CSED321 Assignment 4 - *Operational Semantics of IMP*

Due Tuesday, March 26th

Congratulations on finishing the first three assignments of this course! In this assignment, you will implement the operational semantics of the simple imperative language IMP:

$$
\begin{array}{lrcl}
\textit{expression} & e & ::= & n \mid b \mid x \\
 & & \mid & e + e \mid e - e \mid e * e \mid e \,/\, e \\
 & & \mid & e = e \mid e < e \mid e \;\&\&\; e \mid e \;||\; e \mid \,!\, e \\
\textit{statement} & s & ::= & \text{skip} \mid x := e \mid s \,;\, s \\
 & & \mid & \text{if } e \text{ then } s \text{ else } s \text{ fi} \mid \text{if } e \text{ then } s \text{ fi} \\
 & & \mid & \text{while } e \text{ do } s \text{ end} \\
\textit{program} & p & ::= & s \,;\, e
\end{array}
$$

As shown in the above syntax, an IMP program is given by a sequence $s\,;e$ of statement s and expression e, where e represents the return value of the program. For example, the return value of the IMP program "`a := 1; if 0 < a then b := 2 fi; a * b`" is the number 2. The operational semantics of IMP discussed in the class is summarized as follows.

$$\frac{}{\langle v, \sigma\rangle \Downarrow v}\text{Const} \qquad \frac{v = \sigma(x)}{\langle x, \sigma\rangle \Downarrow v}\text{Var} \qquad \frac{\langle e, \sigma\rangle \Downarrow b}{\langle !\, e, \sigma\rangle \Downarrow \neg b}\text{Not}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma\rangle \Downarrow n_1 + n_2}\text{Add} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma\rangle \Downarrow n_1 - n_2}\text{Sub}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 * e_2, \sigma\rangle \Downarrow n_1 * n_2}\text{Mul} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad n_2 \neq 0}{\langle e_1 / e_2, \sigma\rangle \Downarrow n_1 / n_2}\text{Div}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow v \quad \langle e_2, \sigma\rangle \Downarrow v}{\langle e_1 = e_2, \sigma\rangle \Downarrow \textit{true}}\text{EqT} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow v_1 \quad \langle e_2, \sigma\rangle \Downarrow v_2 \quad v_1 \neq v_2}{\langle e_1 = e_2, \sigma\rangle \Downarrow \textit{false}}\text{EqF}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 < e_2, \sigma\rangle \Downarrow n_1 < n_2}\text{Less} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow b_1 \quad \langle e_2, \sigma\rangle \Downarrow b_2}{\langle e_1 \;\&\&\; e_2, \sigma\rangle \Downarrow b_1 \wedge b_2}\text{And}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow b_1 \quad \langle e_2, \sigma\rangle \Downarrow b_2}{\langle e_1 \;||\; e_2, \sigma\rangle \Downarrow b_1 \vee b_2}\text{Or} \qquad \frac{}{\langle \textit{skip}, \sigma\rangle \Downarrow \sigma}\text{Skip}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle x := e, \sigma\rangle \Downarrow \sigma[v/x]}\text{Assign} \qquad \frac{\langle s_1, \sigma\rangle \Downarrow \sigma'' \quad \langle s_2, \sigma''\rangle \Downarrow \sigma'}{\langle s_1 \,;\, s_2, \sigma\rangle \Downarrow \sigma'}\text{Seq}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \textit{true} \quad \langle s_1, \sigma\rangle \Downarrow \sigma'}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma\rangle \Downarrow \sigma'}\text{IteT} \qquad \frac{\langle e, \sigma\rangle \Downarrow \textit{false} \quad \langle s_2, \sigma\rangle \Downarrow \sigma'}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi}, \sigma\rangle \Downarrow \sigma'}\text{IteF}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \textit{true} \quad \langle s, \sigma\rangle \Downarrow \sigma'}{\langle \text{if } e \text{ then } s \text{ fi}, \sigma\rangle \Downarrow \sigma'}\text{ItT} \qquad \frac{\langle e, \sigma\rangle \Downarrow \textit{false}}{\langle \text{if } e \text{ then } s \text{ fi}, \sigma\rangle \Downarrow \sigma}\text{ItF}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \textit{false}}{\langle \text{while } e \text{ do } s \text{ end}, \sigma\rangle \Downarrow \sigma}\text{WhilF} \qquad \frac{\langle e, \sigma\rangle \Downarrow \textit{true} \quad \langle s \,;\, \text{while } e \text{ do } s \text{ end}, \sigma\rangle \Downarrow \sigma'}{\langle \text{while } e \text{ do } s \text{ end}, \sigma\rangle \Downarrow \sigma'}\text{WhilT}$$

The goal of this assignment is not just to implement the operational semantics, but to develop the skill of *interpreting inference rules as algorithms*. That is, given a system of inference rules, we wish to extract an algorithm corresponding to the inference rules. Specifically, we introduce two functions `eval` and `exec` with the following specification:

- `eval` takes a state σ and an expression e. `eval` σ e returns a value v if e evaluates to v in state σ (i.e., $\langle e, \sigma \rangle \Downarrow e'$). Otherwise, `eval` σ e raises an exception.
- `exec` takes a state σ and a statement s. `exec` σ e returns another state σ' if executing s from σ results in σ' (i.e., $\langle s, \sigma \rangle \Downarrow \sigma'$). Otherwise, `exec` σ s raises an exception.

The operational semantics of IMP can be expressed using these functions. For example, consider the case of the rule Assign where `exec` takes σ and $x := e$ as arguments. First, `eval` σ e is called to determine the value that e evaluates to. If `eval` σ e returns v, we return $\sigma[v/x]$. If `eval` σ e raises an exception, we propagate it (by not catching the exception). The other rules can be interpreted in a similar way.

Programming Instruction

Download the zip file `hw4.zip` from the course webpage or by running the command `receive` on the server `141.223.163.223`, and unzip it on your working directory. It will create a bunch of files. First see `imp.ml` in which the abstract syntax of IMP is declared as follows (an IML parser that translates a string to the abstract syntax tree is already given).

```
type id = string

type exp = Num of int
         | Bool of bool
         | Var  of id
         | Add  of exp * exp   | Sub  of exp * exp
         | Mul  of exp * exp   | Div  of exp * exp
         | Eq   of exp * exp   | Less of exp * exp
         | And  of exp * exp   | Or   of exp * exp   | Not  of exp

type stmt = Skip
          | Assign of id * exp
          | Seq    of stmt * stmt
          | If     of exp * stmt * stmt
          | If2    of exp * stmt
          | While  of exp * stmt

type program = stmt * exp
```

The datatype `exp` corresponds to the syntactic category expression, `stmt` corresponds to the syntactic category statement, and `program` corresponds to the syntactic category program. For example, `Var "x"` denotes a variable x, `Bool false` denotes the Boolean constant *false*, `Num 1` denotes the integer constant 1, `Add(Var "x", Num 1)` denotes the expression $x + 1$, and `Assign("x", Num 1)` denotes the statement `x := 1`. The abstract syntax of the IMP program "`a := 1; if 0 < a then b := 2 fi; a * b`" is expressed as the tuple:

```
(Seq(Assign("a", Num 1), If2(Less(Num 0, Var "a"), Assign("b", Num 2))),
 Mul(Var "a", Var "b"))
```

Next see `hw4.ml`. The goal of this assignment is to implement two function `eval` and `exec`, together with properly defining the type `state` and the value `emptystate` for the empty state. The function `run`, which "runs" an IMP program from the empty state and returns the resulting value, is already given by using `eval`, `exec`, and `emptystate`.

```
open Imp

exception NotImplemented

exception RuntimeError of string

type state = unit   (* dummy type, to be chosen by students *)

(* emptystate: state *)
let emptystate = raise NotImplemented

(* eval : state -> exp -> exp *)
let eval _ _ = raise NotImplemented

(* exec : state -> stmt -> state *)
let exec _ _ = raise NotImplemented

(* run : program -> exp *)
let run (s,e) = eval (exec emptystate s) e
```

As explained, your implementation of `eval` and `exec` must raise exceptions if expressions or statements cannot be properly evaluated *according to the operational semantics*. Specifically, `RuntimeError` must be raised for the following cases:

- a value of a variable x is required but not defined in a given state σ (for the rule Var) ;
- an integer is expected but a Boolean is given (for the rules Add, Sub, Mul, Div, Less);
- a Boolean is expected but an integer is given (for the rules Not, And, Or, IteT, IteF, ItT, ItF, WhilF, WhilT); and
- division by zero happens (for the rule Div).

A string argument of `RuntimeError` is to show an informative message and can be arbitrary. It is worth noting that the rules *EqT* and *EqF* can compare values of different types. For example, `1 = true` is a valid expression that evaluates to *false*.

Here are some hints for you. First, a state σ can be considered as a dictionary with variable identifiers as keys, and you have implemented various dictionaries in the second assignment. Second, some semantic rules can be simplified using the program equivalences explained in the class. (e.g., why is `if` e `then` s `fi` a syntactic sugar?) Finally, you can test your solution by using the IMP programs in the `tests` directory.

After implementing the functions `eval` and `exec` in `hw4.ml`, run the command `ocamlbuild` to compile the sources files as follows to create an executable `main.native`:

```
$ ocamlbuild main.native
Finished, 19 targets (0 cached) in 00:00:00.
$ ls
_build   hw4.ml   imp.ml   lexer.mll   main.ml   main.native   parser.mly
```

Now you can run IMP programs in the `tests` directory (or your own IMP program in a file) using the executable `main.native`. For example:

```
$ ./main.native tests/test3.imp
true
$ ./main.native tests/test4.imp
2550
$ ./main.native tests/testerr0.imp
Runtime error: unbound identifier y
$ ./main.native tests/testerr1.imp
Runtime error: expected integer but true
```

Alternatively you can load those functions in the interactive mode of OCaml as follows. In this case, you may directly write data types for the IMP abstract syntax without parsing.

```
$ ocaml
        OCaml version 4.05.0

# #mod_use "imp.ml";;
module Imp :
  sig
    type id = string
    type exp =
        Num of int
      | Bool of bool
    ...
  end
# #use "hw4.ml";;
exception NotImplemented
exception RuntimeError of string
...
# open Imp;;
# eval emptystate (Add(Num 1, Num 2));;
- : Imp.exp = Num 3
# eval emptystate (And(Bool true, Num 1));;
Exception: RuntimeError "expected boolean but 1".
# exec emptystate (Assign("a", Num 1));;
...
```

Submission instruction

1. Make sure that you can compile `hw4.ml` by running `ocamlbuild main.native`.

2. When you have the file `hw4.ml` ready for submission, run the `handin` command in the same directory, and your file will be submitted automatically.