Documentación Proyecto1

Juan David Cárdenas - 202221834

Alejandro Pardo – 202223709

Declaración de la Gramática:

Primeramente, definimos todos los tokens que íbamos a utilizar dentro de la gramática del lenguaje del robot. Para esto utilizamos la notación BNF dentro de javacc e implementamos tokens de esta forma:

```
TOKEN: /* Nombres de Comandos */
{

        <MOV:  "Move">
      //|     <RIGHT: "RIGHT"> "GENERABA ERRORES"
      |   <PUT: "Put">
      |   <PICK: "Pick">
      |  < POP: "Pop" >
      |  <GO: "GO" >
      |  < HOP:  "HOP" >
      |  < JUMP: "jump" >
      |  < WALK: "walk" >
      |  < LEAP: "leap" >
      |  < TURN: "turn" >
      |  < TURNTO: "turnto" >
      |  < DROP: "drop" >
      |  < GET: "get" >
      |  < GRAB: "grab" >
      |  < LETGO: "letGo" >
      |  < NOP: "nop" >
      |  < DEFVAR: "defVar" >
      |  < FACING: "facing" >
      |  < CAN: "can" >
      |  < NOT: "not" >
      |  < IF: "if" >
      |  < ELSE: "else" >
      |  < WHILE: "while" >
      |  < REPEAT: "repeat" >
      |  < TIMES: "times" >
      |  < DEFPROC: "defProc" >

}
```

Un aspecto importante de la definición de tokens es que para poder declarar correctamente la gramática de los comandos simples turn(D), walk(v, D) y leap(v, D) tuvimos que crear tres tipos de tokens para las direcciones:

```
| < DIR:  (< RIGLEF >) >
| < #RIGLEF:("right"|"left") >
| < AROUNDT:  (< AROUNN >) >
| < #AROUNN:  ("around") >
| < BACKFRONT:  (< BACFRON >) >
| < #BACFRON:  ("back"|"front") >
```

Puesto que las direcciones que aceptaban estos comandos varia. Por ejemplo, turn(D) solo acepta left, right y around. A diferencia de walk(v, D), que acepta right, left, back y front.

Ahora bien, para los tokens de números y cadenas creamos rangos de caracteres y los implementamos de tal manera que los tokens tengan mínimo un carácter:

```
<NUM: (<DIGIT>)+ >
|   <#DIGIT: ["0"-"9"] >
```

```
| < CAD: (< CADENA >)+ >
| < #CADENA: ["a"-"z","A"-"Z","0"-"9"] >
```

Definición de la Gramática e implementación:

En esta sección lo que hicimos fue definir varias funciones que leen los tokens y que se comunican con World para ejecutar los comandos del robot.

Inicialmente, creamos una función llamada comandos() que verifica que todos los comandos estén bien construidos, es decir, que contengan paréntesis, punto y coma, números o variables. Para ello se implementó la siguiente función:

```
void comandos():
{
    int x,y;
    String o,d;
    boolean retorno=true;
    salida=new String();
}
{
      (
       //<RIGHT> "(" ")" {world.turnRight();salida = "Command: Turnright";}
        <MOV>  "(" (x=num()|x=var()) ")" {world.moveForward(x,false); salida = "Command: Moveforward ";}
    |   <HOP>  "(" (x=num()|x=var()) ")" {world.moveForward(x,true); salida = "Command:Jumpforward ";}
    |   <GO>  "(" (x=num()|x=var()) "," (y=num()|y=var())")" {world.setPostion(x,y); salida = "Command:GO ";}
    |  <PUT> "("  put() ")"
    |  <PICK> "("  get()  ")"
    |  < POP > "(" (x=num()|x=var()) ")" {world.popBalloons(x); salida = "Command:  Pop";}
    |  < JUMP > "(" (x=num()|x=var()) "," (y=num()|y=var()) ")" {world.setPostion(x,y);
                                                                 salida = "Command: Jump";}
    |  < WALK > "(" walk() ")"
    |  < LEAP > "(" leap() ")"
    |  < TURN > "(" d=dirturn() ")" {if (d.equals("right"))
                                        world.turnRight();
                                    else if (d.equals("left"))
                                        for (int i = 0; i < 3; i++) {
                                            world.turnRight(); }
                                    else if (d.equals("around"))
                                        for (int i = 0; i < 2; i++) {
                                            world.turnRight(); }
                                    salida = "Command: Turn";}
    |  < TURNTO > "(" o=ori() ")" {toTurnTo(o);
                                    salida = "Command: TurnTo";}
    |  < DROP > "(" (x=num()|x=var()) ")" {world.putChips(x);
                                            salida = "Command:  Drop"; }
    |  < GET > "(" (x=num()|x=var()) ")" {world.pickChips(x);
                                            salida = "Command:  Get";}
    |  < GRAB > "(" (x=num()|x=var()) ")" {world.grabBalloons(x);
                                            salida="Command:  Grab";}
    |  < LETGO > "(" (x=num()|x=var()) ")" {world.putBalloons(x);
                                            salida = "Command:  LetGo";}
    |  < NOP > "(" ")" {salida = "Command: Nop";}
    ) ";"
}
```

Para esto, primero definimos funciones que leen los tokens de números, direcciones, y orientaciones, y retornan un tipo de dato que se pueda ingresar dentro de funciones implementadas en World o que nosotros mismos hayamos creado.

```
public int obtainOri(String ori) throws Error {
    int o;
    if (ori.equals("north")){
        o = NORTH;
        }
    else if (ori.equals("south")) {
        o = SOUTH;
    }
    else if (ori.equals("east")) {
        o = EAST;
    }
    else {
        o = WEST;
    }
    return o;
}
```

```
int num() throws Error:
{
    int total=1;
}
{
    <NUM>
    {
        try
        {
            total = Integer.parseInt(token.image);
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

```
String dir() throws Error:
{
    String total;
}
{
    (<DIR>|< BACKFRONT >)
    {
        try
        {
            total = (token.image);
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

```
String dirturn() throws Error:
{
    String total;
}
{
    (<DIR>|< AROUNDT >)
    {
        try
        {
            total = (token.image);
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

```
String cad() throws Error:
{
    String total="";
}
{
    <CAD>
    {
        try
        {
            total = (token.image);
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

Posteriormente, realizamos la definición de variables implementando las funciones mostradas a continuación.

```
String vardef() throws Error:
{
  String total,variableF;
}
{
    < CAD >
    {
      try
        {
            variableF = token.image;
            total = Integer.toString(obtenerValor(variableF));
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

```
void definicion():
{
    int d = 0;
    String x = null;
    String c;
    salida=new String();
}
{
     (
       < DEFVAR > c=cad() (d=num()|x=vardef()) {if (d != 0 && x == null) {
                                                x = Integer.toString(d); }

                                                try {
                                                    int valor = Integer.parseInt(x);
                                                    variable.put(c, x);
                                                } catch(Exception e) {
                                                    if (variable.containsKey(x)) {
                                                        variable.put(c, variable.get(x));
                                                    }
                                                    else {
                                                        throw new Error("No es un valor valido");
                                                    }
                                                }
                                                salida="Definicion Variable"; }
    )
}
```

Estas funciones se encargan de asignarle a una cadena el valor especificado por la definición de variables, esta función le puede asignar el valor de una variable a otra. Esto lo logramos a partir de la creación de un mapa que guarda como llaves el nombre de las variables y como valores el carácter del número en formato String. Si a una variable se le esta asignando el valor de otra variable, lo que hace la función es obtener el valor de la variable asignada en el mapa y asignarlo a la variable definida.

Las variables se pueden utilizar en funciones con la siguiente función:

```
int var() throws Error:
{
  int total;
  String variableF;
}
{
    < CAD >
    {
      try
        {
            variableF = token.image;
            total = (obtenerValor(variableF));
        }
        catch (NumberFormatException ee)
        {
            throw new Error("Number out of bounds: "+token.image+" !!");
        }
        return total;
    }
}
```

Lo que hace esta función es buscar el valor de la cadena dentro del mapa de variables y convertirlo a un dato simple de tipo int. Si la variable no ha sido definida previamente saca un Error.

Ahora bien, para la implementación de comandos simples utilizamos funciones que ya estaban implementadas dentro de World y también creamos algunas funciones para facilitar su implementación. Las implementaciones más desafiantes fueron las de los comandos walk() y leap(), debido a que estos comandos pueden aceptar uno o más parámetros dependiendo del caso. Para el comando leap() creamos la siguiente función:

```
void leap():
    {
  int f;
  String o=null;
  String d=null;
  boolean retorno=false;
}
{
  ((f=num()|f=var()) ("," (o=ori()|d=dir()))?)
  {
    if (o == null && d == null) {
        world.moveForward(f, true);
        salida = "Command: Leap(n)";

    } else if (o!=null){
        boolean jump = true;
        toTurnTo(o);
        int orient = obtainOri(o);
        if(orient == NORTH)
            world.moveVertically(-f,jump);
        else if(orient == SOUTH)
            world.moveVertically(f, jump);
        else if(orient == EAST)
            world.moveHorizontally(f, jump);
        else
            world.moveHorizontally(-f, jump);
        salida = "Command: Leap(n,orientation)";

    } else if (d!=null) {
        int miraInicial = world.getFacing();
        if (d.equals("right"))
            world.turnRight();
        else if (d.equals("left"))
            for (int i = 0; i < 3; i++) {
                world.turnRight(); }
        else if (d.equals("back"))
            for (int i = 0; i < 2; i++) {
                world.turnRight(); }
        world.moveForward(f, true);
        turnTo(miraInicial);
        salida = "Command: Leap(n,direction)";
    }
    }
}
```

Esta función inicializa varias variables con null, posteriormente se utiliza una serie de else/if que ejecuta distintos bloques de código dependiendo de que variables se hayan utilizado. Para el comando walk() se realizó una función muy parecida, la cual tiene como diferencia principal que le asigna a la variable jump un booleano falso, ya que con walk no puede saltar obstáculos.

Seguidamente, realizamos la definición de condiciones creando la siguiente función:

```
boolean commandConditional():
{
    boolean com=false;
    boolean retorno=false;
    String o;
    ArrayList<String> comandoArray = new ArrayList<String>();
}
{
  (
    < NOT > ":"  com=commandConditional() {retorno=(!com);
                                          salida = "Condition: Not"; }
     | < FACING > "(" o=ori()")" {int orien = obtainOri(o);
                                 if (orien == world.getFacing())
                                     retorno=true;
                                 else
                                     retorno=false;
                                  salida = "Condition: Facing";}
     | < CAN > "(" (comandoArray=comandosNoEjecutables()) ")" {retorno=auxiliarCan(comandoArray);
                                                               salida = "Condition: Can"; }
  )
{return retorno;}
}
```

Esta función ejecuta distintos bloques de código Java dependiendo del Token que reconozca el compilador. Si reconoce un not() la función utiliza recursión y niega el booleano retornado por la condición. En cambio, sí reconoce un facing() la función llama a la función getFacing de World y retorna un booleano dependiendo de la orientación.

Por otro lado, la implementación del can () fue un poco más complicada porque tocaba verificar si se podía ejecutar el comando dentro de la condición. Para esto, creamos una función auxiliar la cual guarda las coordenadas y la orientación del robot antes de intentar ejecutar el comando. Luego, con un try/catch ejecuta el comando. Si el comando se puede ejecutar la función devuelve al robot a sus condiciones previas a la ejecución del comando y retorna True. Contrariamente, si el comando no se puede ejecutar correctamente y por ende saca un error, el catch atrapa el error y devuelve false.

```
public boolean auxiliarCan(ArrayList<String> comandoArray)
{
  boolean retorno;
  int oriIni = world.getFacing();
  Point posInicial =world.getPosition();
  int coorX = posInicial.x;
  int coorY = posInicial.y;

  try {
    for (String elemento : comandoArray) {
           ejecutarComandoBloque(elemento); }
    world.setPostion(coorX,coorY);
    turnTo(oriIni);
    return true;
    } catch (Error e) {return false;}
}
```

Para el condicional If-Else, implementamos los comandos condicionales anteriormente mencionados junto con un arreglo que contiene todos los comandos que estén dentro del bloque de tanto el If como del Else. Posteriormente, se ejecuta el bloque de comandos uno por uno hasta llegar al final del arreglo de comandos. La función ejecutarComandoBloque(elemento) corre de acuerdo con el comando ingresado por parámetro. La implementación fue la siguiente:

```
void condicionalIf():
{
  ArrayList<String> bloqueArray = new ArrayList<String>();
  ArrayList<String> bloqueArray2 = new ArrayList<String>();
  boolean reto=false;
  salida=new String();
  int x;
}
{
  < IF > (reto=commandConditional()) "{" (bloqueArray=comandosNoEjecutables()) "}"  <ELSE > "{" (bloqueArray2=comandosNoEjecutables()) "}"
  {//ACA SI CORRER, DEPENDIENDO EL VALOR DE RETO, ALMACENAR EL BLOQUE EN UN ARRAY
    if (reto) {
        for (String elemento : bloqueArray) {
            ejecutarComandoBloque(elemento);
            }
        //EN TEORIA EN COMANDOBLOQ ESTA EL COMANDO..., AHORA COMO EJECUTO, PODRIA HACER UN IF GIGANTE

        salida = "If-else";}
    else {
        for (String elemento : bloqueArray2) {
            ejecutarComandoBloque(elemento);
            }
        salida = "If-else";}
  }
}
```

La implementación de los ciclos repeat times y while fue desafiante, el ciclo repeat times funciona correctamente, con lo realizado en el if-else simplemente se repite el bloque de codigo con un for las veces que entre por parametro. El ciclo while no logro ser implmentado correctamente.

```
void ciclos():
{
  int x;
  boolean reto=false;
  ArrayList<String> bloqueArray = new ArrayList<String>();
}
{
  < REPEAT > (x=num()|x=var()) < TIMES > "{" (bloqueArray=comandosNoEjecutables()) "}"
  { for (int i = 0; i<= x; i++) {
        for (String elemento : bloqueArray) {
            ejecutarComandoBloque(elemento); } }
    salida = "Ciclo repeat";}
| < WHILE > (reto=commandConditional())"{" (bloqueArray=comandosNoEjecutables()) "}"
{
  if (reto) {
      for (String elemento : bloqueArray) {
              ejecutarComandoBloque(elemento);}
      }
  else {}
    salida = "Ciclo while";}
}
```

Errores y aspectos por mejorar:

Para el defProc se definió únicamente la gramática y la estructura, no fue completamente implementado.

Algo que dificulto la ejecución de los ciclos y los if/else fue la forma en que decidimos realizar la ejecución del código. Debido a que nuestro compilador leía y ejecutaba el código simultáneamente, lo cual solucionamos parcialmente con la creación de un arreglo que contuviera todos los comandos del bloque.

En nuestra solución siempre despues de un comando se escribe ";", es necesario que este para el correcto funcionamiento

Adicionalmente cada vez que se ejecuta el programa y se ingresan comandos, o condiciones etc.. Se genera un error: "Error Missing return statement in function"; sin embargo, el robot se mueve correctamente y sin irregularidades.