Exercise: simple expression evaluation

```
type exp = Operation of exp * binary_operation * exp
              | Id of string | Integer of int
and binary_operation = Equal | LessThan | Plus

type value = Int of int | Bool of bool

let rec eval e dict =
```

(see midterm 1
answer sheet)

```
and apply bop v1 v2 =
```

eval for MJ

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
and state = (varname * value) list
and varname = string


let rec eval (e:exp) (sigma:state) (prog:program) : value = match e with

        Null ->      NullV

      | True ->      BoolV true

      | False ->     BoolV false

      | Integer i ->   IntV i

      | String s ->    StringV s

      (* assume id is in state sigma *)
      | Id id ->      lookup id sigma
```

applyOp for MJ (cont.)

```
type value = IntV of int | StringV of string | BoolV of bool | NullV

    | Operation(e1, bop, e2) ->  (* for non-boolean operations *)
         applyOp bop (eval e1 sigma prog) (eval e2 sigma prog)

let applyOp (bop:binary_operation) (v1:value) (v2:value) : value =
   match bop with
   Multiplication ->  match (v1, v2) with
          (IntV i1, IntV i2) -> IntV (i1 * i2)


   Plus ->   match (v1, v2  with
       (IntV i1, IntV i2) -> IntV (i1 + i2)
     | (StringV s1, StringV s2) -> StringV (s1 ^ s2)

     |    :
```

eval for MJ, with exceptions

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
and state = (varname * value) list
and varname = string
exception TypeError of string
exception RuntimeError of string

    | Id id -> if isin id sigma
               then lookup id sigma
               else raise TypeError

    | Not e -> match (eval e sigma prog) with
          BoolV b -> BoolV (not b)
        | _       -> raise TypeError
```

Ex: SOS for binary operations

(BinopInt) $e_1 + e_2,\sigma,\pi \Downarrow \texttt{IntV}\ (i_1 + i_2)$

$$e_1,\sigma,\pi \Downarrow \texttt{IntV}\ i_1$$

$$e_2,\sigma,\pi \Downarrow \texttt{IntV}\ i_2$$

(BinopInt) $e_1 * e_2,\sigma,\pi \Downarrow \texttt{IntV}\ (i_1 * i_2)$

$$e_1,\sigma,\pi \Downarrow \text{IntV}\ i_1$$

$$e_2,\sigma,\pi \Downarrow \text{IntV}\ i_2$$

(LessThan) $e_1 < e_2,\sigma,\pi \Downarrow$ $\text{BoolV}\ (i_1 < i_2)$

$$e_1,\sigma,\pi \Downarrow \text{IntV}\ i_1$$

$$e_2,\sigma,\pi \Downarrow \text{IntV}\ i_2$$

Ex: SOS for boolean operations

(OrTrue) $e_1 || e_2, \sigma, \pi \Downarrow$ `BoolV true`

$e_1, \sigma, \pi \Downarrow$ `BoolV true`

(OrFalse) $e_1 || e_2, \sigma, \pi \Downarrow$ `BoolV` t

$e_1, \sigma, \pi \Downarrow$ `BoolV false`

$e_2, \sigma, \pi \Downarrow$ `BoolV` t

(AndFalse) $e_1 \&\& e_2, \sigma, \pi \Downarrow$ `BoolV false`

$e_1, \sigma, \pi \Downarrow$ BoolV false

(AndTrue) $e_1 \&\& e_2, \sigma, \pi \Downarrow$ `BoolV` t

$e_1, \sigma, \pi \Downarrow$ BoolV true

$e_2, \sigma, \pi \Downarrow$ BoolV t

(Not) $!e, \sigma, \pi \Downarrow$ `BoolV (not` b`)`

$e, \sigma, \pi \Downarrow$ BoolV b

Statements

- **You will also need to write function** `exec: statement → state → program → state` **to execute some simple statements:**

```
statement = Block of (statement list)
     | If of exp * statement * statement
     | Assignment of id * exp

let rec exec s sigma prog = match s with
       Assignment(s, e) ->
            asgn s (eval e sigma prog) sigma

     | If(e,s1,s2) -> match (eval e sigma prog) with
            BoolV b -> if b then exec s1 sigma prog
                            else exec s2 sigma prog

            | _ -> raise TypeError
```

Method calls

eval clause for method call:

argument list

| MethodCall (_ , m, [e_1; ...; e_n])

① Look up m; suppose it is:

```
t  m (x1, ..., xn) {
   y1; ...; ym; // local vars
   S1; ...; Sn;
   return e; }
```

② Evaluate eval e_1 sigma prog, eval e_2 sigma prog, ..., yielding values $v_1, ..., v_n$

③ Create state pairing x_1 with v_1, x_2 with v_2, etc., and pairing y_1 with NullV, y_2 with NullV, etc. Call this state σ

④ Execute statements $S_1, ..., S_n$, starting in state σ, yielding state σ'

⑤ Evaluate e in σ' and return its value