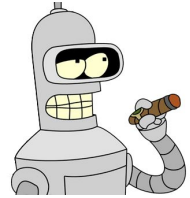# Mandatory Exercise 2 INF3110

Write an interpreter for the ROBOL language in SML, with the same requirements (except parse tree/AST drawings), and the same example programs, as for Mandatory Exercise 1 (attached below).

Deliverables
- The entire program and the design document should be placed in a single .zip file
- The name of the file should be INF3110_Mandatory2_<username>.zip
- The submission is done through Devilry: https://devilry.ifi.uio.no/

Deadline: 2016-11-04.

Program sketch

You may use this sketch as a *starting point* for your implementation. You are allowed to change these declarations as you see fit.

```
datatype direction = …
datatype grid = Size of int * int;

datatype exp =
    Identifier of string
…

datatype boolexp =
    BiggerThan of exp * exp
  | LessThan of exp * exp
…

datatype stmt =
 Move of direction * exp
…

datatype robot = Robot of vardecl list * start * stmt list …
datatype program = Program of grid * robot  ;

fun evalExp(BiggerThan(e1, e2), decls) =
        if evalExp(e1, decls) > evalExp(e2, decls) then 1 else 0
…
exception OutOfBounds;

fun interpret ( Program(Size(x, y),
      Robot(decls, Start(xpos, ypos),
            Move(direction, exp) :: stmtlst))) = …
```

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 an object oriented language[1] (e.g. Java or C#). 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 (you can assume that some benevolent entity have already written those for you) and can write the AST instances directly into your program.

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

---

[1] In this assignment "object oriented" is defined as having classes and inheritance. If you choose something other than Java or C# make sure to get your choice approved by a group teacher.

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 should be easy to extend (for your own sake too, as you will have to later on).

- 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 command line.

- Your delivery should also include a drawing of a parse tree (corresponding to the grammar below) and an abstract syntax tree (corresponding to your own code) of the following code[2]:
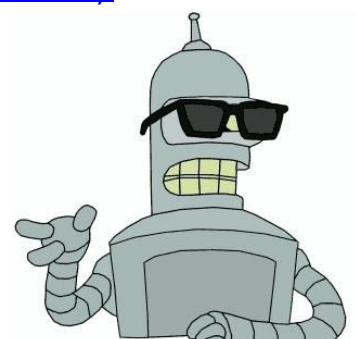   ```
    while (i < 5)
    {
            north 2
            i = i - 1
    }
    ```

**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/

---

[2] Just to be clear: You don't need to do this part again for the SML mandatory.

## 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>

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

// statements control the robot's movement
<stmt> ::=
     <stop>
   | <move>
   | <assignment>
   | <while>
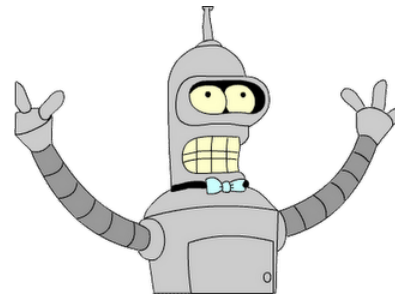
// how to handle the stop statement is up to you, and should be mentioned in the
design document
<stop> ::= **stop**

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

<assignment> ::= <identifier> **=** <exp>

<while> ::= **while (**<boolean-exp>**) {** <stmt>+ **}**

// 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> ::=
     <exp> **>** <exp>
   | <exp> **<** <exp>
   | <exp> **=** <exp>

Legend:
<non-terminal>
**terminal**

<arithmetic-exp> ::=
    <exp> **+** <exp>
  | <exp> - <exp>
  | <exp> * <exp>

**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.

```java
public 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 While 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+4
north 10 + 27
stop
**********************************
The result is (14, 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 + 5 // assume ordinary operator precedence
stop
************************************
The result is (14, 17)


************************************
Testing Code 3: Example with while loop and assignment
size (64, 64)
var i = 5
var j = 3
start (24, 6)
north 3*i
west 15
east 4
while (j>0)
{
   south j
   j = j -1
}
stop
************************************
The result is (13, 15)


************************************
Testing Code 4: Example with movement over the edge
size (64, 64)
var j = 3
start (1, 1)
while (j>0)
{
   north j
}
stop
************************************
```

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