Instituto Politécnico Nacional
Escuela Superior de Cómputo

“Manual técnico mini LOGO”

Materia: Compiladores

Profesor: Tecla Parra Roberto

Integrantes:

Córdova Pérez Christian

Páramo Romero Luis Antonio

Popoca Martinez Zuriel

Grupo:3CM8

Guerra Vargas Irving Cristobal

Grupo:3CM7

Índice

Gramática

En esta sección, se muestra la gramática utilizada en la realización del proyecto en el archivo “logoyacc.”

list -> list'\n' | list linea '\n'
linea -> exp ';' | stmt | linea exp ';' | linea stmt
exp -> VAR | '-' exp | NUMBER | VAR '=' exp | exp '' exp | exp '+' exp | exp '-' exp | '(' exp ')'*
exp -> exp COMP exp | exp DIFERENTES exp | exp MEN exp | exp MENI exp | exp MAY exp
exp -> exp MAYI exp | exp AND exp | exp OR exp | '!' exp | RETURN exp | PARAMETRO
exp -> nombreProc '(' arglist ')'
arglist -> exp | arglist ',' expstmt -> if '(' exp stop ')' '{' linea stop '}' ELSE '{' linea stop'}'
stmt -> if '(' exp stop ')' '{' linea stop '}' nop stop | while '(' exp stop ')' '{' linea stop '}' stop
stmt -> for '(' instrucciones stop ';' exp stop ';' instrucciones stop ')' '{' linea stop '}' stop
stmt -> funcion nombreProc '(' ')' '{' linea null '}' | procedimiento nombreProc '(' ')' '{' linea null '}'
stmt -> instruccion '[' arglist ']' ';'
instruccion -> FNCT
procedimiento -> PROC
funcion -> FUNC
nombreProc -> VARif -> IF
while -> WHILE
for -> FOR
instrucciones -> exp | instrucciones ',' exp

Mapa de memoria

A continuación, se muestran los distintos mapas de memoria que representan ciertas partes del proyecto.

If-else

IF_ELSE
stop
stop
stop
~~exp~~

stop
~~stmt~~
stop
~~stmt~~
stop

If

IF
stop
stop
stop
~~exp~~
stop
~~stmt~~
stop
nop
stop

While

WHILE
stop
stop
~~exp~~
stop
~~stmt~~
stop
stop

For

FOR
stop
stop
stop
stop
~~exp~~
stop
~~exp~~
stop
~~exp~~
stop
~~stmt~~
stop
stop

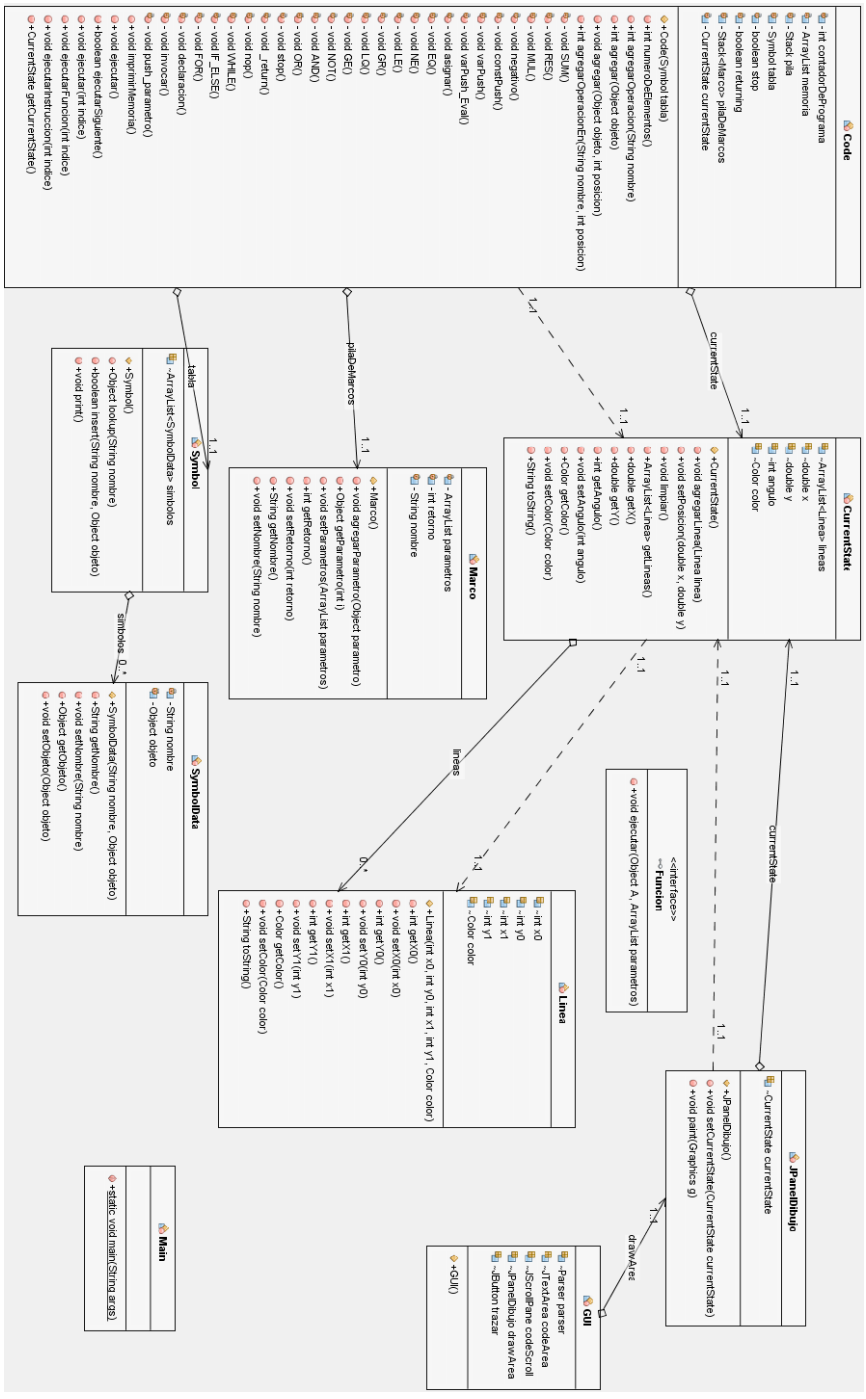
Función

declaracion
~~stmt~~

Procedimiento

declaracion
~~stmt~~

Diagrama de clases



Explicación del código

El proyecto es como un LOGO, que se utiliza para graficar funciones o crear dibujos, dándole a una flecha las instrucciones que queremos que siga para dibujar lo que deseamos.
El programa usa una máquina virtual de pila para simular la parte de generación de código y ejecución de código. A su vez utiliza una pila para guardar los datos con los que va a trabajar, utiliza una tabla de símbolos para que sea capaz de almacenar funciones, procedimientos, comandos y variables.
La parte de generación de código se observa en la figura 1.1.

```java
public int addOperation(String name){
    int posicion = memory.size();
    try{
        memory.add(this.getClass().getDeclaredMethod(name, null));
        return posicion;
    }
    catch(Exception e ){
        System.out.println("[ ERROR ] Error al agregar la operación " + name);
    }
    return -1;
}

public int add(Object object){
    int posicion = memory.size();
    memory.add(object);
    return posicion;
}

public void add(Object object, int pos){
    memory.remove(pos);
    memory.add(pos, object);
}

public int addOperationIn(String name, int pos){
    try{
        memory.add(pos, this.getClass().getDeclaredMethod(name, null));
    }
    catch(Exception e ){
        System.out.println("[ ERROR ] Error al agregar la operación " + name);
    }
    return pos;
}
```

Figura 1.1: Generación de código

Las funciones anteriores se utilizan para agregar métodos u objetos a la memoria en la posición correspondiente, ya sea en el último lugar de la pila o en una posición específica. La parte de ejecución está en la figura 1.2.

```
public void ejecutar(){
    //imprimirMemoria();
    stop = false;
    while(programCounter < memory.size())
        ejecutarInstruccion(programCounter);
}

public boolean ejecutarSiguiente(){
    //imprimirMemoria();
    if(programCounter < memory.size()){
        ejecutarInstruccion(programCounter);
        return true;
    }
    return false;
}

public void ejecutar(int indice){
    programCounter = indice;
    while(!stop && !returning){
        ejecutarInstruccion(programCounter);
    }
    stop = false;
}

public void ejecutarFuncion(int indice){
    programCounter = indice;
    while(!returning && memory.get(programCounter) != null){
        ejecutarInstruccion(programCounter);
    }
    returning = false;
    programCounter = stackMacros.lastElement().getRetorno();
    stackMacros.removeElement(stackMacros.lastElement());
}
```

Figura 1.2: Ejecución de código

```java
public void ejecutarInstruccion(int indice){
    try{
        Object objetoLeido = memory.get(indice);
        if(objetoLeido instanceof Method){
            Method metodo = (Method)objetoLeido;
            metodo.invoke(this, null);
        }
        if(objetoLeido instanceof Funcion){
            ArrayList parametros = new ArrayList();
            Funcion funcion = (Funcion)objetoLeido;
            programCounter++;
            while(memory.get(programCounter) != null){
                if(memory.get(programCounter) instanceof String){
                    if(((String)(memory.get(programCounter))).equals("Limite")){
                        Object parametro = stack.pop();
                        parametros.add(parametro);
                        programCounter++;
                    }
                }
                else{
                    ejecutarInstruccion(programCounter);
                }

            }
            funcion.ejecutar(currentState, parametros);
        }
        programCounter++;
    }
    catch(Exception e){}
}
```

Figura 1.3: Ejecución de instrucciones

En la figura 1.3, se observa la función para ejecutar instrucciones, ya se directamente o mediante una función que ejecuta varias instrucciones, y al ejecutarlas incremente en uno el programCounter para ir a la siguiente posición de la memoria y seguir ejecutando las siguientes instrucciones.

```java
public class SymbolData {

    private String nombre;
    private Object objeto;

    public SymbolData(String nombre, Object objeto){
        this.nombre = nombre;
        this.objeto = objeto;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Object getObjeto() {
        return objeto;
    }

    public void setObjeto(Object objeto) {
        this.objeto = objeto;
    }

}
```

Figura 1.4: Clase SymbolData

Lo figura 1.4 es la definición de la clase SymbolData que nos servirá para poder almacenar nuestras variables o el nombre de los procesos y funciones que declaremos en momento de ejecución.

```java
public class Symbol {

    ArrayList<SymbolData> simbolos;

    public Symbol(){
        simbolos = new ArrayList<SymbolData>();
    }

    public Object lookup(String nombre){
        for(int i = 0; i < simbolos.size(); i++)
            if(nombre.equals(simbolos.get(i).getNombre()))
                return simbolos.get(i).getObjeto();
        return null;
    }

    public boolean insert(String nombre, Object objeto){
        SymbolData par = new SymbolData(nombre, objeto);
        for(int i = 0; i < simbolos.size(); i++)
            if(nombre.equals(simbolos.get(i).getNombre())){
                simbolos.get(i).setObjeto(objeto);
                return true;
            }
        simbolos.add(par);
        return false;
    }

    public void print(){
        for(int i = 0; i < simbolos.size(); i++){
            System.out.println(simbolos.get(i).getNombre() + simbolos.get(i).getObjeto().toString());
        }
    }

}
```

Figura 1.5: Búsqueda de símbolos

Finalmente, la clase mostrada en la figura 1.5, es propiamente las operaciones para buscar e insertar nuestros objetos de la clase SymbolData en la tabla de símbolos, que es Symbol, que es donde se guardarán los nombres de las variables, procesos y funciones declaradas en momento de ejecución.