INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

COMPILADORES

MANUAL TÉCNICO: MINI-LOGO

GRUPO: 3CM7

IBARRA SOTO ALEJANDRO
MENDOZA SÁNCHEZ MARCO ANTONIO

PROFESOR: TECLA PARRA ROBERTO

FECHA: 25/06/2020

Manual técnico: Mini – Logo

La gramática implementada para la implementación de un intérprete para un lenguaje imperativo de un subconjunto del lenguaje logo, es la siguiente, la cual se basó en el uso de hoc6, es decir se realizó la implementación de la siguiente gramática:

Estas primeras producciones permiten generar el código que se introducirá, las producciones de **oper** permiten poder llamar a las producciones para definir funciones y procedimientos, ciclos y decisiones y operaciones de elementos de tipo double; **asgn** permite realizar asignaciones a variables y argumentos.

```
%%
list:
    | list '\n'
    | list oper '\n'
    ;
oper :defn ';' { maq.code("STOP");return 1 ;}
     | stmt  { maq.code("STOP"); return 1 ;}
     | expr ';' { maq.code("print");
                  maq.code("STOP"); return 1 ;}
  ;
asgn: VAR '=' expr {
          $$=$3; maq.code("varpush");
                 maq.code(((Algo)$1.obj).simb);
      maq.code("assign");
  }
     |  ARG '=' expr
      {
        defnonly("$");
        //code2(argassign,(Inst)$1);
        maq.code("argassign");
        maq.code($1.ival+"");
        $$=$3;
      }
     ;
```

Las producciones de **stmt** permiten generar llamadas a procedimientos, las sentencias de return, las sentencias de ciclos y decisiones,

```
stmt:     expr ';'  {
                maq.code("pop");}
   | RETURN ';' { defnonly("return"); maq.code("procret"); }
   | RETURN expr ';'
         { defnonly( "return" ); $$ = $2; maq.code("funcret") ; }
   | PROCEDURE begin '(' arglist ')'
         { $$ = $2;
           maq.code("call");
           maq.code(((Algo)$1.obj).simb);
           maq.code($4.ival+"");
    }  ';'
   | PRINT prlist ';' { $$ = $2; }
   | for '(' forexp ';' condFor ';' forexp ')' stmt end {
     maq.getProg().setElementAt(
                new Integer(((Algo)$5.obj).inst),
                ((Algo)$1.obj).inst+1);
     maq.getProg().setElementAt(
                new Integer(((Algo)$7.obj).inst),
                ((Algo)$1.obj).inst+2);
```

```
      maq.getProg().setElementAt(
                  new Integer(((Algo)$9.obj).inst),
                  ((Algo)$1.obj).inst+3);
      maq.getProg().setElementAt(
                  new Integer(((Algo)$10.obj).inst),
                  ((Algo)$1.obj).inst+4);
   }
  | while cond stmt end {
          maq.getProg().setElementAt(
              new Integer(((Algo)$3.obj).inst),
              ((Algo)$1.obj).inst+1);
          maq.getProg().setElementAt(
              new Integer(((Algo)$4.obj).inst),
              ((Algo)$1.obj).inst+2);
   }
     | if cond stmt end {
              maq.getProg().setElementAt(
              new Integer(((Algo)$3.obj).inst),
              ((Algo)$1.obj).inst+1);
              maq.getProg().setElementAt(
              new Integer(((Algo)$4.obj).inst),
              ((Algo)$1.obj).inst+3);
               }
| if cond stmt end ELSE stmt end {
              maq.getProg().setElementAt(
              new Integer(((Algo)$3.obj).inst),
              ((Algo)$1.obj).inst+1);
              maq.getProg().setElementAt(
              new Integer(((Algo)$6.obj).inst),
              ((Algo)$1.obj).inst+2);
               maq.getProg().setElementAt(
              new Integer(((Algo)$7.obj).inst),
             ((Algo)$1.obj).inst+3);
  }
|   '{'  stmtlist  '}' {
              $$  =  $2;
              }
      ;
```

Las siguientes producciones permiten generar el mapa de memoria de los diferentes ciclos y condiciones que se implementaron.

```
while:  WHILE   {
  $$ = new ParserVal(new Algo(maq.code("whilecode")));
       maq.code("STOP");maq.code("STOP");
       }
  ;
for:  FOR  {
  $$ = new ParserVal(new Algo(maq.code("forcode")));
       maq.code("STOP");maq.code("STOP");maq.code("STOP");maq.code("STOP");
       }
  ;
if:IF   { //$$ = code(ifcode); code3(STOP,STOP,STOP);
  $$ = new ParserVal(new Algo(maq.code("ifcode")));
       maq.code("STOP");maq.code("STOP");maq.code("STOP");
       }
  ;
begin:  /* nada */          {
       $$=new ParserVal(new Algo(maq.getProgP()));
       //$$ = new ParserVal(maq.getProgP());
       }
   ;

condFor :  iniFor expr   {   maq.code("STOP");
        ((Algo)$$.obj).inst = ((Algo)$1.obj).inst;//checar
    }
```

Las producciones de **expr** generan principalmente las operaciones que se pueden realizar entre elementos de tipo doble, sin embargo, estas también permiten generar las instrucciones básicas del interprete del lenguaje Logo

```
expr:     NUMBER  { ((Algo)$$.obj).inst=maq.code("constpush");
                    maq.code(((Algo)$1.obj).simb); }
        | VAR { ((Algo)$$.obj).inst=maq.code("varpush");
                    maq.code(((Algo)$1.obj).simb); maq.code("eval");}
        | ARG    {
          defnonly("$");
          //$$ = code2(arg, (Inst)$1);
          //erroneo ((Algo)$$.obj).inst= maq.code("arg");
          $$ = new ParserVal(new Algo(maq.code("arg")));
          maq.code($1.ival+"");
          }
        | asgn
        | FORWARD expr { $$=$2; maq.code("forward");}
        | UP expr { $$=$2; maq.code("upPencil");}
        | DOWN expr { $$=$2; maq.code("downPencil");}
        | GD expr { $$=$2; maq.code("girarDerecha");}
        | GI expr { $$=$2; maq.code("girarIzquierda");}
        | COLOR '[' expr ',' expr ',' expr ']' { $$=$2; maq.code("color");}
        | FUNCTION begin '(' arglist ')'
          { $$ = $2;
            maq.code("call");
            maq.code(((Algo)$1.obj).simb);
            maq.code($4.ival+"");
        }
        | BLTIN '(' expr ')'{ $$=$3; maq.code("bltin"); maq.code(((Algo)$1.obj).simb); }
        | expr '+' expr { maq.code("add");}
        | expr '-' expr { maq.code("sub"); }
        | expr '*' expr { maq.code("mul");}
        | expr '/' expr { maq.code("div"); }
        | expr '^' expr { maq.code("power"); }
```

Además de las instrucciones para interpretar un subconjunto del lenguaje Logo, también incluye la llamada o invocación de las funciones pasándole una lista de argumentos, e igual implementa los operadores relacionales para poder realizar la comparación entre valores.

```
        | '(' expr ')'  { $$= $2;}
        | '-' expr %prec UNARYMINUS { $$=$2; maq.code("negate");}
        |expr EQ expr  { maq.code("eq"); }
        |expr NE expr  { maq.code("ne"); }
        |expr GT expr  { maq.code("gt"); }
        |expr GE expr  { maq.code("ge"); }
  |expr LT expr  { maq.code("lt"); }
        |expr LE expr  { maq.code("le"); }
  |expr AND expr { maq.code("and"); }
  |expr OR expr  { maq.code("or"); }
  |NOT expr      { $$ = $2; maq.code("not"); }
  ;
```

Las producciones de **defn** se encargan de definir las funciones o procedimientos que se quieran implementar.

```
defn:    FUNC procname { ((Algo)$2.obj).simb.tipo=FUNCTION; indef=true; }
'(' ')' stmt { maq.code("procret"); maq.define(((Algo)$2.obj).simb); indef=false; }
   | PROC procname { ((Algo)$2.obj).simb.tipo=PROCEDURE; indef=true; }
'(' ')' stmt { maq.code("procret"); maq.define(((Algo)$2.obj).simb); indef=false; }
   ;
```

Finalmente, las producciones de **procname** permite guardar el nombre de variables, procedimientos y funciones; **arglist** permite generar una lista de los argumentos que se le pasan a una función cuando se le invoca.

```
procname: VAR
    | FUNCTION
    | PROCEDURE
    ;
arglist:  /* nada */   { $$ = new ParserVal(0); }
    | expr                 { $$ = new ParserVal(1); }
    | arglist ',' expr     { $$ =  new ParserVal($1.ival + 1); }
    ;
```

La gramática, se ejecuta con el comando **byacc -J logo.y**, lo que da con resultado un archivo Parser.java, el cual entre otras cosas, se encarga de analizar la cadena introducida en el área de código, pero antes se realiza la inicialización de la tabla de símbolos que se implementó.

```
maq=new Maquina();
 tabla=new Tabla();
 tabla.install("sin",BLTIN, 0.0);
 tabla.install("cos",BLTIN, 0.0);
 tabla.install("tan",BLTIN, 0.0);
 tabla.install("asin",BLTIN, 0.0);
 tabla.install("acos",BLTIN, 0.0);
 tabla.install("atan",BLTIN, 0.0);
 tabla.install("exp",BLTIN, 0.0);
 tabla.install("log",BLTIN, 0.0);
 tabla.install("sqrt",BLTIN, 0.0);
 tabla.install("if", IF, 0.0);
 tabla.install("else", ELSE, 0.0);
 tabla.install("while", WHILE, 0.0);
 tabla.install("print", PRINT, 0.0);
 tabla.install("proc", PROC, 0.0);
 tabla.install("func", FUNC, 0.0);
 tabla.install("return", RETURN, 0.0);
 tabla.install("for", FOR, 0.0);
 tabla.install("avanzar", FORWARD, 0.0);
 tabla.install("subir", UP, 0.0);
 tabla.install("bajar", DOWN, 0.0);
 tabla.install("girarDerecha", GD, 0.0);
 tabla.install("girarIzquierda", GI, 0.0);
 tabla.install("color", COLOR, 0.0);
```

Asimismo, se encarga de la inicialización de la interfaz en la que se estará trabajando y a la que el usuario tendrá acceso, para ello se implementaron las siguientes líneas de código.

```
maq.setTabla(tabla);
jf=new JFrame("Logos");
canv=new Canvas();
//canv.setSize(800,800);
panelEjecuta = new PanelEjecuta(maq, this);
jf.setLayout(new GridLayout (1,2));
jf.add(panelEjecuta);
jf.add(canv);
jf.setSize( 1800, 900);
jf.setVisible(true);
g=canv.getGraphics();
g.setColor(new Color(121,222,228));
g.fillRect(0, 0, 800, 800);
g.setColor(new Color(0,0,0));
maq.setGraphics(g);
panelEjecuta.setCanvas(g);

jf.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

Dentro del archivo PanelEjecuta.java está la generación del área de código y de los botones que se muestran en la interfaz para el usuario.

```
GridBagLayout gridBag = new GridBagLayout ();
GridBagConstraints restricciones = new GridBagConstraints ();
setLayout(gridBag);
restricciones.fill = GridBagConstraints.BOTH;

txtInstrucciones = new JTextArea(8, 40);
txtInstrucciones.setFont(txtInstrucciones.getFont().deriveFont(24f));
JScrollPane qScroller = new JScrollPane(txtInstrucciones);
restricciones.gridwidth = GridBagConstraints.REMAINDER;
restricciones.weighty = 1.5;
restricciones.weightx = 0;
restricciones.gridheight = 10;
gridBag.setConstraints (qScroller, restricciones);
add(qScroller);

paleta=new Paleta(etiqs,new GridLayout(1, etiqs.length), new ManejaAccionInt());
restricciones.weightx = 0.0;
restricciones.gridwidth = 1;
restricciones.gridheight = 1;
restricciones.weighty = 0;
gridBag.setConstraints (paleta, restricciones);
add(paleta);
```

Asimismo, dentro de este archivo se encuentra la implementación de las acciones para los botones, donde el primer botón es el encargado de ejecutar el código que el usuario introduzca, este botón analiza cada sentencia que se introduzca y al final gráfica el cursor en la posición final, el segundo botón permite hacer el borrado tanto del área de texto como del panel de dibujo que se tenga.

```
class ManejaAccionInt implements AccionInt {
  public void accion(int n, ActionEvent e){
        //System.out.println("accion");
    if(n==0){
      par.setTokenizer(new StringTokenizer(txtInstrucciones.getText()));
      par.setNewline(false);
    //g.setColor(getBackground());
      g.setColor(Color.WHITE);
      g.setColor(new Color(121,222,228));
      g.fillRect(0, 0, 900, 900);
      g.setColor(new Color(0,0,0));
      maq.setX(400);
      maq.setY(400);
      maq.setAngulo(0);
      for(maq.initcode(); par.yyparse ()!=0; maq.initcode())
        maq.execute(maq.progbase);
      Polygon poligono = null;
      int xs[] = new int[3];
      int ys[] = new int[3];
      double x = maq.getX();
      double y = maq.getY();;
      double angulo = maq.getAngulo();
      xs[0] = (int) x;
      ys[0] = (int) y;
      xs[1] = (int) (x - 10*Math.cos(Math.toRadians(angulo+20)));
      ys[1] = (int) (y - + 10*Math.sin(Math.toRadians(angulo+20)));
      xs[2] = (int) (x - 10*Math.cos(Math.toRadians(angulo-20)));
      ys[2] = (int) (y - + 10*Math.sin(Math.toRadians(angulo-20)));
      poligono = new Polygon(xs,ys,3);
      g.setColor(Color.BLACK);
      g.drawPolygon(poligono);
```

Se puede ver que, de la misma forma, se encarga de inicializar el cursor en la mitad del panel de dibujo, así como el ángulo en el que esté apunte, es importante mencionar, que cada vez que se da clic en este botón de igual manera se limpia el panel de dibujo. El siguiente es el código del segundo botón.

```
if(n==1){
  txtInstrucciones.setText("");
  g.setColor(Color.WHITE);
  g.setColor(new Color(121,222,228));
  g.fillRect(0, 0, 900, 900);
  g.setColor(new Color(0,0,0));
  maq.setX(400);
  maq.setY(400);
  maq.setAngulo(0);
  Polygon poligono = null;
  int xs[] = new int[3];
  int ys[] = new int[3];
  double x = maq.getX();
  double y = maq.getY();;
  double angulo = maq.getAngulo();
  xs[0] = (int) x;
  ys[0] = (int) y;
  xs[1] = (int) (x - 10*Math.cos(Math.toRadians(angulo+20)));
  ys[1] = (int) (y - + 10*Math.sin(Math.toRadians(angulo+20)));
  xs[2] = (int) (x - 10*Math.cos(Math.toRadians(angulo-20)));
  ys[2] = (int) (y - + 10*Math.sin(Math.toRadians(angulo-20)));
  poligono = new Polygon(xs,ys,3);
  g.setColor(Color.BLACK);
  g.drawPolygon(poligono);
  g.fillPolygon(poligono);
}
```

El archivo Paleta.java, permite la generación de los botones, así como agregar las acciones que realizarán cada uno de los botones.

```
public Paleta(String captions[], LayoutManager lm, AccionInt ai){
    botones=new JButton[captions.length];
    setLayout(lm);
    actor=ai;
    for(int i=0; i<botones.length; i++){
        botones[i]=new JButton(captions[i]);
        add(botones[i]);
        botones[i].addActionListener(new ManejaBotones());
    }
}

int getSeleccion(){
    return sel;
}

class ManejaBotones implements ActionListener {
    public void actionPerformed(ActionEvent e){
        JButton j=(JButton)e.getSource();
        for(int i=0; i<botones.length; i++){
            if(botones[i]==j){
                sel=i;
                                    //cad.Insertar(i);
            }
        }
        actor.accion(sel, e);
    }
}
```

La clase Marco.java tiene la definición de un marco, que posteriormente se colocará en la pila de llamadas a funciones, la implementación de esta permite obtener recursividad, pues aquí se almacenan el símbolo, la dirección a donde se regresará cuando la función a la que corresponde el marco termine, la cantidad de argumentos y el último argumento que se añadió necesario para la función.

```
class Marco {
    Simbolo s;
    int retpc;
    int argn;//n ele pila
    int nargs;
    Marco(Simbolo s, int retpc, int argn, int nargs){
        this.s=s;
        this.retpc=retpc;
        this.argn=argn;
        this.nargs=nargs;
    }
}
```

La clase Linea.java permite implementar el trazado de una línea con base en cuatro puntos que corresponden a la coordenada de inicio y de fin de la línea.

```
public class Linea implements Dibujable {
    private int x1=0;
    private int y1=0;
    private int x2=0;
    private int y2=0;

    public Linea(int x1, int y1, int x2, int y2)
    {
        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
    }
    public void dibuja(Graphics g)
    {
        g.drawLine(x1,y1,x2,y2);
    }
}
```

La clase **Maquina.java** es la encarga del funcionamiento de todo, pues se encarga de llevar a cabo las llamadas a funciones, la ejecución de los ciclos y decisiones, el trazado de las líneas, el girado, el cambio de color y la modificación de la posición del cursor.

La función initcode permite la generación de una nueva pila de datos, pila de llamadas a funciones y permite eliminar todo elemento de la pila de programa que no sea una función o un procedimiento que se haya declarado.

```
void initcode(){
   progp = progbase;
   pila=new Stack();
   marcos=new Stack();
   //prog=new Vector();
   returning = false;
   int index;
   for (index=prog.size()-1;index>=progbase;index--)
      prog.removeElementAt(index);
}
```

La función code permite introducir en la RAM del programa las operaciones que se llevarán a cabo, es decir, se encarga de la generación de código.

```
int code(Object f){
   //System.out.println("Gen ("+f+") size="+prog.size());
   prog.addElement(f);
   progp = prog.size();
   return prog.size()-1;
}
```

La función execute es la encarga de la ejecución del código.

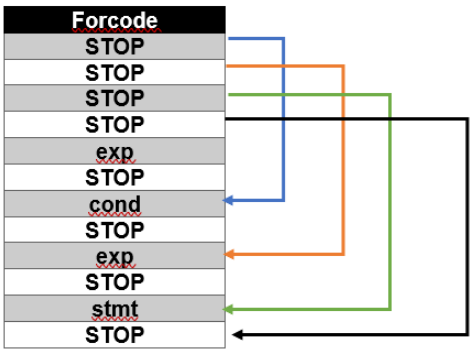
```
void execute(int p){
   String inst;
   for(pc = p; !(inst=(String)prog.elementAt(pc)).equals("STOP") && !returning && pc<prog.size(); ){
      try {
         inst=(String)prog.elementAt(pc);
         pc=pc+1;
         c=this.getClass();
         metodo=c.getDeclaredMethod(inst, null);
         metodo.invoke(this, null);
      } catch(NoSuchMethodException e){
         System.out.println("No metodo "+e);
      } catch(InvocationTargetException e){
         System.out.println(" inst = "+inst+"  "+e+"this "+this+" pc = [ "+pc+" ]");
      } catch(IllegalAccessException e){
         System.out.println(e);
      }
   }
}
```

Las siguientes funciones permiten añadir una constante o una variable a la pila de datos para poder manejarla después, por ejemplo, al obtener el tamaño de la línea que se trazara.

```
void constpush(){
   Simbolo s;
   Double d;
   s=(Simbolo)prog.elementAt(pc);
   pc=pc+1;
   pila.push(new Double(s.val));
}

void varpush(){
   Simbolo s;
   double d;
   s=(Simbolo)prog.elementAt(pc);
   pc=pc+1;
   pila.push(s);
}
```

Las instrucciones de ciclos y decisiones se basan en el análisis del mapa de memoria que se este usando, por ejemplo, el ciclo for presenta el siguiente mapa de memoria.



Por lo que la función dentro de Maquina.java encarga de la ejecución de esta instrucción es la siguiente, en la que se analiza primero si tiene una expresión de inicialización, posteriormente se ejecuta la condición y mientras está se cumpla se está accediendo al cuerpo del for, ejecutando la expresión del tercer argumento del for, y ejecutando y evaluando la condición, para finalmente regresar a donde termina.

```
void forcode(){
   boolean d;
   int savepc = pc;
   if(!prog.elementAt(savepc+4).toString().equals("STOP")){
      execute(savepc+4);    /* condition */
      pila.pop();
   }
   execute(((Integer)prog.elementAt(savepc)).intValue());
   d=((Boolean)pila.pop()).booleanValue();
   while (d) {
      execute(((Integer)prog.elementAt(savepc+2)).intValue());
      if(returning) break;
      if(!prog.elementAt(savepc+1).toString().equals("STOP")){
         execute(
         ((Integer)prog.elementAt(savepc+1)).intValue());
      }
      execute(((Integer)prog.elementAt(savepc)).intValue());
      d=((Boolean)pila.pop()).booleanValue();
   }
   if(!returning)
      pc=((Integer)prog.elementAt(savepc+1)).intValue();
}
```

El ciclo while y la decisión if tienen un comportamiento similar, solo que, con un diferente mapa de memoria, el cual se verifica por las siguientes funciones.

```
void whilecode(){
   boolean d;
   int savepc = pc;
   execute(savepc+2);    /* condition */
   d=((Boolean)pila.pop()).booleanValue();
   while (d) {
      execute(((Integer)prog.elementAt(savepc)).intValue());
      if (returning) break;
      execute(savepc+2);
      d=((Boolean)pila.pop()).booleanValue();
   }
   if(!returning)
      pc=((Integer)prog.elementAt(savepc+1)).intValue();
}
```

```
void ifcode(){
   boolean d;
   int savepc=pc;
   execute(savepc+3);
   d=((Boolean)pila.pop()).booleanValue();
   if(d){
      execute(((Integer)prog.elementAt(savepc)).intValue());
   }
   else if(!prog.elementAt(savepc+1).toString().equals("STOP"))
      execute(((Integer)prog.elementAt(savepc+1)).intValue());
   if (!returning)
      pc=((Integer)prog.elementAt(savepc+2)).intValue();
}
```

Para la definición de funciones se guarda el valor del inicio del código, y se modifica el valor para que en la RAM no se sobrescriba sobre esos datos de las funciones o procedimientos.

```
void define(Simbolo s){
   s.defn=progbase;
   progbase=prog.size();
}
```

Al realizar una llamada a una función se almacenan los datos del marco de la función, esto permite obtener la recursividad, y se ejecuta el código de la función a la que se está invocando.

```
void call(){
   Simbolo s;
   Marco m;
   s=(Simbolo)prog.elementAt(pc);
   m=new Marco(s, pc+2, pila.size()-1, Integer.parseInt((String)prog.elementAt(pc+1)));
   marcos.push(m);
   execute(s.defn);
   returning = false;
}
```

Cuando se termina una función o un procedimiento lo que se hace es sacar el marco de la pila de llamadas a funciones, obtener todos los argumentos de la pila y se activa la bandera **returning** empleada dentro de ciclos y decisiones para un correcto posicionamiento del PC.

```
void ret(){
   Marco m=(Marco)marcos.peek();
   for(int i=0 ; i< m.nargs; i++)
      pila.pop();
   pc=m.retpc;
   marcos.pop();
   returning = true;
}
```

Las siguientes funciones permite retornar después de una función o procedimiento,

```
void funcret(){
   Object o;
   Marco m=(Marco)marcos.peek();
   if(m.s.tipo==PROCEDURE)
      System.out.println(m.s.nombre+" (proc) regresa valor");
   double d;
   d=((Double)pila.pop()).doubleValue();
   ret();
   pila.push(new Double(d));
}

void procret(){
   Marco m=(Marco)marcos.peek();
   if(m.s.tipo == FUNCTION)
      System.out.println(m.s.nombre+" (func) regresa valor");
   ret();
}
```

Dentro de la función o procedimiento se requiere acceder al valor de los argumentos que se están solicitando, para ello se emplean las siguientes funciones.

```
int getarg(){
   Marco m=(Marco)marcos.peek();
   int nargs =Integer.parseInt((String)prog.elementAt(pc));
   pc=pc+1;
   if(nargs > m.nargs)
      System.out.println(m.s.nombre+" argumentos insuficientes");
   return m.argn+nargs-m.nargs;
}

void arg(){
   Object o;
   double d;
   d=((Double)pila.elementAt(getarg())).doubleValue();
   pila.push(new Double(d));
}
```

Para poder implementar el comando **avanzar** del lenguaje logo, lo que se realiza es el trazado de una línea del tamaño que el usuario le indique.

```
void forward(){
   double d1;
   d1=((Double)pila.pop()).doubleValue();
   if(g!=null){
      (new Linea((int)x,(int)y,(int)(x+d1*Math.cos(Math.toRadians(angulo))),
         (int)(y+d1*Math.sin(Math.toRadians(angulo))) ).dibuja(g);
   }
   x=x+d1*Math.cos(Math.toRadians(angulo));
   y=y+d1*Math.sin(Math.toRadians(angulo));
   pila.push(new Double(d1));
}
```

El comando del leguaje Logo **subir** lo que hace es modificar la posición (x,y) en donde se traza, esto considerando el ángulo que tiene, se modifican ambos valores porque, por ejemplo, al tener un ángulo de 45º se sube el lápiz en esa dirección sin trazar nada, la función **downPencil** realiza un procedimiento análogo, solo que en vez de sumar se resta lo que el usuario requiera.

```
void upPencil(){
   double d1;
   d1=((Double)pila.pop()).doubleValue();
   //y = y - (int)d1 ;
   x=x+d1*Math.cos(Math.toRadians(angulo));
   y=y+d1*Math.sin(Math.toRadians(angulo));
   pila.push(new Double(d1));
}
```

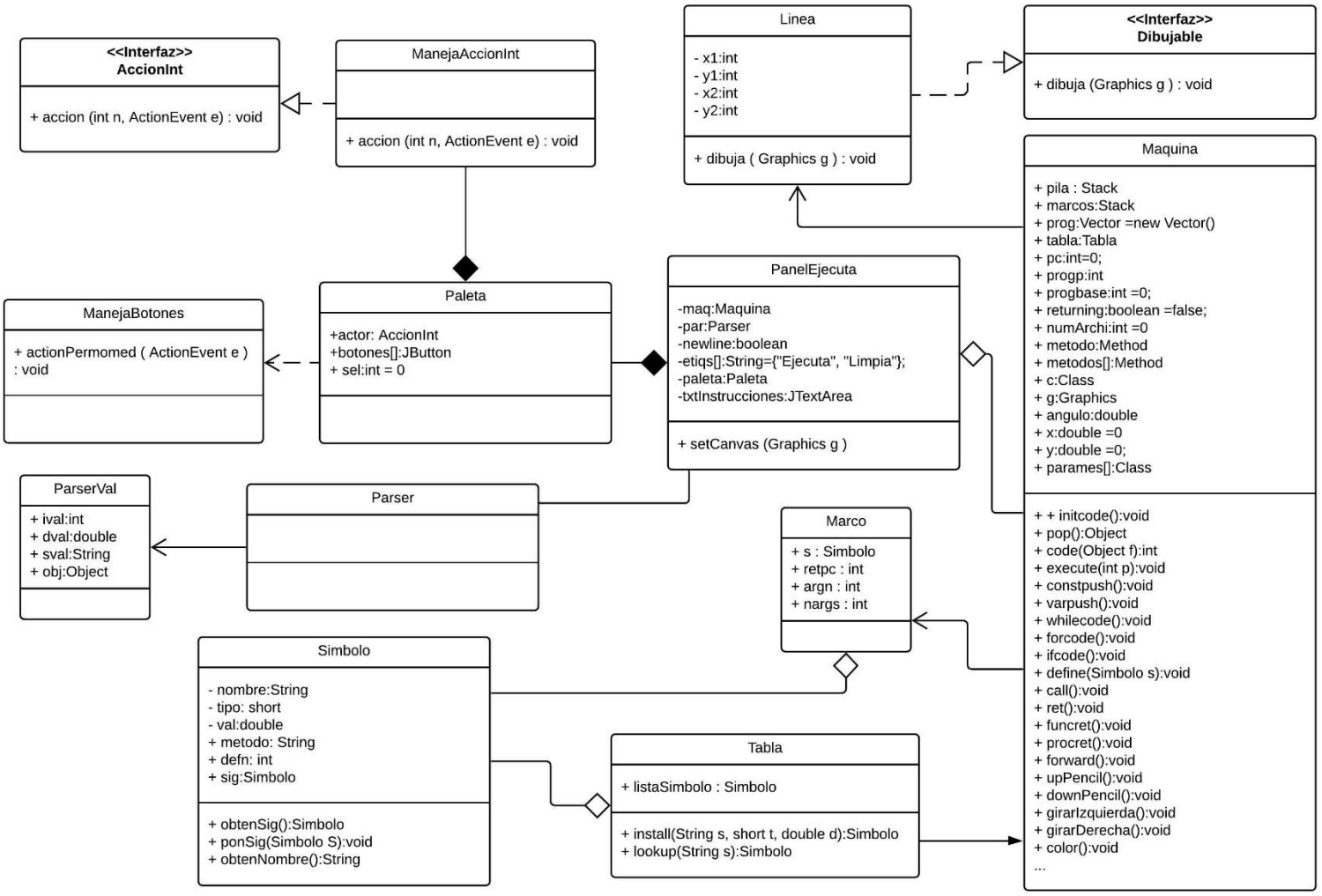
Las funciones de girar únicamente añaden el valor que el usuario proporcione al valor del ángulo que tenga el cursor, si el giro es a la izquierda al valor del cursor se le resta el valor dado por el usuario, porque a la izquierda es en un sentido antihorario.

```
void girarIzquierda(){
   double d1;
   d1=((Double)pila.pop()).doubleValue();
   angulo = angulo - d1 ;
   angulo = angulo%360;
   pila.push(new Double(d1));
}
```

Para cambiar el color con el que se traza se implementó la función color, la cual toma tres valores de la pila, que son los correspondientes a los valores RGB y se establece el color del lienzo como resultado de esos valores.

```
void color(){
   double d1, d2, d3;
   d1=((Double)pila.pop()).doubleValue();
   d2=((Double)pila.pop()).doubleValue();
   d3=((Double)pila.pop()).doubleValue();
   g.setColor(new Color((int)d3%256,(int)d2%256,(int)d1%256));
   pila.push(new Double(d1));
}
```

A continuación, se observa el diagrama de clases para comprender la organización que se tiene de los archivos necesarios para que funcione el programa.


<<Interfaz>>
AccionInt
+ accion (int n, ActionEvent e) : void
ManejaAccionInt
+ accion (int n, ActionEvent e) : void
Linea
- x1:int
- y1:int
- x2:int
- y2:int
+ dibuja (Graphics g) : void
<<Interfaz>>
Dibujable
+ dibuja (Graphics g) : void
Maquina
+ pila : Stack
+ marcos:Stack
+ prog:Vector =new Vector()
+ tabla:Tabla
+ pc:int=0;
+ progp:int
+ progbase:int =0;
+ returning:boolean =false;
+ numArchi:int =0
+ metodo:Method
+ metodos[]:Method
+ c:Class
+ g:Graphics
+ angulo:double
+ x:double =0
+ y:double =0;
+ parames[]:Class
+ + initcode():void
+ pop():Object
+ code(Object f):int
+ execute(int p):void
+ constpush():void
+ varpush():void
+ whilecode():void
+ forcode():void
+ ifcode():void
+ define(Simbolo s):void
+ call():void
+ ret():void
+ funcret():void
+ procret():void
+ forward():void
+ upPencil():void
+ downPencil():void
+ girarIzquierda():void
+ girarDerecha():void
+ color():void
...
ManejaBotones
+ actionPermomed (ActionEvent e) : void
Paleta
+actor: AccionInt
+botones[]:JButton
+ sel:int = 0
PanelEjecuta
-maq:Maquina
-par:Parser
-newline:boolean
-etiqs[]:String={"Ejecuta", "Limpia"};
-paleta:Paleta
-txtInstrucciones:JTextArea
+ setCanvas (Graphics g)
ParserVal
+ ival:int
+ dval:double
+ sval:String
+ obj:Object
Parser
Marco
+ s : Simbolo
+ retpc : int
+ argn : int
+ nargs : int
Simbolo
- nombre:String
- tipo: short
- val:double
+ metodo: String
+ defn: int
+ sig:Simbolo
+ obtenSig():Simbolo
+ ponSig(Simbolo S):void
+ obtenNombre():String
Tabla
+ listaSimbolo : Simbolo
+ install(String s, short t, double d):Simbolo
+ lookup(String s):Simbolo