Running the interpreter with:

```
sml oblig2.sml
```

will produce the following printout:

```
exception OutOfBounds
exception VariableNotFound
datatype direction = EAST | NORTH | SOUTH | WEST
datatype expression
  = Add of expression * expression
  | Bool of boolean
  | ID of string
  | Mul of expression * expression
  | Num of int
  | Sub of expression * expression
datatype boolean
  = Equal of expression * expression
  | Less of expression * expression
  | More of expression * expression
datatype statement
  = Assign of string * expression
  | Move of direction * expression
  | Stop
  | While of boolean * statement list
datatype variable = Var of string * int
datatype declaration = Decl of string * expression
datatype start = Start of expression * expression
datatype board = Size of int * int
datatype robot = Robot of declaration list * start * statement list
datatype program = Program of board * robot
type position = int * int
type state = board * variable list * position
val addVar = fn : variable * variable list -> variable list
val findVar = fn : string * variable list -> int
val evalExpr = fn : variable list * expression -> int
val evalDecls = fn : declaration list * variable list -> variable list
val move = fn : state * int * int -> state
val evalStmts = fn : state * statement list -> state
val evalMove = fn : state * direction * expression -> state
val evalAssign = fn : state * string * expression -> state
val evalWhile = fn : state * boolean * statement list -> state
val interpret = fn : program -> unit
val prog1 = Program (Size (64,64),Robot ([],Start #,[#,#,#,#,#])) : program
val prog2 = Program (Size (64,64),Robot ([#,#],Start #,[#,#,#,#,#])) : program
val prog3 = Program (Size (64,64),Robot ([#,#],Start #,[#,#,#,#,#])) : program
val prog4 = Program (Size (64,64),Robot ([#],Start #,[#,#])) : program
```

Some Peculiarities about My Design

(1) The given Robol grammar clearly specifies that `<arithmetic-exp>` is a subtype of `<exp>`, but I have decided to not create a separate datatype for `<arithmetic-exp>`. The reason is because none of the Robol grammars explicitly requires `<arithmetic-exp>` in order to function properly. On the other hand, I need to define an explicit type for `<boolean-exp>` (in the code, the type name is `boolean`), because it is used as the type for the condition of the while loop expression.

(2) Since an identifier can more or less be used interchangeably as a string, I have decided to use the `ID` constructor only when we want make an `expression`. But when an identifier is explicitly required in a piece of grammar (like in a `<var-decl>` or in an `<assignment>`), then a string will be used instead (to make it less complicated).

(3) There is a difference between a variable and a variable declaration. I define a variable as a tuple of a string and an integer (which specifies the value stored in that variable). On the other hand, a variable declaration is a tuple of a string and an expression (which will be evaluated later on).

```
datatype variable = Var of string * int
datatype declaration = Decl of string * expression
```

(4) Two of the most import type aliases in my code are `position` and `state`. I created the aliases for them because they are used a lot throughout the program. The `state` is very important because we want to change the state every time a statement is evaluated (i.e. any function that evaluates a statement will receive the current state as an input and produce a next state as the output).

```
type position = int * int
type state = board * variable list * position
```

(5) I have implemented the interpreter using a purely functional style without any destructive operations or lazy evaluations. So for example, the `filter` function defined inside the `addVar` function does not use destructive operations to change an existing list, but instead, it builds up a *new* list that excludes the element being filtered. This is inefficient when we deal with a large variable list, but that is just an inherent weakness of functional programming that we can't avoid.

(6) Apart from the `evalMove` function, I have also written another `move` function that explicitly handles the move and raises `OutOfBounds` exception if necessary. This `move` function is very similar to the Java implementation that I have done for the previous oblig. It receives the current state of the program as well as a "vector" that specifies how many steps the robot must move *relative* to the current position. I wrote it as a stand-alone function because it will be used by both the `evalMove` and the `interpret` functions. This is because I also want to test whether the `Start` statement contains a valid start position.

```
val move = fn : state * int * int -> state
```

(7) My implementation for the interpreter allows the possibility of "re-declaring" a variable. For example, the Robol code below will be considered as valid even though the variable `i` is declared twice. This can easily be fixed if I just write another `filter` function that can raise an exception if the variable being filtered out is not there in the list. However, I think this would overcomplicate the assignment. Besides, I think this is a rather cool feature for the Robol language.

```
size(64,64)
var i = 7
var i = 3
start(0,0)
stop
```