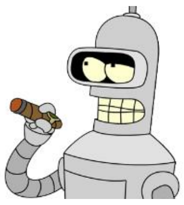
Mandatory Exercise 2 INF3110



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

Deliverables

- A README.txt-file that explains how to run your program. If your program does not work correctly you should also explain this in the README.
- The entire program and README.txt 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/

Optional

There are some extra exercises here (goo.gl/6P3WYk), if you would like to practice your
SML and functional programming a little more before the exam.
These exercises are probably a bit closer to the exercises in the upcoming exam compared to the oblig.
You do not need to deliver your solutions to these exercises and they will not be corrected, but there will be posted a solution.

Deadline: 2018-10-26 23:59.

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 =
   BiggerThan of exp * exp
  | LessThan of exp * exp
…
  | Identifier of string
…

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))) = …
```

Because this is an exercise in functional programming, you should not use any assignment statements in your SML code. Furthermore, since the HashMap in the SML library uses assignments, it is also not allowed to be used.

That means that you can't use mutable references (destructive updates) and assign new values to these.

Like:

```
val mutVar = ref 1;

mutVar := 2;
```

You are of course allowed to use variable bindings as in

```
val x = 1;
```

or

```
let val (a,b) in a+b end;
```

The easiest way to manage your varDecls would then probably be to implement an association list (a list of key-value pairs) with the following functions to operate on the assoc list:

```
type assocList = (string * int) list;
fun lookup key (list:assocList) =
fun add key value (list:assocList) =
fun change key value (list:assocList) = map
```

You can of course choose another way to deal with the varDecls, as long as it is pure (that is, without any assignments).

How to run your program:
Your program should contain one file that runs all 4 test cases when executed, preferably named robol.sml
So :

```
$ sml robol.sml
```

should in some way show the result of all 4 tests

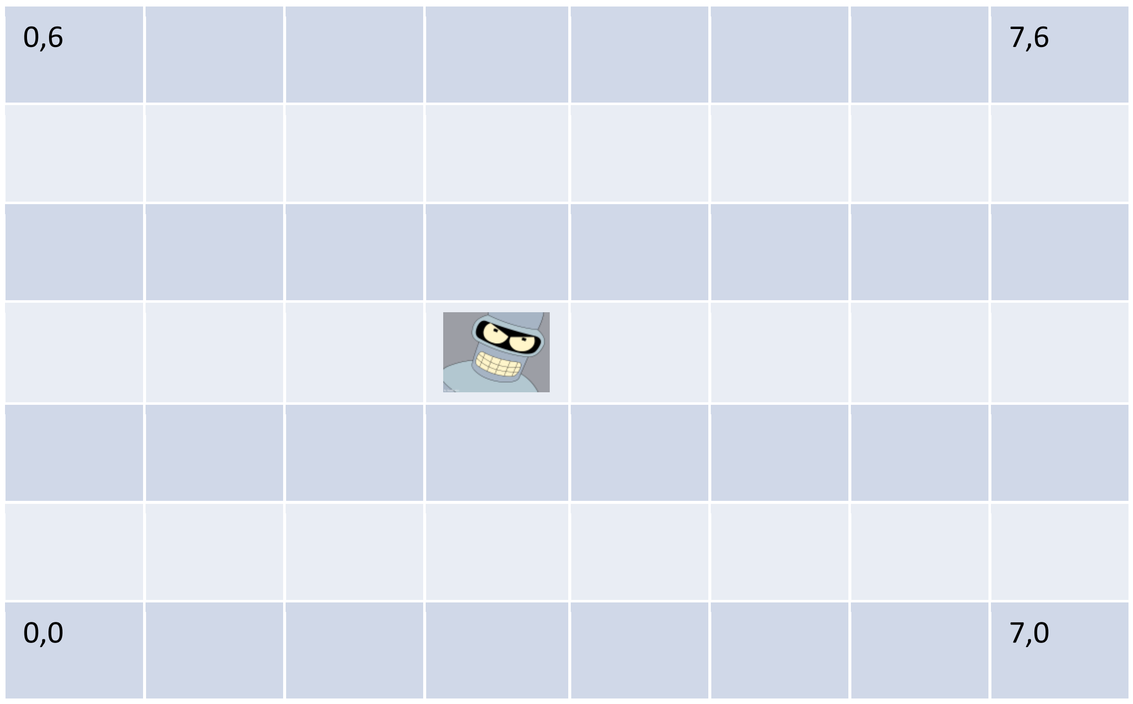
For example:

```
$ sml robol.sml
Standard ML of New Jersey v110.80 [built: Thu Aug 18 15:00:00 2016]
[opening robol.sml]
type number = int
type identifier = string
datatype exp
  = ArithmeticExp of arithmeticExp
.
.
.
.
val it = "The result is (14, 52)" : string
val it = "The result is (14, 17)" : string
val it = "The result is (13, 15)" : string
val it = "BOUNDS OF GRID OVERSTEPPED!!!" : string
```

Attachment: Mandatory 1 revisited

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:



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 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 has already written those for you).

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.

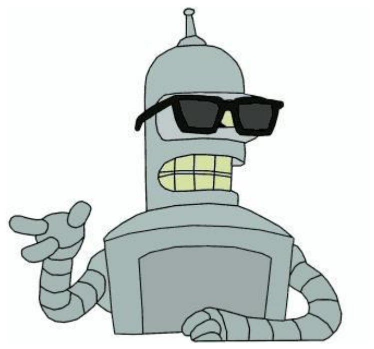
Requirements:

- The interpreter 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.



ROBOL Grammar

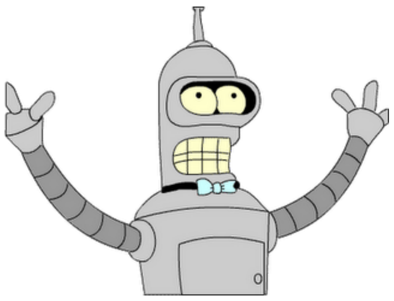
// a program consists of a robot, and a grid on which it can move around
<program> ::= <grid> <robot>

// 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>
  | <while>



// 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>--

<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>| <arithmetic-exp> | <boolean-exp>

//prefix notation. (< 1 2) should be evaluated as 1 < 2
<Boolean-exp> ::=
  (<boolean-op> <exp> <exp>**)**

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

//prefix notation. (+ 1 2 3) should be evaluated as 1 + 2 + 3
<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 try 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.

```
*************************************
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)   5)
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(23,6)
north (* 3 i)
west 15
east 4
while(> j 0)
{
   south j
   j--
}
east 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)
while(> j 0)
{
   north j
}
stop
*************************************
```

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