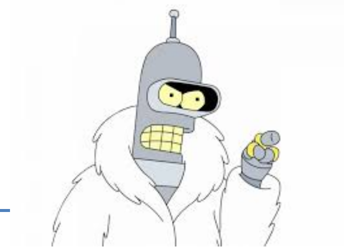
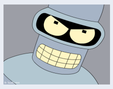
Mandatory Exercise 1 INF3110



In this exercise, you are going to write a small interpreter for a simple language for controlling a robot on a 2-dimensional grid. The language is called ROBOL, a clever acronym for “ROBOT LANGUAGE”, and its grammar is defined below.

The grid on which the robot can move about is defined by its x and y bounds, for instance:

0,6							7,6
							
0,0							7,0

The grid above is defined by the bound (7, 6), and the robot is currently located at position (3, 3). Moving the robot 1 step north would put it at (3,4). Moving it one step east would put it at (4,3), etc.

Assignment

Make an interpreter for the ROBOL language in your object-oriented language of choice (e.g. Java or Python; if you want to choose another language, please ask the gruppelærer). The interpreter shall operate on an *abstract syntax tree* (AST) representing a ROBOL program. You **do not** need to write a scanner or a parser for the language (however, if you want to rise to the challenge, feel free!).

You can design the classes for the AST as you like, but they should provide a somewhat faithful representation of the grammar listed below. The outermost element *program* from the grammar should be represented by a class Program, that provides an *interpret*-method which, when called, will interpret the entire program.

Requirements:

- The interpreter must check that the poor robot does not fall off the edge of the world (i.e., moves beyond the bounds of the grid).

- You can display the state of the program in any form you like during execution, but at minimum, the program should, upon termination, print its state in the form of the current location of the robot.

- There are some example programs below. You should check that your implementation returns the correct result after running these programs, and include instructions on how to run their AST representations with your implementation.

- Write a design document that explains how you have implemented the interpreter, and why you have done it in this way. Furthermore, the document should explain how to run your program from the Linux command line.

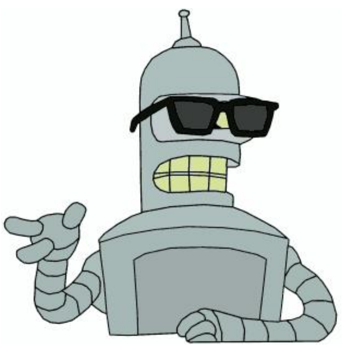
- It should be possible to use the program from the command line like this:
  `<ProgramName> 1|2|3|4|all`
  Example: a Java program with the main method in a class called Oblig1.
  `java Oblig1 2` – should run test code 2 and print the results
  `java Oblig1 all` – should run all the 4 programs and print the results
  There is an implementation of this provided in the program sketch

Deliverables

- The entire program and the design document should be placed in a single .zip file

- The name of the file should be INF3110_Mandatory1_<username>.zip

- The submission is done through Devilry: https://devilry.ifi.uio.no/



ROBOL Grammar

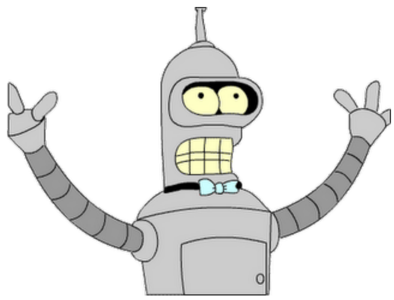
// a program consists of a robot, and a grid on which it can move around
<program> ::= <grid> <robot>

// size of the grid given as a bound for the x axis and the y axis; both axes
// start at 0, number is a positive integer.
<grid> ::= **size (**<number> * <number>**)**

// the robot has a list of variable declarations, a starting point, and a
// a set of statements that control its movement
<robot> ::= <var-decl>* <start> <stmt>*

// a variable declaration consists of a name and an initial value
<var-decl> ::= **var** <identifier> **=** <exp>

// statements control the robot's movement
<stmt> ::=
<stop>
| <move> <exp>
| <assignment>
| <loop>



// start gives the initial position for the robot
<start> ::= **start (**<exp> , <exp>**)**
<stop> ::= **stop**

// on the grid, moving north means up along the y axis, east means to the right
// along the x axis, etc.
<move> ::= **north** | **south**| **east**| **west**

<assignment> ::= <identifier>++ | <identifier>--

<loop> ::= **loop {** (<stmt> |<break>) + **}**
<break> ::= **break-if** <boolean-exp>

// expressions; number is an integer, identifier is a string of
// letters and numbers, starting with a letter
<exp> ::=
<identifier> | <number>| **(**<exp>**)** | <arithmetic-exp> | <boolean-exp>

<boolean-exp> ::=
(<boolean-op> <exp> <exp>)

<boolean-op> ::= < | > | =

<arithmetic-exp> ::=
    (<arithmetic-op> <args>**)**

<arithmetic-op> ::= + | - | *
//at least 2 arguments, but could be more
<args> ::= <exp> <exp> <exp>*

Legend:
<non-terminal>
terminal

Hints, program sketch and example programs

Hints:

- You may assume that expressions are type-correct (so you do not have to implement a type checker). You can assume that no-one writes programs that tries to add Booleans and numbers, for instance.
- It might simplify things if all expressions can calculate an integer value. Boolean expressions can, for instance, return 1 for true and 0 for false.
- The robot probably needs to have a reference to the grid, and the statements probably need to have a reference to the robot. This can be achieved in many ways, choose one that fits with your overall design.

Program sketch:

Below is a Java sketch of an implementation of the interpreter. You can use this as a starting point for your own implementation, if you like. You may also change all of these definitions if you think that is necessary.

```
class Oblig1 {
  public static void main(String[] args) {
    TestCode testCode = new TestCode();

    switch(args.length > 0 ? args[0] : "" ) {
      case "1": testCode.runProgram1(); return;
      case "2": testCode.runProgram2(); return;
      case "3": testCode.runProgram3(); return;
      case "4": testCode.runProgram4(); return;
      case "all": testCode.runAll(); return;
      default: System.out.println("USAGE: java Oblig1 1|2|3|4|all"); return;
    }
  }
}

class TestCode {
  void runProgram1() {
    // Create the AST based on testing code 1
   // This code is just to help you understand how to create an AST

    Grid grid = new Grid(new NumberExp(64),new NumberExp(64));
    Start start = new Start(new NumberExp(23),new NumberExp(30));
```

```
    statements.add(new Move(Direction.west, new NumberExp(15)));
    statements.add(new Move(Direction.south, new NumberExp(15)));
    Program prog;
    // Fill in rest of the code

    // Run the interpreter
    prog.interpret();
  }

  void runProgram2() {
    //same as runProgram1 but with the AST based on testing code 2
  }

  void runProgram3() {

  }

  void runProgram4() {

  }

  void runAll() {
    runProgram1();
    runProgram2();
    runProgram3();
    runProgram4();
  }
}

interface Robol {
  void interpret();
}

class Program implements Robol {
  Grid grid;
  Robot robot;

  public Program(Grid grid, Robot robot) {
    this.grid = grid;
    this.robot = robot;
  }

  public void interpret() {
    robot.interpret();
  }
}

class Robot implements Robol {
  public void interpret() {
```

```
      // write interpreter code for the robot here
    }
}

abstract class Statement implements Robol {
    public abstract void interpret();
}
class Assignment extends Statement {
   public void interpret() {
      // write interpreter code here
    }
}

class Loop extends Statement {
    BoolExp condition;
    List<Statement> statements;

    public void interpret() {
      // write interpreter code here
    }
}

abstract class Expression {  ... }

abstract class BoolExp extends Expression {
    protected Expression left;
    protected Expression right;
    ...
}
```

Example programs:

```
***************************************
Testing Code 1: Simple Example
size (64*64)
start(23,30)
west 15
south 15
east (+ 2 3)
north (+ 10 27)
stop
***************************************
The result is (13, 52)

***************************************
Testing Code 2: Example with variables
size(64*64)
var i = 5
```

```
var j = 3
start(23,6)
north (* 3 i)
east 15
south 4
west (+ (* 2 i) (* 3 j) )
stop
```

****

The result is (19, 17)

****

Testing Code 3: Example with loop and assignment

```
size(64*64)
var i = 5
var j = 3
start(23,6)
north (* 3 i)
west 15
east 4
loop
{
   break-if (< j 1)
   south j
   j--
}
stop
```

****

The result is (12, 15)

****

Testing Code 4: Example with movement over the edge

```
size(64*64)
var j = 3
start(1,1)
loop
{
   north j
   break-if (> j 100)
}
stop
```

****

The result should be an error saying that the bounds of the grid have been overstepped.