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.

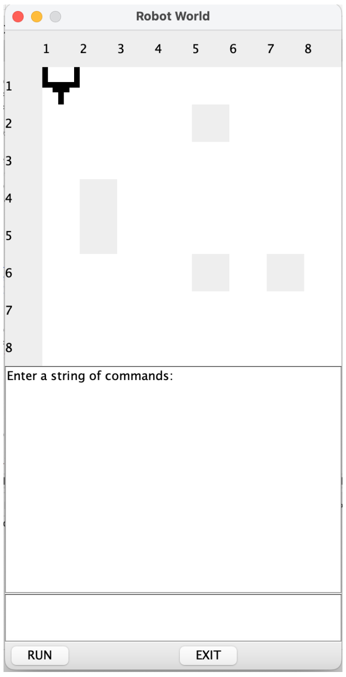


Figure 1: Initial state of the robot's world

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 ";".

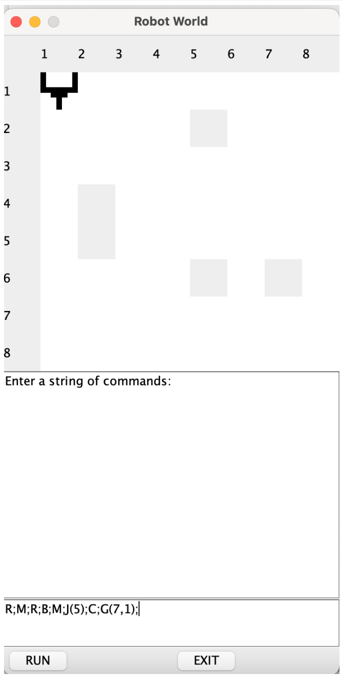
A command can be any one of the following:

- 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`

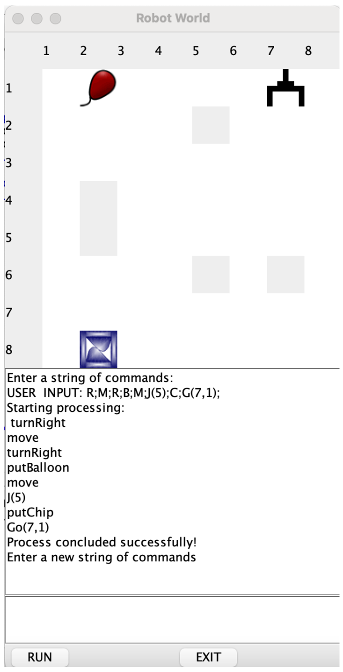
Figure 2 shows the robot before executing the commands that appear in the text box area at the bottom of the interface.

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.



```
Enter a string of commands:
R;M;R;B;M;J(5);C;G(7,1);
```

Figure 2: Robot before executing commands



```
Enter a string of commands:
USER  INPUT: R;M;R;B;M;J(5);C;G(7,1);
Starting processing:
 turnRight
move
turnRight
putBalloon
move
J(5)
putChip
Go(7,1)
Process concluded successfully!
Enter a new string of commands
```

Figure 3: Robot executed commands

To ease use, we want to build a language for the robot that will allow us to succinctly are more clearly write complete procedures for the robot to execute.

- Robot programs are a sequence of variable definitions, procedure definitions, variable accesses, and procedure and instruction calls (within blocks).
- Variables are always declared at the beginning of a a program (top of the file) and are defined as a list of names (starting with a small caps letter) and separated by spaces. The list is defined in a block separated by a pipe `|` symbol. Variables declared inside a procedure are local to that procedure and should be defined at the beginning of the procedure. For example `|t1 t2 t3|` declares three variables named `t1`, `t2`, and `t3`, respectively
- Procedure declaration starts with the keyword `proc` followed by the procedure name, its parameters, and a code block.
- Procedure names are defined as a combination of strings and parameter values. Procedure names are given by an alphanumeric string starting with a smallcaps letter and ending with a colon (`:`). After the name the parameter is given. Note that the name may be separated by spaces and parameters from the previous and following parts of the name. An example of a procedure definition with three parameters splitting a name is `proc myMethod: param1 with: param2 and: param3 [ ]`. Note that for procedures without parameters there is no need to add the colon at the end of the name, for example, `proc new [ ]`.
- A procedure call is given by the procedure name and its parameters, finishing with a period (`.`). For example, `myMethod: 3 and: 4.` calls the procedure previously declared with parameters 3 and 4.
- A code block is a sequence of instructions (*e.g.,* variable declarations, assignments, procedure calls, conditionals, loops) separated by periods within square brackets `[ ]`.
- The robot is able to recognize the following set of instructions:
  - Variable assignments that are defined as a variable name the assignment operator (`:=`), the given value, and a period. For example, `t1 := 3 .` The result of this instruction is to assign the value of the number to the variable.
  - constants of the language are alphanumeric strings that start with a `#`. For example `#balloons`

- `goto: x with: y .` – where `x` and `y` are numbers or variables. The robot should go to position `(x, y)`.
- `move: n .` – where `n` is a number or a variable. The robot should move n steps forward.
- `turn: D .` – where `D` can be `#left`, `#right`, or `#around`. In the first two cases the robot should turn 90 degrees in the direction of the parameter (according to the direction the robot is facing), and in the last case the robot should turn 180 degrees (end up facing the oposite direction).
- `face: O .` – where `O` can be `#north`, `#south`, `#west`, or `#east`. The robot should turn so that it ends up facing direction `O`.
- `put: n ofType: X .` – where `X` corresponds to either `#balloons` or `#chips`, and `n` is a number or a variable. The Robot should put `n` `X`'s.
- `pick: n ofType: X .` – where `X` is `#balloons` or `#chips` and `n` is a number or a variable. The robot should pick `n` `X`'s.
- `move: n toThe: D .` – where `n` is a number or a variable. `D` is a direction, either `#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.
- `move: n inDir: 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.
- `jump: n toThe: D .` – where `n` is a number or a variable. `D` is a direction, either `#front`, `#right`, `#left`, `#back`. The robot should jump `n` positions to the front, to the left, the right or back and end up facing the same direction as it started.
- `jump: n inDir: O .` – here `n` is a number or a variable. `O` is `#north`, `#south`, `#west`, or `#east`. The robot should face `O` and then jump `n` steps.
- `nop .` The robot does not do anything.

Conditional: `if: condition then: Block1 else: Block2` – Executes `Block1` if `condition` is true and `Block2` if `condition` is false.

Loop: `while: condition do: Block` – Executes `Block` while `condition` is true.

RepeatTimes: `for: n repeat: Block` – Executes `Block` for `n` times, where `n` is a variable or a number.

- A condition can be:

- `facing: O .` – where `O` is one of: `#north`, `#south`, `#west`, or `#east`
- `canPut: n ofType: X .` – where `X` can be `#balloons` or `#chips`, and `n` is a number or a variable
- `canPick: n ofType: X .` – where `X` can be `#balloons` or `#chips`, and `n` is a number or a variable
- `canMove: n inDir: D .` – where `D` is one of: `#north`, `#south`, `#west`, or `#east`
- `canJump: n inDir: D .` – where `D` is one of: `#north`, `#south`, `#west`, or `#east`
- `canMove: n toThe: O .` – where `O` is one of: `#front`, `#right`, `#left`, `#back`
- `canJump: n toThe: O .` – where `O` is one of: `#front`, `#right`, `#left`, `#back`
- `not: cond` – where `cond` is a condition

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

Task 1. The task of this project is to implement the grammar of the new language for the Robot to generate a lexer and parser for it. In this instance of the project, you should read a text file that contains an input program for the robot, but now you will have to use JavaCC to parse the program and effectively execute the functions in the program with the Robot. You must implement the connection between JavaCC and the Robot.

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

An example of the connection is given as an attachment to the project. The attached Java project includes a simple JavaCC interpreter for the robot (`uniandes.lym.robot.control.Robot.jj`).[1] The interpreter reads a sequence of instructions and executes them. An instruction is a command followed by an end of line.

A command in the example language 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

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

- 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.RootWorldDec`

ONLY hand in the `.jj` file for your project.

As before, below we show examples of valid inputs which you can use as inspiration to test full robot programs.

```
|nom x y one|

proc putChips: n andBalloons: m [
    |c, b|
    c := n .
    b := m .
    put : c ofType: #chips .  put: b ofType:  #balloons ]

proc goNorth  [
    while: canMove: 1 inDir: #north do: [ move: 1 inDir: #north .
        ]
]


proc goWest [
   if: canMove: 1 inDir: #west then: [move: 1 inDir: #west]  else
      : [nop .]]

[
    goTo: 3 with: 3 .
    putChips: 2 andBalloons: 1 .
]
```