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 1, we will use JavaCC to build an interpreter for the Robot Language introduced in Project 0.

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.

The attached Java project includes a simple JavaCC interpreter for the robot.[1] The interpreter reads a sequence of instructions and executes them. An instruction is a command followed by an end of line.

A command can be any one of the following:

- move(n): to move forward n steps
- turnright(): to turn right
- Put(chips,n): to drop n chips
- Put(balloons,n): to place n balloons
- Pick(chips,n): to pickup n chips
- Pick(balloons,n): to grab n balloons
- Pop(n): to pop n balloons

[1] The given interpreter is used for a different robot language, but can be used as a starting point for your own interpreter.

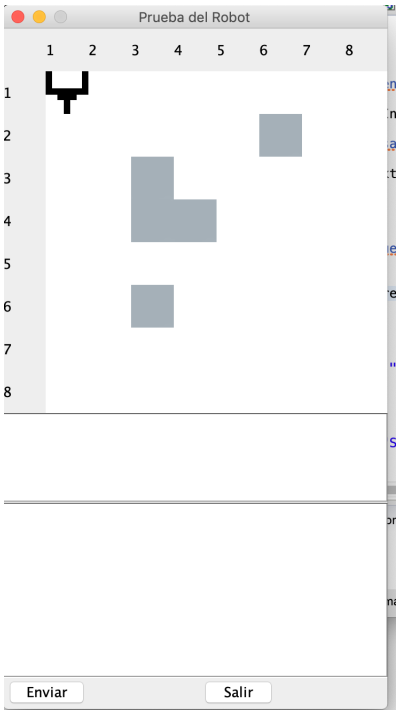

Figure 1: Initial state of the robot's world

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.

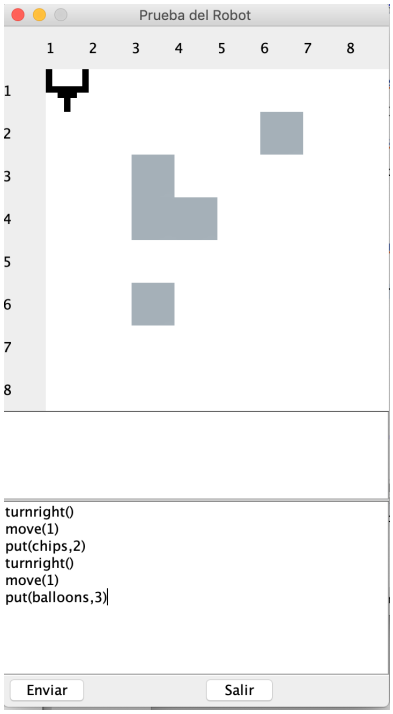

Figure 2: Robot before executing commands

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 commands and blocks.

- An instruction can be a command or a control structure.
  - A command can be any one of the following:
    * `(defvar name n)` where `name` is a variable's name and `n` is a number initializing the variable.
    * `(= name n)` where `name` is a variable's name and `n` is a number. The result of this instruction is to assign the value of the number `n` to the variable.
    * `(move n)`: where `n` is a number or a variable. The robot should move `n` steps forward.

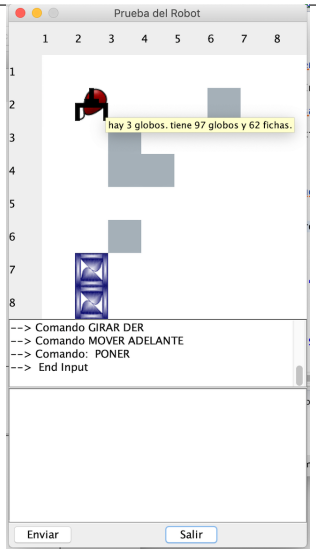

Figure 3: Robot executed commands

* `(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 number or a variable. The Robot should put `n` `X`'s.
* `(pick X n)`: where `X` is either `:balloons` or `:chips`, and `n` is a number or a variable. The robot should pick `n` `X`'s.
* `(move-dir n D)`: where `n` is a number or a variable. `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 number or a variable. `O` is `:north`, `:south`, `:west`, or `:east`. The robot should face `O` and then move `n` steps.
* `(skip)`: a instruction that does not do anything
* a function cll

– A control structure can be:

Conditional: `(if condition Block1 Block2)`: Executes `Block1` if condition is true and `Block2` if `condition` is false.

Repeat: `(loop condition Block)`: Executes `Block` while `condition` is true.

RepeatTimes: `(repeat n Block)` where `n` is a variable or a number. `Block` is executed `n` times.

FunctionDefinition: `(defun name (Params)Block)` where `name` is the function name, `(Params)` is a list of parameter names for the function (separated by spaces) and `Block` is the set of instructions for the function. Note that functions are called given their name and parameters as with any other instruction, for example by calling `(name Params)`.

- A condition can be:
  - `(facing-p O)` where `O` is one of: north, south, east, or west
  - `(can-put-p X n)` where `X` can be chips or balloons, and `n` is a number or a variable
  - `(can-pick-p X n)` where `X` can be chips or balloons, and `n` is a number or a variable
  - `(can-move-p D)` where `D` is one of: `:north`, `:south`, `:west`, or `:east`
  - `(not cond)` where `cond` is a condition.
  - A defined function

Blocks of instructions are defined by a single function call, or a list of functions delimited by parenthesis `()`. Spaces, newlines, and tabulators are separators and should be ignored.

Task 1. The task of this project is to modify the parser defined in the JavaCC file `uniandes.lym.robot.control.Robot.jj` (you must **only** send this file), so that it can interpret the new language described above. You may not modify any files in the other packages, nor `uniandes.lym.robot.control.iterprete.java`.

Below we show an example first of an invalid input, then of a valid input.
An example of a valid input for the robot would be the following:

```
(defvar rotate 3)

(if (blocked-p) (move 1)
  (skip))


(if (blocked-p) (move 1) ((skip)))

(left 90)

(defvar one 1)

(defun foo (c p)
   (put :chips c)
   (put :balloons p)
    (move n))

(foo 1 3)

(defun goend ()
   ((if (not blocked-p)
   ((move 1)
    (goend))
   (skip))))
```

```
(defvar rotate 3)

(defun blocked-p ()
   (not (can-move-p :north)))
(if (blocked-p)
    (move 1)
    (skip))

(
(if (blocked-p) ((move 1) (move 1))  (skip))
(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-p)
   ((move one)
      (goend))
   (skip)
   ))

(run-dirs  (:left :front :left :back :right))
```