Throughout this course we will develop a project in several stages. The project consists of managing and operating a language to program a factory robot 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 will like to see 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.

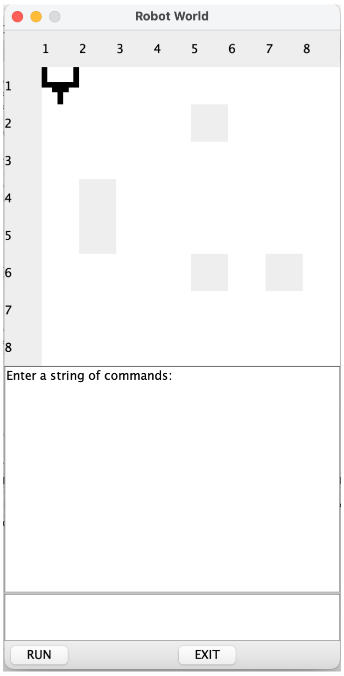

Figure 1: Initial state of the robot's world

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.

A program for the robot is a sequence of instructions.

- An instruction can be a command, a control structure or a function call.
  - A command can be any one of the following:
    * `(defvar name n)` where `name` is a variable's name and `n` is a value used initializing the variable.
    * `(= 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.
    * `(move n)`: where `n` is a value. The robot should move `n` steps forward.
    * (**skip** `n)`: where `n` is a value. The robot should jump `n` steps forward.

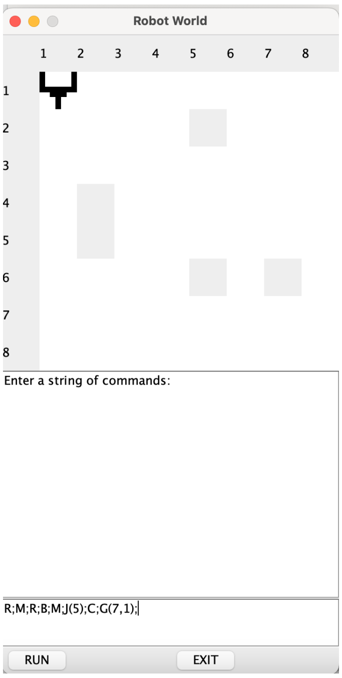

Figure 2: Robot before executing commands

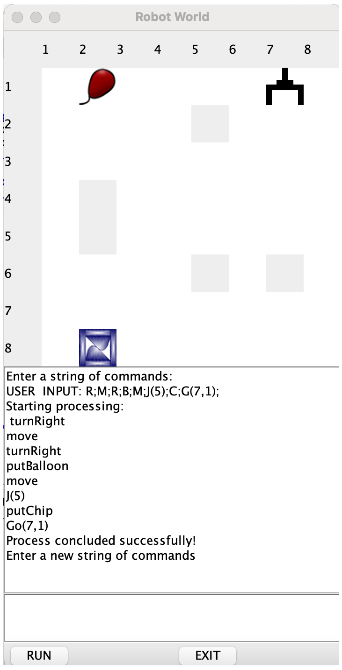

Figure 3: Robot executed commands

* `(turn D)`: where `D` can be `:left`, `:right`, or `:around` (defined as constants). The robot should turn 90 degrees in the direction of the parameter in the first to cases, and 180 in the last case.
* `(face O)`: where `O` can be `:north`, `:south`, `:east`, or `:west` (all constants). The robot should turn so that it ends up facing direction `O`.
* `(`**put** `X n)`: where `X` corresponds to either `:balloons` or `:chips`, and `n` is a value. The Robot should put `n` `X`'s.
* `(`**pick** `X n)`: where `X` is either `:balloons` or `:chips`, and `n` is a value. The robot should pick `n` `X`'s.
* `(move-dir n D)`: where `n` is a value. `D` is one of `:front`, `:right`, `:left`, `:back`. The robot should move `n` positions to the front, to the left, the right or back and end up facing the same direction as it started.
* `(run-dirs Ds)`: where `Ds` is a non-empty list of directions: `:front`, `:right`, `:left`, `:back`. The robot should move in the directions indicated by the list and end up facing the same direction as it started.
* `(move-face n O)`: here `n` is a value. `O` is `:north`, `:south`, `:west`, or `:east`. The robot should face `O` and then move `n` steps.
* `(`**null**`)`: a instruction that does not do anything
* A vaue is a number, a variable, or a contant.
* These are the constants that can be used:
  · Dim : the dimensions of the board
  · myXpos: the x postition of the robot
  · myYpos: 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
  · Spaces: number of chips that can be dropped
* A control structure can be:

  Conditional: `(`**if** `condition B1 B2)`: Executes `B1` if condition is true and `B2` if `condition` is false. B1 and B2 can be a single command or a Block

  Repeat: `(loop condition B)`: Executes `B` while `condition` is true. B can be a single command or a block.

  RepeatTimes: `(`**repeat** `n B)` where `n` is a value. `B` is executed `n` times. B is a cingle command or a block.

FunctionDefinition: `(defun name (Params)Cs)` where `name` is the function name, `(Params)` is a list of parameter names for the function (separated by spaces) and `Cs` is a sequence of commands for the function. A function is called by giving its name followed by its parameter values within parenthesis as with any other instruction, for example `(funName p1 p2 p3)`.

∗ A condition can be:

- `(facing? O)` where `O` is one of: north, south, east, or west
- `(blocked?)` This is true if the Robot cannot move forward.
- `(can-put? X n)` where `X` can be chips or balloons, and `n` is a value.
- `(can-pick? X n)` where `X` can be chips or balloons, and `n` is a value.
- `(can-move? D)` where `D` is one of: `:north`, `:south`, `:west`, or `:east`
- `(isZero? V)` where `V` is a value.
- `(not cond)` where `cond` is a condition.

Blocks are sequences of instructions delimited by parenthesis `()`.

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.

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 program 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 functions, 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.

Below we show examples of valid inputs.

```
(defvar rotate 3)



(if (can-move? :north ) (move-dir 1 :north) (null))

(
(if (not (blocked?)) (move 1) (null))
(turn :left)
)

(defvar one 1)

(defun foo (c p)
   (put :chips c)
   (put :balloons p)
   (move rotate))
(foo 1 3)

(defun goend ()
   (if (not (blocked?))
   ((move one)
      (goend))
   (null)))

(defun fill ()
 (repeat Spaces (if (not (isZero? myChips)) (put :chips 1) ))
)

(defun pickAllB ()
 (pick :balloons balloonsHere)
)

(run-dirs  :left :up :left :down :right)
```