In this project we will use JavaCC to build an interpreter for the robot language introduced in Project 0.

Robot Description

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). Also, the robot cannot move into, through, nor interact with obstacles in the world (gray cells).

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.

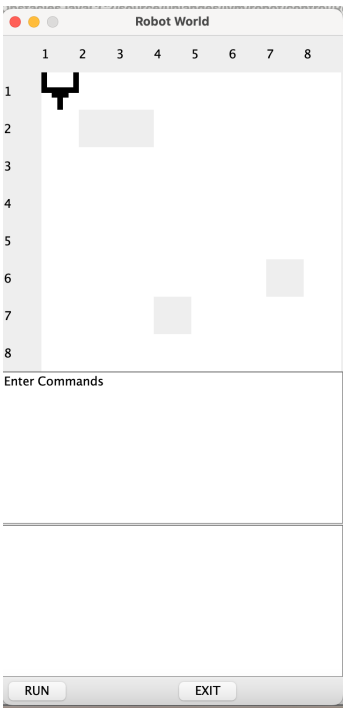

Figure 1: Initial state of the robot's world

The attached Java project includes a simple 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
- right(): 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
- Hop(n): To jump n postions forward
- Go(x,y): To go to position x,y

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

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.

This is the language for robot programs:

A program for the robot is a sequence of *definitions* and *instructions*.

- A *definition* can be a variable definition or a procedure definition.
  - A variable definition has the form `(defvar name n)` where `name` is the variable's name and `n` is a value used initializing the variable.

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

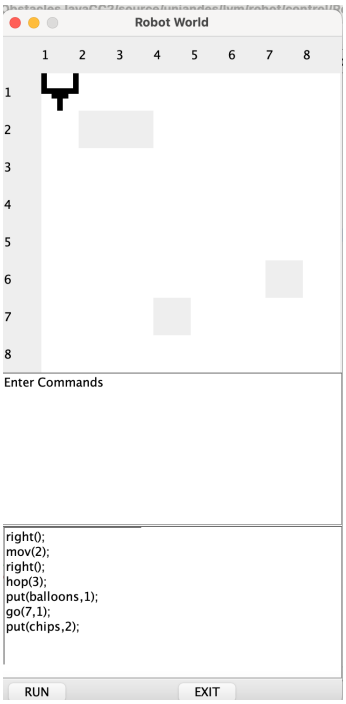

Figure 2: Robot before executing commands

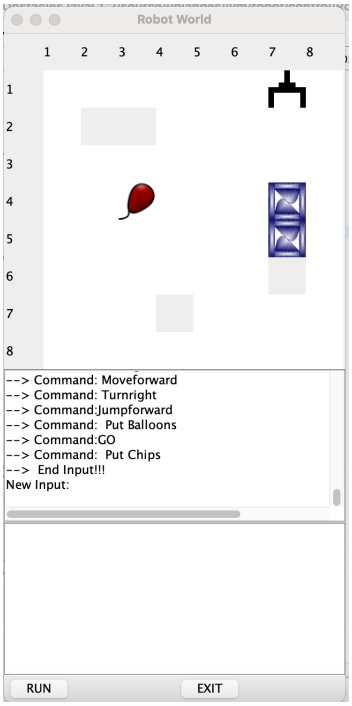

Figure 3: Robot executed commands

  - A procedure definition has the form `(defun name (Params)Is)` . This is used to define a procedure. Here `name` is the procedure name, `(Params)` is a list of parameter names for the procedure, separated by spaces, and `Is` is a sequence of instructions for the procedure.

- An *instruction* can be a procedure call, a command, or a control structure.
  - A procedure is called by giving its name followed by its parameter values within parenthesis, as in `(funName p1 p2 p3)`.
  - A command can be any one of the following:
    * `(= name n)` where `name` is the 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.
    * `(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 single command or a block.

* 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 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 language described above. You may not modify any files in the other packages, nor `uniandes.lym.robot.control.Interpreter.java`.

Below we show an example of a valid program.

```
(defvar myvar 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 myvar))
(foo 1 3)

(defvar anothervar 0)

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

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

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

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