Throughout this course we will develop a language in several stages. The project consists of managing and operating a language to program a factory robot that lives in a two-dimensional world. The robot is able to move in the world (delimited by an $n \times n$ matrix); the robot moves from cell to cell. Cells are indexed by rows and columns. The top left cell is indexed as (1,1). North is top; West is left. The robot interacts (picks and puts down) with two different types of objects (chips and balloons). Additionally, note that the robot cannot move on, or interact with obstacles in the world (gray cells).

Robot Description

In this project, Project 0, we will try to understand the robot language. That is, given a program for the robot, we want to determine if this satisfies the language specification and robot behavior, explained in the following.

Figure 1 shows the robot facing North in the top left cell. The robot carries chips and balloons which he can put and pickup. Chips fall to the bottom of the columns. If there are chips already in the column, chips stack on top of each other (there can only be one chip per cell). Balloons float in their cell, there can be more than one balloon in a single cell.

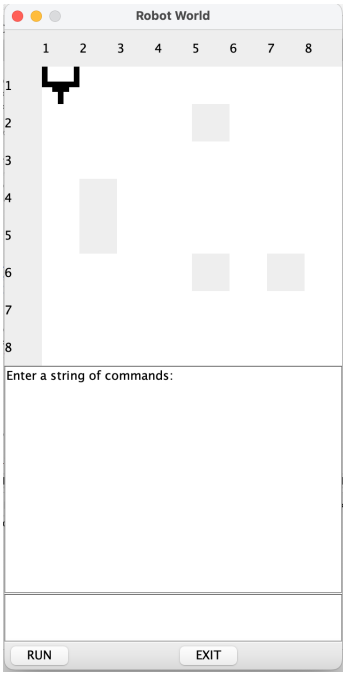

Figure 1: Initial state of the robot's world

The attached Java project includes a simple interpreter for the robot. The interpreter

reads a sequence of instructions and executes them. An instruction is a command followed by ";".

A command can be any one of the following:

- M: to move forward
- R: to turn right
- C: to drop a chip
- B: to place a balloon
- c: to pickup a chip
- b: to grab a balloon
- P: to pop a balloon
- J(n): to jump forward n steps. It may jump over obstacles, but the final position should not have an obstacle.
- G(x,y): to go to position (x,y). Position (x,y) should not have an obstacle.

The interpreter controls the robot through the class `uniandes.lym.robot.kernel.RootWorldDec`

Figure 2 shows the robot before executing the commands that appear in the text box area at the bottom of the interface.

Figure 3 shows the robot after executing the aforementioned sequence of commands. The text area in the middle of the figure displays the commands executed by the robot.

Below we define a language for programs for the robot.

Input for the robot can be an execution command or a definition.

- An execution command is preceded by the word `EXEC` and followed by a block.
- Definitions are preceded by the keyword `NEW`. We can define variables and macros.
  - A variable definition is of the form: `NEW VAR name=n` where `name` is a variable's name and `n` is a value used to initialize the variable.
  - A macro definition is of the form: `NEW MACRO name (params)B`. This is used to define a procedure. Here `name` is the procedure name, `(Params)` is a list of parameters separated by commas, and `B` is a block. A parameter is a name.
- A block is of the form $\{instruction; \ldots instruction;\}$

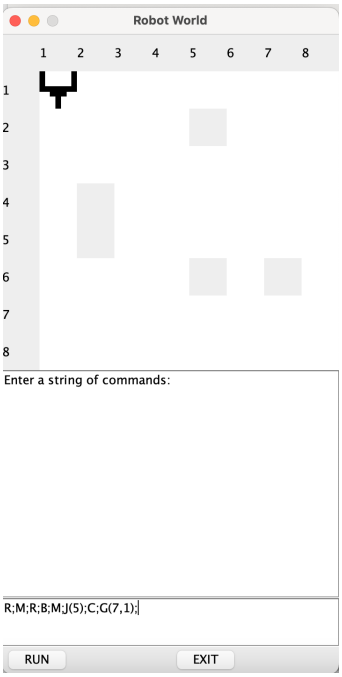

Figure 2: Robot before executing commands

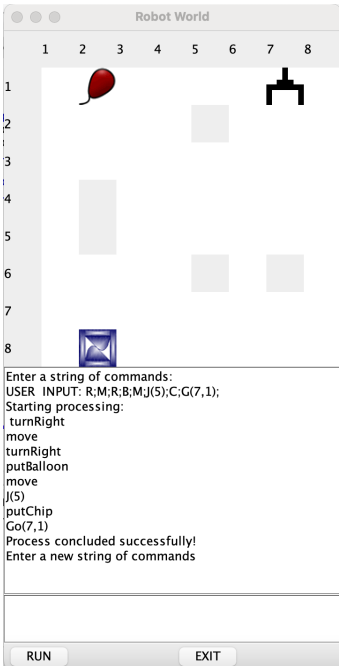

Figure 3: Robot executed commands

- An instruction can be a command or a control structure.
- A command can be:

  1. Assignment: `name = n` where `name` is a variable's name and `n` is a value. The result of this instruction is to assign the value `n` to the variable.
  2. Macro invocation: A macro name followed by its parameter values within parenthesis and separted by commas, for example `macroName(p1,p2,p3)`.
  3. `turnToMy(D)`: where `D` can be `left`, `right`, or `back` (The robot should turn 90 degrees in the direction of the parameter (if the parameter is `left` or `right`), and 180 if the parameter is `back`.
  4. `turnToThe(O)`: where `O` can be `north`, `south`, `east`, or `west`. The robot should turn so that it ends up facing direction `O`.
  5. **walk**`(n)`: where `n` is a value. The robot should move `n` steps forward.
  6. `jump(n)`: where `n` is a value. The robot should jump `n` steps forward.
  7. **drop**`(n)`: Where `n` is a value. The robot should drop `n` chips.
  8. **pick**`(n)`: Where `n` is a value. The robot should pick `n` chips.
  9. **grab**`(n)`: Where `n` is a value. The robot should get `n` balloons.
  10. `letGo(n)`: Where `n` is a value. The robot should put `n` balloons.
  11. `pop(n)`: Where `n` is a value. The robot should pop `n` balloons.
  12. `moves (Ds)`: where `Ds` is a non-empty list of directions: `forward`, `right`, `left`, or `backwards` separated by commas. The robot should move in the directions indicated by the list and end up facing the same direction as it started.
  13. `nop`: a command that does not do anything
  14. `safeExe(C)`: C is any command (5-11, above). C is executed if it will not cause an error. Otherwise, it does not do anything.

- Values: A value is a number, a variable, or a constant. The constants that can be used are:
  - size : the dimensions of the board
  - myX: the x position of the robot
  - myY: the y position of the robot
  - myChips: number of chips held by the robot
  - myBalloons: number of balloons held by the robot

  - balloonsHere: number of balloons in the robot's cell
  - chipsHere: number of chips that can be picked
  - roomForChips: number of chips that can be dropped

- Control Structures:

  Conditional: `if (condition)then B1 else B2 fi`: Executes `B1` if `condition` is true and `B2` if `condition` is false. `B1` and `B2` are blocks.

  Loop: `do (condition)B od`: Executes `B` while `condition` is true. `B` is a block.

  RepeatTimes: `rep n times B per` where `n` is a value and `B` is a block. `B` is executed `n` times.

- These are the conditions:

  - `isBlocked?(D)`: `D` is `left`, `right`, `front` or `back`. The condition is true if the robot is blocked at the corresponding direction.
  - `isFacing?(O)`: `O` is `north`, `south`, `east`, or `west`.The condition is true if the robot is facing the corresponding orientation.
  - `zero?(n)`: Returns true if `n` is zero.
  - `not(C)`: `C` is a condition. It is true if C is false.

Spaces, newlines, and tabulators are separators and should be ignored.

The language is not case-sensitive. This is to say, it does not distinguish between upper and lower case letters.

Remember the robot cannot walk over obstacles, and when jumping, it cannot land on an obstacle. The robot cannot walk off the board or land off the board when jumping.

Below we show examples of valid inputs.

Task 1. The task of this project is to use Python or Java to implement a simple yes/no parser. The program should read a text file that contains a input for the robot, and verify whether the syntax is correct.

You must verify that used variable names have been previously defined and in the case of macros, that they have been previously defined and are called with valid parameter values. You must allow recursion.

Spaces and tabulators are separators and should be ignored.

You may use APIs to generate a lexer. However, you may dot use APIs to generate a parser.

```
EXEC  {
if not(blocked?(left)) then  { turnToMy(left); walk(1); } else {
   nop;}  fi
}

EXEC {
 safeExe(walk(1));
 moves(left,left, forward, right, back);
}

NEW VAR rotate= 3
NEW MACRO foo (c, p)
{  drop(c);
   letgo(p);
   walk(rotate);
}
EXEC  { foo (1 ,3); }

NEW VAR one= 1
NEW MACRO      goend ()
{
   if not (blocked?(front))
   then  { move(one); goend();  }
   else  { nop; }
    fi;
}

NEW MACRO fill ()
  {
  repeat roomForChips times
  {  if not (zero?(myChips)) { drop(1);}  else { nop; } fi ;} ;
  }

  NEW MACRO fill1 ()
  {
  while not zero?(rooomForChips)
  {  if not (zero?(myChips)) { drop(1);}  else { nop; } fi ;
  } ;
  }

NEW MACRO grabAll ()
{ grab (balloonsHere);
}
```