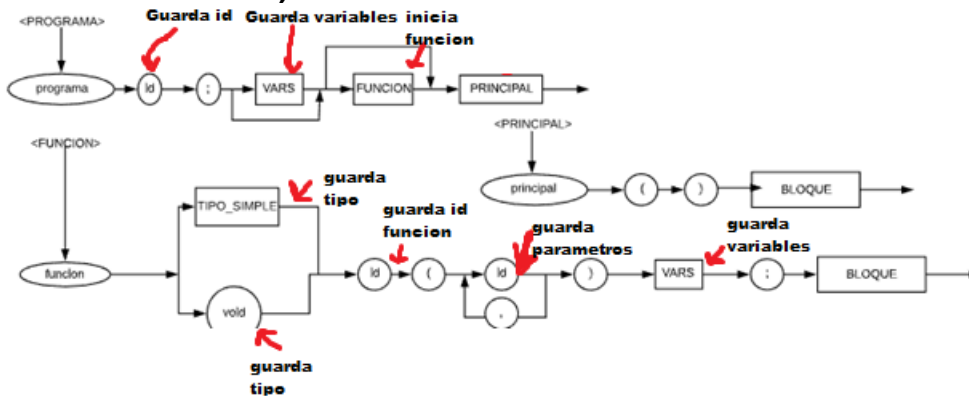
#Se incluye una de varias funciones de manejo #de memoria para el tipo de variable de 1 elemento

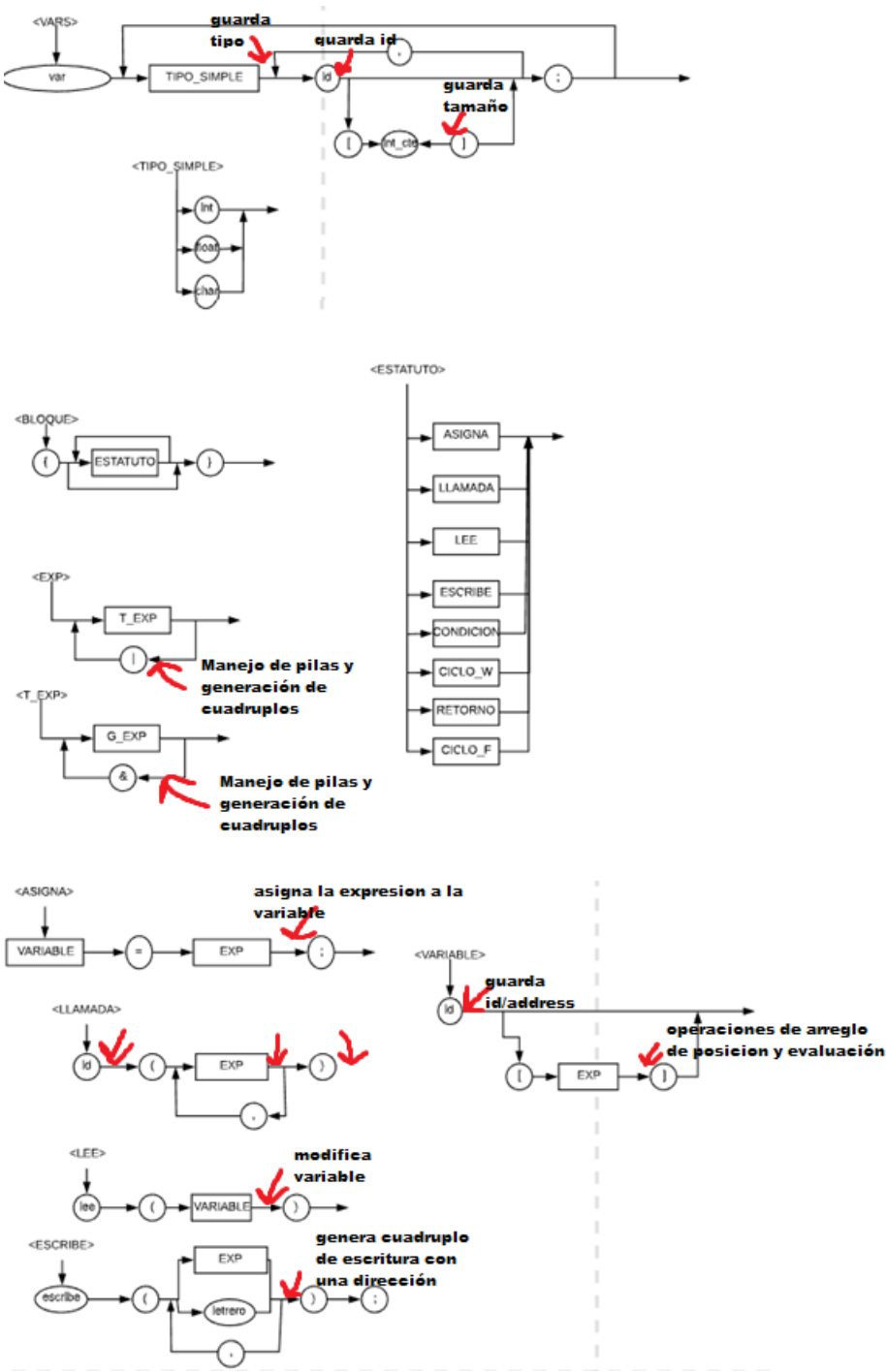
```
def get_next_var_address(p):
    global dir_func
    if current_func == 'global':
        base = DIR_BASE_GLOBAL
    else:
        base = DIR_BASE_LOCAL
```

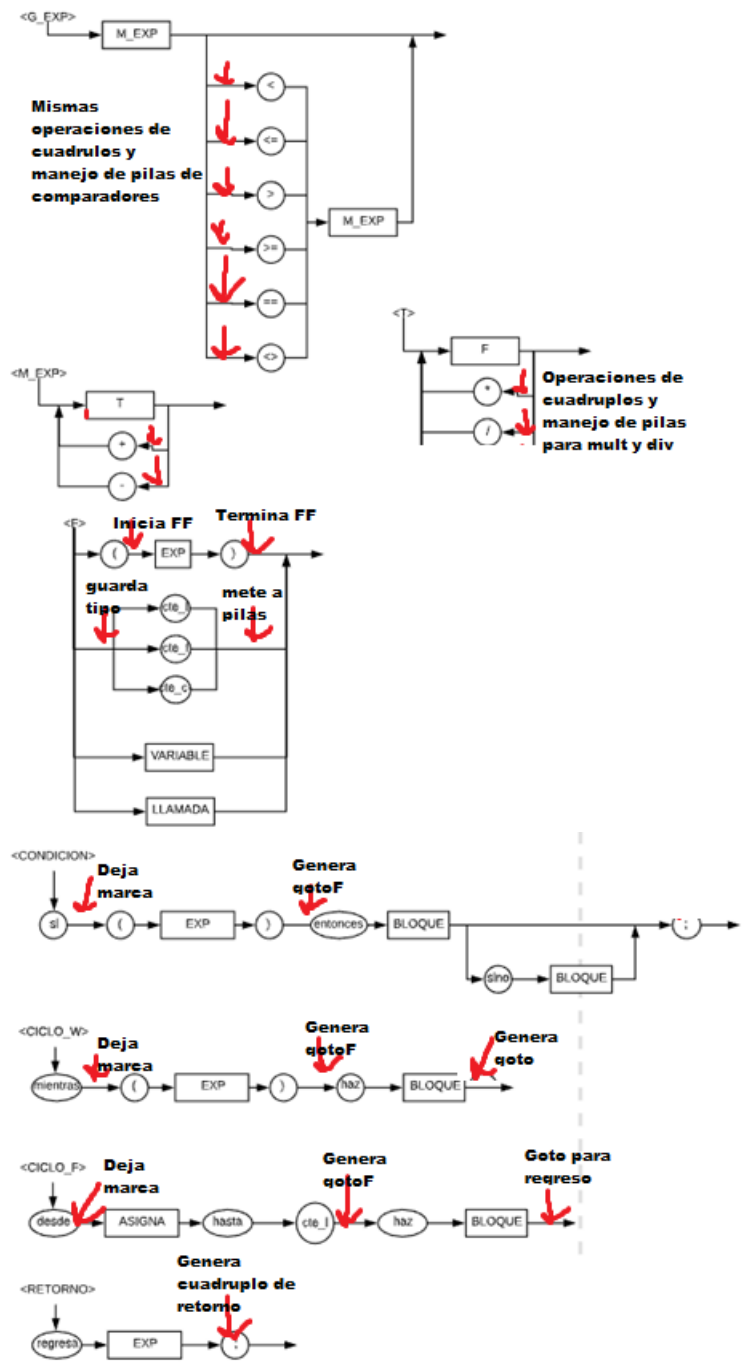
```
if(current_type == 'int'):
    baseType = 0;
elif(current_type == 'float'):
    baseType = 2500;
elif(current_type == 'char'):
    baseType = 5000;
else:
    baseType = 7500;
```

```
if dir_func[current_func]['next_' + current_type] - base
- baseType < DIR_LENGTH:
    aux = dir_func[current_func]['next_' + current_type]
    dir_func[current_func]['next_' + current_type] += 1
    return aux
else:
    error(p, "Numero de variables en su limite")
```

- Diagramas de Sintaxis con las acciones correspondientes.
- Breve descripción de cada una de las acciones semánticas y de código (no más de 2 líneas).







- **Tabla de consideraciones semánticas.**

```
from collections import defaultdict
```

```
semantic_cube = defaultdict(  
    lambda: defaultdict(lambda: defaultdict(lambda:  
        None)))
```

```
semantic_cube['int']['+']['int'] = 'int'  
semantic_cube['int']['+']['float'] = 'float'  
semantic_cube['float']['+']['int'] = 'float'  
semantic_cube['float']['+']['float'] = 'float'
```

```
semantic_cube['int']['-']['int'] = 'int'  
semantic_cube['int']['-']['float'] = 'float'  
semantic_cube['float']['-']['int'] = 'float'  
semantic_cube['float']['-']['float'] = 'float'
```

```
semantic_cube['int']['*']['int'] = 'int'  
semantic_cube['int']['*']['float'] = 'float'  
semantic_cube['float']['*']['int'] = 'float'  
semantic_cube['float']['*']['float'] = 'float'
```

```
semantic_cube['int']['/']['int'] = 'int'  
semantic_cube['int']['/']['float'] = 'float'  
semantic_cube['float']['/']['int'] = 'float'  
semantic_cube['float']['/']['float'] = 'float'
```

```
semantic_cube['int']['=']['int'] = 'int'  
semantic_cube['float']['=']['int'] = 'float'  
semantic_cube['float']['=']['float'] = 'float'  
semantic_cube['char']['=']['char'] = 'char'
```

```
semantic_cube['int']['<']['int'] = 'bool'  
semantic_cube['float']['<']['int'] = 'bool'  
semantic_cube['int']['<']['float'] = 'bool'
```

```
semantic_cube['float']['<']['float'] = 'bool'
```

```
semantic_cube['int']['<=']['int'] = 'bool'  
semantic_cube['float']['<=']['int'] = 'bool'  
semantic_cube['int']['<=']['float'] = 'bool'  
semantic_cube['float']['<=']['float'] = 'bool'
```

```
semantic_cube['int']['>']['int'] = 'bool'  
semantic_cube['float']['>']['int'] = 'bool'  
semantic_cube['int']['>']['float'] = 'bool'  
semantic_cube['float']['>']['float'] = 'bool'
```

```
semantic_cube['int']['>=']['int'] = 'bool'  
semantic_cube['float']['>=']['int'] = 'bool'  
semantic_cube['int']['>=']['float'] = 'bool'  
semantic_cube['float']['>=']['float'] = 'bool'
```

```
semantic_cube['int']['==']['int'] = 'bool'  
semantic_cube['float']['==']['int'] = 'bool'  
semantic_cube['int']['==']['float'] = 'bool'  
semantic_cube['float']['==']['float'] = 'bool'  
semantic_cube['char']['==']['char'] = 'bool'
```

```
semantic_cube['int']['<>']['int'] = 'bool'  
semantic_cube['float']['<>']['int'] = 'bool'  
semantic_cube['int']['<>']['float'] = 'bool'  
semantic_cube['float']['<>']['float'] = 'bool'  
semantic_cube['char']['<>']['char'] = 'bool'
```

```
semantic_cube['bool']['&']['bool'] = 'bool'  
semantic_cube['bool']['|']['bool'] = 'bool'
```


c.5) Descripción de memoria detallada de la compilación.

Se tiene memoria para direcciones globales en el rango de 0 a 9999, para direcciones locales(funciones) de 10000 a 19999, para constantes de 20000 a 29999 y para auxiliares apuntadores a arreglos del 30000 al 49999.

Cada uno de los rangos de 10000 está dividido en los cuatro tipos de dato que se manejan en compilación y ejecución: integer, float, char y booleano.

Se definen los rangos de direcciones por tipo de dato.

```
DIR_BASE_INT = 0
DIR_BASE_FLOAT = 2500
DIR_BASE_CHAR = 5000
DIR_BASE_BOOL = 7500
```

Se define el largo por seccion y los rangos de direcciones base.

```
DIR_LENGTH = 2500
```

```
DIR_BASE_GLOBAL = 0
DIR_BASE_LOCAL = 10000
DIR_BASE_CTE = 20000
DIR_BASE_POINTER = 30000
```

```
dir_func = {
  'global': {
    'vars': {},
    'params': {},
    'next_int': DIR_BASE_GLOBAL + DIR_BASE_INT,
    'next_float': DIR_BASE_GLOBAL + DIR_BASE_FLOAT,
    'next_char': DIR_BASE_GLOBAL + DIR_BASE_CHAR,
    'next_bool': DIR_BASE_GLOBAL + DIR_BASE_BOOL
  },
  'constants': {
    'next_int': DIR_BASE_CTE + DIR_BASE_INT,
    'next_float': DIR_BASE_CTE + DIR_BASE_FLOAT,
    'next_char': DIR_BASE_CTE + DIR_BASE_CHAR,
    'next_bool': DIR_BASE_CTE + DIR_BASE_BOOL,
    'cte': {
    }
  },
  'array_pointers': {
    'next_pointer': DIR_BASE_POINTER
  }
}
```

#Funcion de manejo de memoria para conseguir la siguiente direccion de un arreglo

#Toma en cuenta el tamaño del arreglo para dar la siguiente direccion.

```
def get_next_arr_address(p, size):
    global dir_func
    if current_func == 'global':
        base = DIR_BASE_GLOBAL
    else:
        base = DIR_BASE_LOCAL

    if(current_type == 'int'):
        baseType = 0;
    elif(current_type == 'float'):
        baseType = 2500;
    elif(current_type == 'char'):
        baseType = 5000;
    else:
        baseType = 7500;

    if dir_func[current_func]['next_' + current_type] + size - base - baseType < DIR_LENGTH:
        aux = dir_func[current_func]['next_' + current_type]
        dir_func[current_func]['next_' + current_type] += size
        return aux
    else:
```

```
error(p, "Numero de variables en su limite")

#Funcion de manejo de memoria para conseguir la siguiente direccion de una variable
def get_next_var_address(p):
    global dir_func
    if current_func == 'global':
        base = DIR_BASE_GLOBAL
    else:
        base = DIR_BASE_LOCAL

    if(current_type == 'int'):
        baseType = 0;
    elif(current_type == 'float'):
        baseType = 2500;
    elif(current_type == 'char'):
        baseType = 5000;
    else:
        baseType = 7500;

    if dir_func[current_func]['next_' + current_type] - base - baseType < DIR_LENGTH:
        aux = dir_func[current_func]['next_' + current_type]
        dir_func[current_func]['next_' + current_type] += 1
        return aux
    else:
        error(p, "Numero de variables en su limite")

#Funcion de manejo de memoria para conseguir la siguiente direccion de un apuntador
def get_next_pointer_address(p):
    global dir_func, DIR_BASE_POINTER
    if dir_func['array_pointers']['next_pointer'] - DIR_BASE_POINTER < DIR_LENGTH*4:
        aux = dir_func['array_pointers']['next_pointer']
        dir_func['array_pointers']['next_pointer'] += 1
        #dir_func['array_pointers']['pointer'][aux] = {
        #    'points_to' : None
        #}
        return aux
    else:
        error(p, "Numero de variables de arreglos en su limite")

#Funcion de manejo de memoria para conseguir la siguiente direccion de un arreglo
def get_next_cte_address(p, id):
    global dir_func, current_type
    if (id in dir_func['constants']['cte']):
        return dir_func['constants']['cte'][id]['address'];
    else:
        if(current_type == 'int'):
            baseType = 0;
        elif(current_type == 'float'):
            baseType = 2500;
        elif(current_type == 'char'):
            baseType = 5000;
        else:
            baseType = 7500;
        base = DIR_BASE_CTE;
        if(dir_func['constants']['next_' + current_type] - base - baseType < DIR_LENGTH):
            aux = dir_func['constants']['next_' + current_type];
            dir_func['constants']['next_' + current_type] += 1;
            dir_func['constants']['cte'][id] = {
                'address' : aux
            }
            return aux;
        else:
            error(p, "Numero de constantes " + current_type + " en su límite")
```

d). DESCRIPCIÓN DE LA MÁQUINA VIRTUAL.

d.1) Equipo de cómputo, lenguaje y utilerías especiales: Igual que el compilador.

d.2) Descripción detallada del proceso de manejo de memoria en ejecución.

Se utiliza una clase de memoria con operaciones de get y set.

```
#Clase de manejo de memoria con get y set properties
privada.
class Memoria:
    def __init__(self):
        # address valor
        self.__values = {}

    def set_value(self, address, value):
        #print(self.__values)
        self.__values[address] = value

    def get_value(self, address):
        #print(self.__values)
        if address not in self.__values:
            #print(address)
            #print(self.__values)
            print("Error. La direccion no se encuentra en este
            ambiente")
            sys.exit()
        return self.__values[address]
```

Y con la misma se crea un objeto de esta en la máquina virtual declarada como memoria global y de igual manera una pila de memorias para manejar los niveles de las llamadas a las funciones.

```
memorias = Stack()
```

```
#Memoria global
memoria_global = Memoria()
```

De igual manera, se utiliza durante la ejecución de la máquina virtual el get y el set para variables globales y variables en la stack de memoria a continuación

#Funcion para modificar el valor de una direccion de memoria.

```
def set_value(address, value):
    if address < top_limit_global:
        memoria_global.set_value(address, value)
    elif address < top_limit_local:
        memorias.peek().set_value(address, value)
    elif address < top_limit_cte:
        memoria_global.set_value(address, value)
    else:
        #in case of a pointer
        #if(memoria_global.isDeclared(address)):
        #    memoria_global.set_value(get_value(address), value)
        #else:
        memoria_global.set_value(address, value)
```

#Funcion para obtener el valor de una direccion de memoria.

```
def get_value(address):
    if address < top_limit_global:
        return memoria_global.get_value(address)
    elif address < top_limit_local:
        return memorias.peek().get_value(address)
    elif address < top_limit_cte:
        return memoria_global.get_value(address)
    else:
        #in case of a pointer
        #content_pointer = memoria_global.get_value(address)
        #if(memoria_global.isDeclared(get_value(address)) or memorias.peek().isDeclared(get_value(address))):
        #    return memoria_global.get_value(get_value(address))
        #else:
        return memoria_global.get_value(address)
```

Como asociación, se utilizan los valores declarados en compilación y se utilizan diccionarios en la máquina virtual para manejar estos números que ya fueron comprobados únicos y sin errores durante la compilación.

e) PRUEBAS DEL FUNCIONAMIENTO DEL LENGUAJE:

e.1)

Se corre el siguiente código:

```
programa patito;
var
int i, b, c, d;
int x;
```

```
funcion void sinParam()
var
int i;
{
i = 10;
escribe(i);
}
```

```
funcion void conParam(char c){
escribe(c);
}
```

```
funcion int fibo(int x)
{
#sinParam();
si (x == 1 | x == 0) entonces{
regresa(x);
}
sino{
regresa(fibo(x-1) + fibo(x-2));
}
}
```

```
funcion int fact(int x)
{
si(x < 0) entonces{
regresa(0);
}
si(x > 1) entonces{
regresa (x*fact(x-1));
}
sino{
regresa(1);
}
}
```

```
principal()
{
escribe('posterior');
sinParam();
conParam('asd');
escribe('introduce el valor para fibonacci y factorial');
lee(x);
escribe('fibonacci');
desde i=0 hasta x hacer{
escribe(fibo(i));
i = i+1;
}
```

```
escribe('factorial');
desde i=0 hasta x hacer{
escribe(fact(i));
i = i+1;
}
```

```
escribe('comienza fibonacci iterativo');
b = 0;
c = 1;
d = 0;
desde i = 0 hasta x hacer{
escribe(b);
d = b + c;
b = c;
c = d;
i = i+1;
}
```

```
escribe('comienza factorial iterativo');
i = 1;
b = 1;
mientras(i < x) haz {
b = b*i;
escribe(b);
i = i+1;
}
}
```

Así se ven las impresiones de ejemplo del compilador.

```
C:\Users\eduar\Desktop\ForeverAlone>fa_yacc.py testFiles/MVTestFunciones.txt
{'array_pointers': {'next_pointer': 30000},
 'conParam': {'next_bool': 17500,
               'next_char': 15001,
               'next_float': 12500,
               'next_int': 10000,
               'num_vars_bool': 0,
               'num_vars_char': 1,
               'num_vars_float': 0,
               'num_vars_int': 0,
               'params': ['char'],
               'starts': 4,
               'type': 'void'},
 'constants': {'cte': {'"asd"': {'address': 25001},
                        '"comienza factorial iterativo"': {'address': 25006},
                        '"comienza fibonacci iterativo"': {'address': 25005},
                        '"factorial"': {'address': 25004},
                        '"fibonacci"': {'address': 25003},
                        '"introduce el valor para fibonacci y factorial"': {'address': 25002},
                        '"posterior"': {'address': 25000},
                        '0': {'address': 20002},
                        '1': {'address': 20001},
                        '10': {'address': 20000},
                        '2': {'address': 20003}},
                 'next_bool': 27500,
                 'next_char': 25007,
                 'next_float': 22500,
                 'next_int': 20004},
 'fact': {'next_bool': 17502,
           'next_char': 15000,
           'next_float': 12500,
           'next_int': 10004,
           'num_vars_bool': 2,
```

Que solo se imprimen para cuestiones informativas.

<pre> 'next_int': 20004}, 'num_vars_bool': 2, 'num_vars_char': 1, 'num_vars_float': 0, 'num_vars_int': 0, 'params': ['char'], 'starts': 4, 'type': 'void'})} inicio de cuádruplos nombres 0 ['Goto', -1, -1, 40] 1 ['=', '10', -1, 'i'] 2 ['print', -1, -1, 'i'] 3 ['ENDFUNC', -1, -1, -1] 4 ['print', -1, -1, 'c'] 5 ['ENDFUNC', -1, -1, -1] 6 ['==', 'x', '1', 't1'] 7 ['==', 'x', '0', 't2'] 8 [' ', 't1', 't2', 't3'] 9 ['GotoF', 't3', -1, 12] 10 ['RETURN', -1, -1, 'x'] 11 ['Goto', -1, -1, 24] 12 ['ERA', 'fibo', -1, -1] 13 ['-', 'x', '1', 't4'] 14 ['PARAM', 't4', -1, 'par1'] 15 ['GOSUB', 'fibo', -1, 6] 16 ['=', 'fibo', -1, 't5'] 17 ['ERA', 'fibo', -1, -1] 18 ['-', 'x', '2', 't6'] 19 ['PARAM', 't6', -1, 'par1'] 20 ['GOSUB', 'fibo', -1, 6] 21 ['=', 'fibo', -1, 't7'] 22 ['+', 't5', 't7', 't8'] 23 ['RETURN', -1, -1, 't8'] 24 ['ENDFUNC', -1, -1, -1] 25 ['<', 'x', '0', 't1'] 26 ['GotoF', 't1', -1, 28] 27 ['RETURN', -1, -1, '0'] 28 ['>', 'x', '1', 't2'] 29 ['GotoF', 't2', -1, 38] 30 ['ERA', 'fact', -1, -1] 31 ['-', 'x', '1', 't3'] 32 ['PARAM', 't3', -1, 'par1'] 33 ['GOSUB', 'fact', -1, 25] 34 ['=', 'fact', -1, 't4'] </pre>	<pre> inicio de cuádruplos de direcciones 0 ['Goto', -1, -1, 40] 1 ['=', 20000, -1, 10000] 2 ['print', -1, -1, 10000] 3 ['ENDFUNC', -1, -1, -1] 4 ['print', -1, -1, 15000] 5 ['ENDFUNC', -1, -1, -1] 6 ['==', 10000, 20001, 17500] 7 ['==', 10000, 20002, 17501] 8 [' ', 17500, 17501, 17502] 9 ['GotoF', 17502, -1, 12] 10 ['RETURN', -1, -1, 10000] 11 ['Goto', -1, -1, 24] 12 ['ERA', 'fibo', -1, -1] 13 ['-', 10000, 20001, 10001] 14 ['PARAM', 10001, -1, 10000] 15 ['GOSUB', 5, -1, 6] 16 ['=', 5, -1, 10002] 17 ['ERA', 'fibo', -1, -1] 18 ['-', 10000, 20003, 10003] 19 ['PARAM', 10003, -1, 10000] 20 ['GOSUB', 5, -1, 6] 21 ['=', 5, -1, 10004] 22 ['+', 10002, 10004, 10005] 23 ['RETURN', -1, -1, 10005] 24 ['ENDFUNC', -1, -1, -1] 25 ['<', 10000, 20002, 17500] 26 ['GotoF', 17500, -1, 28] 27 ['RETURN', -1, -1, 20002] 28 ['>', 10000, 20001, 17501] 29 ['GotoF', 17501, -1, 38] 30 ['ERA', 'fact', -1, -1] 31 ['-', 10000, 20001, 10001] 32 ['PARAM', 10001, -1, 10000] 33 ['GOSUB', 6, -1, 25] 34 ['=', 6, -1, 10002] </pre>
---	--

Y así se ve la ejecución en la máquina virtual:

```
C:\Users\eduar\Desktop\ForeverAlone>maquinavirtual.py testFiles/MVTestFunciones.txt.obj
'posterior'
10
'asd'
'introduce el valor para fibonacci y factorial'
8
'fibonacci'
0
1
1
2
3
5
8
13
'factorial'
1
1
2
6
24
120
720
5040
'comienza fibonacci iterativo'
0
1
1
2
3
5
8
13
'comienza factorial iterativo'
1
2
6
24
120
720
5040
```

f) LISTADOS PERFERTAMENTE DOCUMENTADOS DEL PROYECTO.

```

import ply.yacc as yacc
import sys
from fa_lex import tokens
from semantic_cube import semantic_cube
from pprint import pprint
from stack import Stack

# Se definen los rangos de direcciones por tipo de dato.
DIR_BASE_INT = 0
DIR_BASE_FLOAT = 2500
DIR_BASE_CHAR = 5000
DIR_BASE_BOOL = 7500

# Se define el largo por seccion y los rangos de
direcciones base.
DIR_LENGTH = 2500

DIR_BASE_GLOBAL = 0
DIR_BASE_LOCAL = 10000
DIR_BASE_CTE = 20000
DIR_BASE_POINTER = 30000

# Contador de parametros y se guarda la función en la
que esta la variable llamada.
address_func_var_global = None
parameter_counter = 0
retornoFuncion = None

curr_Call = None

#Clase operando para el push a la pila de operandos
con id y con address
class Operando:
    def __init__(self):
        self.id = None
        self.address = None

#contadores para manejo de temporales de cuatruplos
legibles.
currTemp = 1;
currTempArr = 1;
#variable para tipo de expresion global
current_type = ""
current_exp = None
#inicializa la funcion actual como global
current_func = 'global'

#se inicializa el directorio de funciones con las
direcciones base de
#integros de variables globales, de constantes y de
apuntadores a arreglos.
dir_func = {
    'global': {
        'vars': {},
        'params': {},
        'next_int': DIR_BASE_GLOBAL + DIR_BASE_INT,
        'next_float': DIR_BASE_GLOBAL +
DIR_BASE_FLOAT,
        'next_char': DIR_BASE_GLOBAL +
DIR_BASE_CHAR,
        'next_bool': DIR_BASE_GLOBAL +
DIR_BASE_BOOL
    },
    'constants': {
        'next_int': DIR_BASE_CTE + DIR_BASE_INT,
        'next_float': DIR_BASE_CTE + DIR_BASE_FLOAT,
        'next_char': DIR_BASE_CTE + DIR_BASE_CHAR,
        'next_bool': DIR_BASE_CTE + DIR_BASE_BOOL,
    }
},
    'array_pointers': {
        'next_pointer': DIR_BASE_POINTER
    }
}

# dir typo
read_quadruples = []
dir_quadruples = []

#manejo operadores
popper = Stack();
pilaOp = Stack();
pilaTipos = Stack();
pilaSaltos = Stack();
#pila avail

success = True

## Funcion que inicializa el programa e imprime los
cuatruplos y el directorio de funciones.
def p_programa(p):
    """
    programa : PROGRAMA p_n_mainJump ID
    SEMICOLON vars mult_funcion principal
              | PROGRAMA p_n_mainJump ID SEMICOLON
mult_funcion principal
              | PROGRAMA p_n_mainJump ID SEMICOLON
vars principal
              | PROGRAMA p_n_mainJump ID SEMICOLON
principal
    """
    del dir_func['global']['vars']
    pprint(dir_func)
    print("inicio de cuatruplos nombres", '\t\t', 'inicio de
cuatruplos de direcciones')
    for i, cuatruplo in enumerate(read_quadruples):
        print(i, cuatruplo, " ", '\t\t', i, dir_quadruples[i])

#genera cuatruplo para el salto a la funcion principal
para comenzar ejecución.
def p_n_mainJump(p):
    """
    p_n_mainJump :
    """
    global dir_quadruples, read_quadruples
    temp_quad = ['Goto', -1, -1, None]
    dir_quadruples.append(temp_quad)
    read_quadruples.append(temp_quad)

#Grupo de funciones para registrar el cuatruplo donde
empieza principal.
def p_principal(p):
    """
    principal : PRINCIPAL n_register_glob LPAREN
    RPAREN bloque
    """

def p_n_register_glob(p):
    """
    n_register_glob :
    """
    global current_func, read_quadruples, dir_quadruples
    current_func = 'global'

```

```

read_quadruples[0][3] = len(read_quadruples)
dir_quadruples[0][3] = len(dir_quadruples)

# variable declaration
def p_vars(p):
    """
    vars : VAR vars_aux
    """

    ## seccion de vars para definir varios tipos de
    id con o sin brackets
    def p_vars_aux(p):
        """
        vars_aux : tipo_simple vars_aux1 SEMICOLON
                  | tipo_simple vars_aux1
        SEMICOLON vars_aux
        """

        ##seccion de vars para ciclo de varias id con brackets
        def p_vars_aux1(p):
            """
            vars_aux1 : vars_aux2
                      | vars_aux2 COMMA vars_aux1
            """

            ## seccion de vars para id con brackets
            def p_vars_aux2(p):
                """
                vars_aux2 : ID n_save_var
                          | ID LSQUARE CTE_I n_save_array
            RSQUARE
            """

            #Codigo para agregar arreglos, igual a variable pero con
            tamaño.
            def p_n_save_array(p):
                """
                n_save_array :
                """
                global current_func, current_type, dir_func
                id = p[-3]
                size = p[-1]
                if(id in dir_func[current_func]):
                    error(p, 'La variable ya fue instanciada')
                else:
                    dir_func[current_func]['vars'][id] = {
                        'type' : current_type,
                        'address' : get_next_arr_address(p, int(size)),
                        'size' : int(size)
                    }

            #Guarda variable con su tipo y direccion en el directorio
            de funciones.
            def p_n_save_var(p):
                """
                n_save_var :
                """
                global current_func, current_type, dir_func
                id = p[-1]
                if (id in dir_func[current_func]['vars'] or id in
                dir_func[current_func]['params']):
                    error(p, 'La variable ya fue instanciada')
                else:
                    dir_func[current_func]['vars'][id] = {
                        'type' : current_type,
                        'address': get_next_var_address(p)
                    }

```

```

#Funcion de manejo de memoria para conseguir la
siguiente direccion de un arreglo
#Toma en cuenta el tamaño del arreglo para dar la
siguiente direccion.
def get_next_arr_address(p, size):
    global dir_func
    if current_func == 'global':
        base = DIR_BASE_GLOBAL
    else:
        base = DIR_BASE_LOCAL

    if(current_type == 'int'):
        baseType = 0;
    elif(current_type == 'float'):
        baseType = 2500;
    elif(current_type == 'char'):
        baseType = 5000;
    else:
        baseType = 7500;

    if dir_func[current_func]['next_' + current_type] + size
    - base - baseType < DIR_LENGTH:
        aux = dir_func[current_func]['next_' + current_type]
        dir_func[current_func]['next_' + current_type] +=
        size
        return aux
    else:
        error(p, "Numero de variables en su limite")

#Funcion de manejo de memoria para conseguir la
siguiente direccion de una variable
def get_next_var_address(p):
    global dir_func
    if current_func == 'global':
        base = DIR_BASE_GLOBAL
    else:
        base = DIR_BASE_LOCAL

    if(current_type == 'int'):
        baseType = 0;
    elif(current_type == 'float'):
        baseType = 2500;
    elif(current_type == 'char'):
        baseType = 5000;
    else:
        baseType = 7500;

    if dir_func[current_func]['next_' + current_type] - base
    - baseType < DIR_LENGTH:
        aux = dir_func[current_func]['next_' + current_type]
        dir_func[current_func]['next_' + current_type] += 1
        return aux
    else:
        error(p, "Numero de variables en su limite")

#Funcion de manejo de memoria para conseguir la
siguiente direccion de un apuntador
def get_next_pointer_address(p):
    global dir_func, DIR_BASE_POINTER
    if dir_func['array_pointers']['next_pointer'] -
    DIR_BASE_POINTER < DIR_LENGTH*4:
        aux = dir_func['array_pointers']['next_pointer']
        dir_func['array_pointers']['next_pointer'] += 1
        #dir_func['array_pointers']['pointer'][aux] = {
        #    'points_to' : None
        #}
        return aux
    else:
        error(p, "Numero de variables de arreglos en su
        limite")

```



```
#Funcion de manejo de memoria para conseguir la
siguiente direccion de un arreglo
def get_next_cte_address(p, id):
    global dir_func, current_type
    if (id in dir_func['constants']['cte']):
        return dir_func['constants']['cte'][id]['address'];
    else:
        if(current_type == 'int'):
            baseType = 0;
        elif(current_type == 'float'):
            baseType = 2500;
        elif(current_type == 'char'):
            baseType = 5000;
        else:
            baseType = 7500;
        base = DIR_BASE_CTE;
        if(dir_func['constants']['next_'+current_type] - base
- baseType < DIR_LENGTH):
            aux = dir_func['constants']['next_'+current_type];
            dir_func['constants']['next_'+current_type] += 1;
            dir_func['constants']['cte'][id] = {
                'address': aux
            }
            return aux;
        else:
            error(p, "Numero de constantes " + current_type
+ " en su límite")
```

#Funciones auxiliares para guardar el tipo de la expresion.

```
def p_tipo_simple(p):
    """
    tipo_simple : INT n_save_type
                  | FLOAT n_save_type
                  | CHAR n_save_type
    """
```

```
def p_n_save_type(p):
    """n_save_type : """
    global current_type
    current_type = p[-1]
```

```
def p_empty(p):
    """
    empty :
    """
```

#Funcion para llamada a la variable, se apeg a la gramática con subfunciones.

```
def p_variable(p):
    """
    variable : ID n_getVarVal LSQUARE n_start_FF
n_hasDim mult_exp RSQUARE n_end_FF n_arr_quad
              | ID n_getVarVal
    """
```

#Maneja las stacks de tipo y operando así como la generación de cuádruplos

#Funcion específica de arreglos y de generación de verificación.

```
def p_n_arr_quad(p):
    """
    n_arr_quad :
    """
    global read_quadruples, dir_quadruples, pilaTipos,
currTemp, current_type, current_func, currTempArr
    temp_exp = pilaOp.pop()
    temp_array = pilaOp.pop()
```

```
temp_exp_type = pilaTipos.pop()
if(temp_exp_type != 'int'):
    error(p, 'El indice del arreglo debe ser un íntegro')
id = temp_array.id
if(id in dir_func[current_func]['vars']):
    aux = current_func
else:
    aux = 'global'
```

```
current_type = 'int'
temp_arr_limit =
dir_func[aux]['vars'][temp_array.id]['size']
temp_cte = get_next_cte_address(p,
str(temp_arr_limit))
```

```
temp_quad = ['VERIFY', temp_exp.id, -1,
temp_arr_limit]
read_quadruples.append(temp_quad)
temp_quad = ['VERIFY', temp_exp.address, -1,
temp_cte]
dir_quadruples.append(temp_quad)
```

```
temp_cte = get_next_cte_address(p,
str(temp_array.address))
dir_array = get_next_pointer_address(p)
```

```
temp_quad = ['+', temp_exp.id, temp_array.id,
dir_array]
read_quadruples.append(temp_quad)
temp_quad = ['+', temp_exp.address, temp_cte,
dir_array]
dir_quadruples.append(temp_quad)
```

```
temporalArr = 't' + str(currTempArr)
currTempArr += 1;
toPush = Operando()
toPush.id = temporalArr
toPush.address = dir_array
pilaOp.push(toPush)
```

#Revisa si una variable que se identifica como arreglo requiere dimensiones

```
def p_n_hasDim(p):
    """
```

```
n_hasDim :
"""
global dir_func, current_func, pilaOp, pilaTipos
temp_op = pilaOp.peek()
#know which function it is in
if(temp_op.id in dir_func[current_func]['vars']):
    aux = current_func
else:
    aux = 'global'
```

```
#print(len(dir_func[aux]['vars'][temp_op.id]))
if(len(dir_func[aux]['vars'][temp_op.id]) < 3):
    error(p, 'La variable que se está tratando de acceder
requiere dimensión')
```

#Obtiene el valor de una variable, se usa para no dimensionadas y dimensionadas

#Obtiene su tipo y si id.

```
def p_n_getVarVal(p):
    """
    n_getVarVal :
    """
    global current_exp, pilaOp, dir_func, current_func,
pilaTipos
#guarda exp para funciones posteriores
id = p[-1]
```

```

current_exp = id
temp_op = Operando()
##segunda parte para cuádruplos
temp_op.id = id
if(id in dir_func[current_func]['vars']):
    temp_op.address =
dir_func[current_func]['vars'][id]['address']
pilaOp.push(temp_op)
type = dir_func[current_func]['vars'][id]['type']
pilaTipos.push(type)
else:
    if(id in dir_func['global']['vars']):
        temp_op.address =
dir_func['global']['vars'][id]['address']
pilaOp.push(temp_op)
type = dir_func['global']['vars'][id]['type']
pilaTipos.push(type)
    else:
        error(p, "La variable no se encuentra en el
ambiente")

#Conjunto de funciones para tener Funciones
#De tipo void o con retorno de char int o float.
#Con parametros y sin parametros.
def p_mult_funcion(p):
    """
    mult_funcion : funcion
                  | funcion mult_funcion
    """

def p_funcion(p):
    """
    funcion : FUNCION tipo_simple ID n_register_func
LPAREN param RPAREN vars bloque n_endof_func
            | FUNCION tipo_simple ID
n_register_func LPAREN param RPAREN bloque
n_endof_func
            | FUNCION VOID n_save_type ID
n_register_func LPAREN param RPAREN vars bloque
n_endof_func
            | FUNCION VOID n_save_type ID n_register_func
LPAREN param RPAREN bloque n_endof_func
            | FUNCION tipo_simple ID n_register_func
LPAREN RPAREN vars bloque n_endof_func
            | FUNCION tipo_simple ID n_register_func
LPAREN RPAREN bloque n_endof_func
            | FUNCION VOID n_save_type ID n_register_func
LPAREN RPAREN vars bloque n_endof_func
            | FUNCION VOID n_save_type ID n_register_func
LPAREN RPAREN bloque n_endof_func
    """
    global dir_func
    del dir_func[current_func]['vars']

#Inicializa la funcion con su pertinente información
#Numero de variables tipo, en qué cuádruplo comienza,
y su manejador de memoria.
def p_n_register_func(p):
    """n_register_func : """
    global dir_func, current_func, current_type
    if (p[-1] in dir_func):
        error(p, 'La función ya existe')
    else:
        current_func = p[-1]
        dir_func[current_func] = {
            'type': current_type,
            'params': [],
            'vars': {},
            'num_vars_int': None,
            'num_vars_float': None,

```

```

            'num_vars_char': None,
            'num_vars_bool': None,
            'starts': len(dir_quadruples),
            'next_int': DIR_BASE_LOCAL + DIR_BASE_INT,
            'next_float': DIR_BASE_LOCAL +
DIR_BASE_FLOAT,
            'next_char': DIR_BASE_LOCAL +
DIR_BASE_CHAR,
            'next_bool': DIR_BASE_LOCAL +
DIR_BASE_BOOL
        }
        #guardando una variable global con el nombre de la
funcion si no es VOID
        if(current_type != 'void'):
            current_func = 'global'
            if p[-1] in dir_func['global']['vars']:
                error(p, "Una funcion y una variable global no
pueden tener el mismo nombre")

    else:
        dir_func['global']['vars'][p[-1]] = {
            'type': current_type,
            'address': get_next_var_address(p)
        }
        current_func = p[-1]

#Funcion para llamada a funciones con parámetro
def p_param(p):
    """
    param : tipo_simple param_aux1
    """

    ##seccion de vars para ciclo de varias id con brackets
def p_param_aux1(p):
    """
    param_aux1 : ID save_param
                | ID save_param COMMA param
    """

#Funcion que revisa si un parámetro de los que
#se tienen en la declaración de la función ya existe.
def p_save_param(p):
    """
    save_param :
    """
    global current_func, dir_func, current_type
    id = p[-1]
    if(id in dir_func[current_func]['vars']):
        error(p, "Parametro ya declarado")
    else:
        dir_func[current_func]['params'].append(current_type)
        dir_func[current_func]['vars'][id] = {
            'type': current_type,
            'address': get_next_var_address(p)
        }

#Punto neurálgico para el final de la función donde se
reinicial las variables
#Y se genera cuádruplo de terminación de la función.
def p_n_endof_func(p):
    """
    n_endof_func :
    """
    global dir_func, current_func, read_quadruples,
dir_quadruples, currTemp
    dir_func[current_func]['num_vars_int'] =
dir_func[current_func]['next_int'] - DIR_BASE_INT -
DIR_BASE_LOCAL

```

```

    dir_func[current_func]['num_vars_float'] =
dir_func[current_func]['next_float'] - DIR_BASE_FLOAT
- DIR_BASE_LOCAL
    dir_func[current_func]['num_vars_char'] =
dir_func[current_func]['next_char'] - DIR_BASE_CHAR -
DIR_BASE_LOCAL
    dir_func[current_func]['num_vars_bool'] =
dir_func[current_func]['next_bool'] - DIR_BASE_BOOL -
DIR_BASE_LOCAL
    temp_quad = ['ENDFUNC', -1, -1, -1]
    read_quadruples.append(temp_quad);
    dir_quadruples.append(temp_quad);
    currTemp = 1

```

#Funciones para la gramática de estatutos.

def p_bloque(p):

```

'''
    bloque : LBRACKET mult_estatutos RBRACKET
           | LBRACKET empty RBRACKET
'''

```

def p_mult_estatutos(p):

```

'''
    mult_estatutos : estatuto
                   | estatuto
'''

```

mult_estatutos

'''

def p_estatuto(p):

```

'''
    estatuto : asigna SEMICOLON
             | llamada SEMICOLON
             | lee
             | escribe
             | condicion
             | ciclo_w
             | retorno
             | ciclo_f
'''

```

#Conjunto de funciones para la acción de asignación.

def p_asigna(p):

```

'''
    asigna : mult_asigna
'''

```

def p_mult_asigna(p):

```

'''
    mult_asigna : variable EQUAL n_Operador mult_exp
n_asignQuad
               | variable EQUAL n_Operador mult_asigna
n_asignQuad
'''

```

def p_n_asignQuad(p):

```

'''
    n_asignQuad :
'''
    global popper, pilaOp, pilaTipos, read_quadruples,
currTemp, dir_quadruples
    if(not popper.isEmpty()):
        aux = popper.peek()
        if(aux == '='):
            opDerecho = pilaOp.pop();
            tipoDerecho = pilaTipos.pop();
            oplzquierdo = pilaOp.pop();
            tipolzquierdo = pilaTipos.pop();
            operador = popper.pop()

```

```

    tipoResultado =
semantic_cube[tipolzquierdo][operador][tipoDerecho]

```

```

    if(tipoResultado != None):
        tempQuad = [operador, opDerecho.id, -1,
oplzquierdo.id]
        pilaOp.push(oplzquierdo)
        pilaTipos.push(tipoResultado)
        read_quadruples.append(tempQuad)
        tempQuad = [operador, opDerecho.address, -1,
oplzquierdo.address]
        dir_quadruples.append(tempQuad)

```

```

    else:
        error(p, "Tipo no compatible para la operacion
de asignación")

```

#Función para tener el exponente de un parámetro.

def p_param_exp(p):

```

'''
    param_exp : mult_exp n_parameter_action
              | mult_exp n_parameter_action COMMA
param_exp
'''

```

#Funcion para generar cuádruplos de parámetros.

def p_n_parameter_action(p):

```

'''
    n_parameter_action :
'''
    global pilaOp, pilaTipos, dir_func, curr_Call,
read_quadruples, dir_quadruples, parameter_counter
    argument = pilaOp.pop()
    argumentType = pilaTipos.pop()
    if dir_func[curr_Call]['params'][parameter_counter] ==
argumentType:
        parameter = 'par' + str(parameter_counter+1)
        temp_quad = ['PARAM', argument.id, -1, parameter]
        read_quadruples.append(temp_quad)

```

```

    #obteniendo la direccion del parametro
    Address_param =
getParamAddress(parameter_counter, argumentType)

```

```

    temp_quad = ['PARAM', argument.address, -1,
Address_param]
    dir_quadruples.append(temp_quad)
    parameter_counter += 1
    else:
        error(p, "El parametro "+ str(parameter_counter+1)
+ " de la funcion es incorrecto");

```

#Manejo de memoria de los parámetros.

def getParamAddress(counter, type):

```

    global current_func
    tempBase = 0
    if type == 'int':
        tempBase = 0;
    elif type == 'float':
        tempBase = 2500;
    else:
        tempBase = 5000;
    return counter + tempBase + DIR_BASE_LOCAL

```

#Conjunto de funciones para llamada a funciones.

def p_llamada(p):

'''

```

    llamada : ID n_verify_func LPAREN n_start_FF
n_start_pcounter param_exp RPAREN n_end_FF
n_last_param_action
    | ID n_verify_func LPAREN n_start_FF
n_start_pcounter RPAREN n_end_FF
n_last_param_action
'''

#Ultimo punto neurálgico que revisa consistencia entre
numero de parámetros y parámetros declarados.
#Empuja a la pila de operandos el resultado de la
llamada como cualquier otra expresion.
def p_n_last_param_action(p):
'''
    n_last_param_action :
'''
    global dir_func, read_quadruples, dir_quadruples,
pilaOp, currTemp, current_type, parameter_counter,
address_func_var_global
    if(parameter_counter <
len(dir_func[curr_Call]['params'])):
        error(p, 'Se declararon menos parámetros de los
requeridos por la función')
    else:
        temp_quad = ['GOSUB', curr_Call, -1,
dir_func[curr_Call]['starts']]
        read_quadruples.append(temp_quad)
        #add call address
        dir_quadruples.append(temp_quad)

        if(dir_func[curr_Call]['type'] != 'void'):
            address_func_var_global =
dir_func['global']['vars'][curr_Call]['address']
            dir_quadruples.pop()
            temp_quad = ['GOSUB',
address_func_var_global, -1,
dir_func[curr_Call]['starts']]
            dir_quadruples.append(temp_quad)
            temp_op = Operando()

pilaTipos.push(dir_func['global']['vars'][curr_Call]['type']
)

    temporal = 't' + str(currTemp)
    currTemp +=1;
    temp_op.id = temporal
    current_type = dir_func[curr_Call]['type']
    temp_op.address = get_next_var_address(p)
    temp_quad = ['=', curr_Call, -1, temp_op.id]
    read_quadruples.append(temp_quad)
    temp_quad = ['=', address_func_var_global, -1,
temp_op.address]

    dir_quadruples.append(temp_quad)
    pilaOp.push(temp_op)

#Funcion para contar el número de parámetros.
def p_n_start_pcounter(p):
'''
    n_start_pcounter :
'''
    global dir_func, parameter_counter, read_quadruples,
dir_quadruples, curr_Call, address_func_var_global
    parameter_counter = 0
    temp_quad = ['ERA', curr_Call, -1, -1];
    read_quadruples.append(temp_quad)
    #if(dir_func[curr_Call]['type'] != 'void'):
    #    address_func_var_global =
dir_func['global']['vars'][curr_Call]['address']
    #temp_quad = ['ERA', address_func_var_global, -1,
-1];

```

```

    dir_quadruples.append(temp_quad)

#Verifica que la función exista.
def p_n_verify_func(p):
'''
    n_verify_func :
'''
    global dir_func, curr_Call
    if(p[-1] not in dir_func):
        error(p, 'La función que se está llamando no existe')
    else:
        curr_Call = p[-1]

#Genera cuádruplo de lectura para el usuario.
def p_lee(p):
'''
    lee : LEE LPAREN variable RPAREN SEMICOLON
'''
    global dir_func, dir_quadruples, read_quadruples,
pilaOp
    elemento = pilaOp.pop()

    temp_quad = ['read', -1, -1, elemento.id]
    read_quadruples.append(temp_quad)
    temp_quad = ['read', -1, -1, elemento.address]
    dir_quadruples.append(temp_quad)

#Conjunto de funciones para generación del cuádruplo
escribe.
def p_escribe(p):
'''
    escribe : ESCRIBE LPAREN mult_exp n_escribeExp
RPAREN SEMICOLON
            | ESCRIBE LPAREN mult_cte_s
n_escribeExp RPAREN SEMICOLON
'''

def p_n_escribeExp(p):
'''
    n_escribeExp :
'''
    global dir_func, read_quadruples, current_exp,
pilaOp, dir_quadruples
    operando = pilaOp.pop();
    currQuad = ['print', -1, -1, operando.id]
    read_quadruples.append(currQuad)
    currQuad = ['print', -1, -1, operando.address]
    dir_quadruples.append(currQuad)

#Conjunto de expresiones para expresiones mult div
exp llamadas etc.
def p_mult_cte_s(p):
'''
    mult_cte_s : CTE_S
            | CTE_S COMMA mult_cte_s
'''

def p_mult_exp(p):
'''
    mult_exp : exp
            | exp COMMA mult_exp
'''

def p_exp(p):
'''
    exp : t_exp n_orQuad
            | t_exp n_orQuad OR n_Operador exp
'''
#Genera cuádruplo y maneja pilas para operaciones de
OR

```

```
def p_n_orQuad(p):
    """
    n_orQuad :
    """
    global popper, pilaOp, pilaTipos, read_quadruples,
    currTemp, dir_quadruples, current_type
    if(not popper.isEmpty()):
        aux = popper.peek()
        if(aux == '|'):
            opDerecho = pilaOp.pop();
            tipoDerecho = pilaTipos.pop();
            oplzquierdo = pilaOp.pop();
            tipolzquierdo = pilaTipos.pop();
            operador = popper.pop()
            tipoResultado =
            semantic_cube[tipolzquierdo][operador][tipoDerecho]
            current_type = tipoResultado
            if(tipoResultado != None):
                #result avail.next()
                temporal = 't' + str(currTemp)
                temp_op = Operando();
                temp_op.id = temporal;

                temp_op.address = get_next_var_address(p);

                tempQuad = [operador, oplzquierdo.id,
                opDerecho.id, temp_op.id]
                currTemp = currTemp + 1;

                pilaOp.push(temp_op)
                pilaTipos.push(tipoResultado)
                read_quadruples.append(tempQuad)
                tempQuad = [operador, oplzquierdo.address,
                opDerecho.address, temp_op.address]
                dir_quadruples.append(tempQuad)

            else:
                error(p, "Tipo no compatible para la operacion
                de OR")
```

```
def p_t_exp(p):
    """
    t_exp : g_exp n_andQuad
    | g_exp n_andQuad AND n_Operador t_exp
    """
    #Genera cuadruplo y maneja pilas para operaciones de
    AND
    def p_n_andQuad(p):
        """
        n_andQuad :
        """
        global popper, pilaOp, pilaTipos, read_quadruples,
        currTemp, dir_quadruples, current_type
        if(not popper.isEmpty()):
            aux = popper.peek()
            if(aux == '&'):
                opDerecho = pilaOp.pop();
                tipoDerecho = pilaTipos.pop();
                oplzquierdo = pilaOp.pop();
                tipolzquierdo = pilaTipos.pop();
                operador = popper.pop()
                tipoResultado =
                semantic_cube[tipolzquierdo][operador][tipoDerecho]
                current_type = tipoResultado
                if(tipoResultado != None):
                    #result avail.next()
                    temporal = 't' + str(currTemp)
                    temp_op = Operando();
                    temp_op.id = temporal;
```

```
temp_op.address = get_next_var_address(p)

tempQuad = [operador, oplzquierdo.id,
opDerecho.id, temp_op.id]
currTemp = currTemp + 1;

pilaOp.push(temp_op)
pilaTipos.push(tipoResultado)
read_quadruples.append(tempQuad)
tempQuad = [operador, oplzquierdo.address,
opDerecho.address, temp_op.address]
dir_quadruples.append(tempQuad);
else:
    error(p, "Tipo no compatible para la operacion
    de AND")
```

```
def p_g_exp(p):
    """
    g_exp : m_exp n_compareQuad
    | m_exp n_compareQuad LESSTHAN
    n_Operador g_exp
    | m_exp n_compareQuad LESSEQUAL n_Operador
    g_exp
    | m_exp n_compareQuad GREATERTHAN
    n_Operador g_exp
    | m_exp n_compareQuad GREATEREQUAL
    n_Operador g_exp
    | m_exp n_compareQuad SAME n_Operador g_exp
    | m_exp n_compareQuad NOEQUAL n_Operador
    g_exp
    """
```

#Genera cuadruplo y maneja pilas para operaciones de comparativos

```
def p_n_compareQuad(p):
    """
    n_compareQuad :
    """
    global popper, pilaOp, pilaTipos, read_quadruples,
    currTemp, dir_quadruples, current_type
    if(not popper.isEmpty()):
        aux = popper.peek()
        if(aux == '<' or aux == '<=' or aux == '>' or aux ==
        '>='):
            or aux == '==' or aux == '<>'):
                opDerecho = pilaOp.pop();
                tipoDerecho = pilaTipos.pop();
                oplzquierdo = pilaOp.pop();
                tipolzquierdo = pilaTipos.pop();
                operador = popper.pop()
                tipoResultado =
                semantic_cube[tipolzquierdo][operador][tipoDerecho]
                current_type = tipoResultado
                if(tipoResultado != None):
                    #result avail.next()
                    temporal = 't' + str(currTemp)
                    temp_op = Operando();
                    temp_op.id = temporal;
                    temp_op.address = get_next_var_address(p);

                    tempQuad = [operador, oplzquierdo.id,
                    opDerecho.id, temp_op.id]
                    currTemp = currTemp + 1;
                    pilaOp.push(temp_op)
                    pilaTipos.push(tipoResultado)
                    read_quadruples.append(tempQuad)
                    tempQuad = [operador, oplzquierdo.address,
                    opDerecho.address, temp_op.address];
```

```

        dir_quadruples.append(tempQuad)
    else:
        error(p, "Tipo no compatible para la operacion
de comparación")

```

```
def p_m_exp(p):
```

```

'''
    m_exp : t n_sumQuad
            | t n_sumQuad PLUS n_Operador m_exp
            | t n_sumQuad MINUS n_Operador m_exp
'''
#Genera cuádruplo y maneja pilas para operaciones de
suma y resta
def p_n_sumQuad(p):
'''
    n_sumQuad :
'''
    global popper, pilaOp, pilaTipos, read_quadruples,
currTemp, dir_quadruples, current_type
    if(not popper.isEmpty()):
        aux = popper.peak()
        if(aux == '+' or aux == '-'):
            opDerecho = pilaOp.pop();
            tipoDerecho = pilaTipos.pop();
            oplzquierdo = pilaOp.pop();
            tipolzquierdo = pilaTipos.pop();
            operador = popper.pop()
            tipoResultado =
semantic_cube[tipolzquierdo][operador][tipoDerecho]
            current_type = tipoResultado
            if(tipoResultado != None):
                #result avail.next()
                temporal = 't' + str(currTemp)
                temp_op = Operando();
                temp_op.id=temporal;
                temp_op.address = get_next_var_address(p);

                tempQuad = [operador, oplzquierdo.id,
opDerecho.id, temp_op.id]
                currTemp = currTemp + 1;
                pilaOp.push(temp_op)
                pilaTipos.push(tipoResultado)
                read_quadruples.append(tempQuad)
                tempQuad = [operador, oplzquierdo.address,
opDerecho.address, temp_op.address]
                dir_quadruples.append(tempQuad)
            else:
                error(p, "Tipo no compatible para la operacion
de suma/resta")

```

```
def p_t(p):
```

```

'''
    t : f n_multQuad
        | f n_multQuad MULT n_Operador t
        | f n_multQuad DIV n_Operador t
'''
#Genera cuádruplo y maneja pilas para operaciones de
multiplicacion y division
def p_n_multQuad(p):
'''
    n_multQuad :
'''
    global popper, pilaOp, pilaTipos, read_quadruples,
currTemp, dir_quadruples, current_type
    if(not popper.isEmpty()):

```

```

        aux = popper.peak()
        if(aux == '*' or aux == '/'):
            opDerecho = pilaOp.pop();
            tipoDerecho = pilaTipos.pop();
            oplzquierdo = pilaOp.pop();
            tipolzquierdo = pilaTipos.pop();
            operador = popper.pop()
            tipoResultado =
semantic_cube[tipolzquierdo][operador][tipoDerecho]
            current_type = tipoResultado;

```

```
if(tipoResultado != None):
```

```

    #result avail.next()
    temporal = 't' + str(currTemp)
    temp_op = Operando();
    temp_op.id=temporal;
    temp_op.address = get_next_var_address(p);

    tempQuad = [operador, oplzquierdo.id,
opDerecho.id, temp_op.id]
    currTemp = currTemp + 1;
    pilaOp.push(temp_op)
    pilaTipos.push(tipoResultado)
    read_quadruples.append(tempQuad)
    tempQuad = [operador, oplzquierdo.address,
opDerecho.address, temp_op.address]
    dir_quadruples.append(tempQuad);
    else:
        error(p, "Tipo no compatible para la operacion
de mult/div")

```

```
def p_n_Operador(p):
```

```

'''
    n_Operador :
'''
    global popper, pilaOp, pilaTipos
    sim = p[-1]
    popper.push(sim)

#Nivel mas bajo de expresion con constantes, variables,
llamadas a funciones y paréntesis,
def p_f(p):
'''
    f : LPAREN n_start_FF mult_exp RPAREN n_end_FF
        | n_tempTypeI MINUS CTE_I n_directPrint_neg
        | n_tempTypeF MINUS CTE_F n_directPrint_neg
        | n_tempTypeI CTE_I n_directPrint
        | n_tempTypeF CTE_F n_directPrint
        | n_tempTypeC CTE_C n_directPrint
        | variable
        | llamada
'''

```

```

#Funcion auxiliar para tipo de variable leida en cada
caso.

```

```
def p_n_tempTypeI(p):
```

```

'''
    n_tempTypeI :
'''
    global current_type
    current_type = "int"

```

```
def p_n_tempTypeF(p):
```

```

'''
    n_tempTypeF :
'''
    global current_type
    current_type = "float"

```

```
def p_n_tempTypeC(p):
```

```

'''
n_tempTypeC :
'''
global current_type
current_type = "char"

#Funcion auxiliar para manejo de constantes negativas
y evitar error.
def p_n_directPrint_neg(p):
'''
n_directPrint_neg :
'''
global current_exp, pilaOp, pilaTipos, current_type
current_exp = p[-1]
operador = Operando();
operador.id = '-' + current_exp;
operador.address = get_next_cte_address(p,
operador.id);
pilaOp.push(operador)
pilaTipos.push(current_type)

#Auxliar para constantes a expresiones y que se
puedan usar en operaciones.
def p_n_directPrint(p):
'''
n_directPrint :
'''
global current_exp, pilaOp, pilaTipos
current_exp = p[-1]
operador = Operando();
operador.id = current_exp;
operador.address = get_next_cte_address(p,
operador.id);
pilaOp.push(operador)
pilaTipos.push(current_type)

#Inicio y termino del fondo falso usado en operaciones
y arreglos.
def p_n_start_FF(p):
'''
n_start_FF :
'''
global popper, pilaOp, pilaTipos
popper.push(p[-1])

def p_n_end_FF(p):
'''
n_end_FF :
'''
global popper, pilaOp, pilaTipos
aux = popper.peek();
if aux == '(' or aux == '[':
    popper.pop();
else:
    error(p, "Error de paréntesis inconsistentes")

#Conjunto de funciones para manejo de If y ELSE
def p_condicion(p):
'''
condicion : SI LPAREN mult_exp RPAREN n_ifQuad
ENTONCES bloque n_endIfQuad
| SI LPAREN mult_exp RPAREN n_ifQuad
ENTONCES bloque SINO p_n_sinoQuad bloque
n_endIfQuad
'''

#Genera cuadruplo de gotoFalse
def p_n_ifQuad(p):
'''
n_ifQuad :

```

```

'''
global popper, pilaOp, pilaTipos, read_quadruples,
dir_quadruples
expType = pilaTipos.pop();
if(expType != 'bool'):
    error(p, 'La expresion condicional no es un
booleano')
else:
    result = pilaOp.pop()
    quad = ['GotoF', result.id, -1, 'empty']
    read_quadruples.append(quad)
    quad = ['GotoF', result.address, -1, 'empty']
    dir_quadruples.append(quad)
    pilaSaltos.push(len(read_quadruples)-1)

#Vacía pila de saltos y rellena espacios dejados vacios
para manejo de saltos.
def p_n_endIfQuad(p):
'''
n_endIfQuad :
'''
global pilaSaltos, read_quadruples, dir_quadruples
end = pilaSaltos.pop()
read_quadruples[end][3] = len(read_quadruples)
dir_quadruples[end][3] = len(dir_quadruples)

#Cuadruplo que maneja el else de la condicion.
def p_n_sinoQuad(p):
'''
p_n_sinoQuad :
'''
global pilaSaltos, read_quadruples, dir_quadruples
falso = pilaSaltos.pop()
quad = ['Goto', -1, -1, 'empty']
read_quadruples.append(quad)
dir_quadruples.append(quad)
pilaSaltos.push(len(read_quadruples)-1)
read_quadruples[falso][3] = len(read_quadruples)
dir_quadruples[falso][3] = len(dir_quadruples)

#conjunto de funciones que manejan el ciclo while.
def p_ciclo_w(p):
'''
ciclo_w : MIENTRAS n_startCicle LPAREN mult_exp
RPAREN n_evalExp HAZ bloque n_endWhile
'''

def p_n_startCicle(p):
'''
n_startCicle :
'''
global pilaSaltos
pilaSaltos.push(len(read_quadruples))

#Maneja específicamente el generar el cuadruplo
gotoFalse.
def p_n_evalExp(p):
'''
n_evalExp :
'''
global pilaTipos, pilaOp, read_quadruples,
dir_quadruples
type = pilaTipos.pop()
if(type != 'bool'):
    error(p, 'La expresion del mientras no es
booleana');
else:
    result = pilaOp.pop();
    quad = ['GotoF', result.id, -1, 'empty'];
    read_quadruples.append(quad)

```

```

quad = ['GotoF', result.address, -1, 'empty'];
dir_quadruples.append(quad)
pilaSaltos.push(len(read_quadruples)-1)

#Termina el while, genera cuadruplo Goto y rellena
cuadruplos de manejo de saltos.
def p_n_endWhile(p):
    """
    n_endWhile :
    """
    global pilaSaltos, read_quadruples, dir_quadruples
    end = pilaSaltos.pop()
    ret = pilaSaltos.pop()
    quad = ['Goto', -1, -1, ret]
    read_quadruples.append(quad)
    read_quadruples[end][3] = len(read_quadruples)

    dir_quadruples.append(quad)
    dir_quadruples[end][3] = len(dir_quadruples)

#Genera operacion del ciclo for. similar a la operacion
anterior.
def p_ciclo_f(p):
    """
    ciclo_f : DESDE asigna n_startCicle HASTA mult_exp
    n_evalExp_for HACER bloque n_endFor
    """

#pilatipos pilaOp
def p_n_evalExp_for(p):
    """
    n_evalExp_for :
    """
    global pilaTipos, pilaOp, read_quadruples,
    semantic_cube, currTemp, pilaSaltos, dir_quadruples,
    current_type
    typeExp = pilaTipos.pop()
    typeAsig = pilaTipos.pop();
    if(semantic_cube[typeAsig]['<'][typeExp] != 'bool'):
        error(p, 'La expresion del for no es compatible');
    else:
        current_type = 'bool'
        opExp = pilaOp.pop();
        opAsig = pilaOp.pop();
        temp = 't' + str(currTemp)
        temp_op = Operando();
        temp_op.id = temp;
        temp_op.address = get_next_var_address(p);

        quad = ['<', opAsig.id, opExp.id, temp_op.id]
        read_quadruples.append(quad)
        quad = ['<', opAsig.address, opExp.address,
temp_op.address]
        dir_quadruples.append(quad);

        quad = ['GotoF', temp_op.id, -1, 'empty'];
        currTemp = currTemp + 1
        read_quadruples.append(quad)
        quad = ['GotoF', temp_op.address, -1, 'empty'];
        dir_quadruples.append(quad)
        pilaSaltos.push(len(read_quadruples)-1)

#Termina el ciclo for.
def p_n_endFor(p):
    """
    n_endFor :
    """
    global pilaSaltos, read_quadruples
    end = pilaSaltos.pop()
    ret = pilaSaltos.pop()

```

```

quad = ['Goto', -1, -1, ret]
read_quadruples.append(quad)
read_quadruples[end][3] = len(read_quadruples)
dir_quadruples.append(quad)
dir_quadruples[end][3] = len(dir_quadruples)

#Conjunto de funciones que manejan el return de una
función.
def p_retorno(p):
    """
    retorno : REGRESA LPAREN mult_exp n_regresaExp
    RPAREN SEMICOLON
    """

#Revisa que se este regresando el mismo tipo que la
definición de la función y regresa memoria.
def p_n_regresaExp(p):
    """
    n_regresaExp :
    """
    global dir_func, read_quadruples, current_exp,
    current_type, current_func, dir_quadruples, pilaTipos
    operando = pilaOp.pop();
    if(pilaTipos.pop() == dir_func[current_func]['type']):
        temp_quad = ['RETURN', -1, -1, operando.id]
        read_quadruples.append(temp_quad)
        temp_quad = ['RETURN', -1, -1, operando.address]
        dir_quadruples.append(temp_quad)
    else:
        error(p, "La función está regresando un tipo distinto
al de la función")

##error function for parser
def p_error(p):
    global success
    success = False
    print('SyntaxError', p.value)
    print("at line ", p.lineno)
    #añadir en que linea
    sys.exit()

#p.lexer.skip(1)

# TODO:parar la compilación

def error(p, message):
    print("Error: ", message)
    sys.exit()

parser = yacc.yacc()

# nombre archivo, nombre programa
if len(sys.argv) != 2:
    print("Please send a file.")
    raise SyntaxError('Compiler needs 1 file.')

program_name = sys.argv[1]
# Compile program.
with open(program_name, 'r') as file:
    try:
        parser.parse(file.read())
        export = {
            'tabla_func': dir_func,
            'dir_quadruples': dir_quadruples
        }
        with open(program_name + '.obj', 'w') as file1:
            file1.write(str(export))
    except:

```



```
    pass

if success == True:
    print("El archivo se ha aceptado")
    sys.exit()
else:
    print("El archivo tiene errores")
    sys.exit()
```

Implementación de máquina virtual.

```

from stack import Stack
from memoria import Memoria
import sys
import six

#Inicializacion de directorio de funciones, los
cuadрупlos a leer
dir_func = {}
dir_quadрупles = {}

#Stack para saber desde donde se llama una funcion y
de los niveles de memoria.
llamadas = Stack()
memorias = Stack()

#Memoria global
memoria_global = Memoria()

#Stack usada para manejo de funciones
this_func = Stack()
param_stack = Stack()

#memory ranges
#global 0 - 9999
top_limit_global = 10000
#local 10000- 19999
top_limit_local = 20000
#constants 20000- 29999
top_limit_cte = 30000
#pointers 30000-+++

#Funcion para modificar el valor de una direccion de
memoria.
def set_value(address, value):
    if address < top_limit_global:
        memoria_global.set_value(address, value)
    elif address < top_limit_local:
        memorias.peek().set_value(address, value)
    elif address < top_limit_cte:
        memoria_global.set_value(address, value)
    else:
        #in case of a pointer
        #if(memoria_global.isDeclared(address)):
        # memoria_global.set_value(get_value(address),
value)
        #else:
        memoria_global.set_value(address, value)

#Funcion para obtener el valor de una direccion de
memoria.
def get_value(address):
    if address < top_limit_global:
        return memoria_global.get_value(address)
    elif address < top_limit_local:
        return memorias.peek().get_value(address)
    elif address < top_limit_cte:
        return memoria_global.get_value(address)
    else:
        #in case of a pointer
        #content_pointer =
memoria_global.get_value(address)
        #if(memoria_global.isDeclared(get_value(address))
or memorias.peek().isDeclared(get_value(address))):
        # return
memoria_global.get_value(get_value(address))

```

```

#else:
return memoria_global.get_value(address)

#Funcion que guarda en memoria global las constantes
desde el parser.
def save_ctes():
    global memoria_global
    for i in dir_func['constants']['cte']:

memoria_global.set_value(dir_func['constants']['cte'][i]['
address'], i)

#Funcion para obtener el parametro por su tipo para
operaciones de maquina virtual.
def getParam(address):
    if(address >= 30000):
        return getParam(get_value(address))
    if address%10000 < 2500:
        return int(get_value(address))
    elif address%10000 < 5000:
        return float(get_value(address))
    elif address%10000 < 7500:
        return get_value(address)
    else:
        return bool(get_value(address))

#Funcon general de calculos de valores.
def calculate(p1, op, p2):
    if(op == '+'):
        return p1 + p2
    elif(op == '-'):
        return p1 - p2
    elif(op == '*'):
        return p1 * p2
    elif(op == '/'):
        return p1 / p2
    elif(op == '<'):
        return p1 < p2
    elif(op == '<='):
        return p1 <= p2
    elif(op == '>'):
        return p1 > p2
    elif(op == '>='):
        return p1 >= p2
    elif(op == '=='):
        return p1 == p2
    elif(op == '<>'):
        return p1 != p2
    elif(op == '&'):
        return p1 and p2
    elif(op == '|'):
        return p1 or p2

#Funcion general de movimiento entre cuádruplos.
def run():
    global memoria_global, param_stack, memorias,
llamadas, pila_returns, this_func
    #añadiendo ctes y apuntadores a memoria.
    save_ctes();
    tracker = 0
    while tracker < len(dir_quadрупles):
        #print("contador " + str(tracker))
        curr_quad = dir_quadрупles[tracker]
        instr = curr_quad[0]
        el2 = curr_quad[1]
        el3 = curr_quad[2]
        el4 = curr_quad[3]

        if instr == 'Goto':
            tracker = el4

```

```

elif instr == 'print':
    if(el4 >= top_limit_cte):
        aux = get_value(el4)
    else:
        aux = el4
    print(get_value(aux))
    tracker += 1

elif instr == 'read':
    # agregar el leer variable
    aux = input()
    if(el4 >= top_limit_cte):
        aux1 = get_value(el4)
    else:
        aux1 = el4

    set_value(aux1, aux)
    tracker+=1

elif instr == '+' or instr == '-' or instr == '*' or instr ==
'/' or instr == '<' or instr == '<=' or instr == '>' or instr ==
'>=' or instr == '==' or instr == '<>' or instr == '&' or instr
== '|':
    if(el2 >= top_limit_cte):
        aux1 = get_value(el2)
    else:
        aux1 = el2
    if(el3 >= top_limit_cte):
        aux2 = get_value(el3)
    else:
        aux2 = el3
    p1 = getParam(aux1)
    p2 = getParam(aux2)
    value = calculate(p1, instr, p2)
    set_value(el4, value)
    tracker += 1

elif instr == '=':
    if(el2 >= top_limit_cte):
        aux1 = get_value(el2)
    else:
        aux1 = el2
    aux1 = get_value(aux1)

    if(el4 >= top_limit_cte):
        aux = get_value(el4)
    else:
        aux = el4
    set_value(aux, aux1)
    tracker += 1

elif instr == 'ENDFUNC':
    #liberar memoria actual
    #recuperar dir ret
    #regresar a la mem anterior
    if(not memorias.isEmpty()):
        memorias.pop()
    tracker = llamadas.pop()

elif instr == 'GotoF':
    evaluar = getParam(el2)
    if(evaluar == False):
        tracker = el4

else:
    tracker += 1

elif instr == 'ERA':
    #Allocates memory
    #aux = Memoria()
    #memorias.push(aux)
    tracker += 1

elif instr == 'PARAM':
    #aux = Memoria()
    #aux = memorias.pop()
    val = get_value(el2)
    #memorias.push(aux)
    #set_value(el4, val)
    param_stack.push(val)
    param_stack.push(el4)
    tracker += 1

elif instr == 'GOSUB':
    aux = Memoria()
    memorias.push(aux)
    #if(not param_stack.isEmpty()):
    this_func.push(el2)
    while(not param_stack.isEmpty()):
        set_value(param_stack.pop(),
param_stack.pop())
    llamadas.push(tracker+1)
    tracker = el4

elif instr == 'RETURN':
    aux_val = get_value(el4)
    memorias.pop()
    tracker = llamadas.pop()
    if isinstance(this_func.peek(), str):
        this_func.pop()
    set_value(this_func.pop(), aux_val)

elif instr == 'VERIFY':
    if getParam(el2) < getParam(el4):
        tracker += 1
    else:
        print('Error en el acceso a un arreglo, fuera de
dimension')
        sys.exit()

else:
    #lets hope this does not get called
    print('error en cuadruplos')
    sys.exit()

if len(sys.argv) != 2:
    print('Please send a file.')
    raise SyntaxError('vm.')

program_name = sys.argv[1]
# Compile program.
with open(program_name, 'r') as file:
    #global dir_func, dir_quadruples
    filename = eval(file.read())
    dir_func = filename['tabla_func']
    dir_quadruples = filename['dir_quadruples']
    run()

```

FIN DE LA DOCUMENTACIÓN