Homework 4
ENE4014 Programming Languages, Spring 2024
due: 5/27(Mon), 23:59

Consider the following programming language, called `miniML`, that features recursive procedures and explicit references.

Syntax The syntax is defined as follows:

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \\
 & | & x \\
 & | & E + E \mid E - E \mid E * E \mid E/E \\
 & | & E = E \mid E < E \\
 & | & \texttt{iszero}\ E \\
 & | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & | & \texttt{let}\ x = E\ \texttt{in}\ E \\
 & | & \texttt{letrec}\ f(x) = E\ \texttt{in}\ E \\
 & | & \texttt{proc}\ x\ E \\
 & | & E\ E \\
 & | & \texttt{ref}\ E \\
 & | & !\ E \\
 & | & E := E \\
 & | & E; E \\
 & | & \texttt{begin}\ E\ \texttt{end}
\end{array}
$$

A program is an expression. Expressions include integers, identifiers, arithmetic expressions, comparisons, conditional expressions, variable or recursive function definitions, function calls, dereferences, assignments, and sequences.

Exercise 1 Consider the following semantics definition with **dynamic scoping**, and implement an interpreter of `miniML`. The semantics is defined with the following domains and evaluation rules.

The set of values (Val) the language manipulate includes integers ($\mathbb{Z}$), booleans ($Bool$), procedures ($Procedure$), and memory locations (Loc). Environments

(*Env*) map program variables (*Var*) to values. Memories (*Mem*) map memory locations (*Loc*) to values. Recall that recursive functions require no special mechanism in dynamic scoping.

$$\begin{array}{rcl} \mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{Loc} \\ \mathit{Procedure} & = & \mathit{Var} \times E \\ \rho \in \mathit{Env} & = & \mathit{Var} \to \mathit{Val} \\ \sigma \in \mathit{Mem} & = & \mathit{Loc} \to \mathit{Val} \end{array}$$

The evaluation rules are defined inductively as inference rules.

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \oplus E_2 \Rightarrow n_1 \oplus n_2, \sigma_2} \quad \oplus \in \{+, -, *\}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \oplus E_2 \Rightarrow n_1 / n_2, \sigma_2} \quad n_2 \neq 0$$

Note that the semantics is defined only when E_1 and E_2 evaluate to integers and that E_1/E_2 is undefined when the value of E_2 is 0 (division-by-zero).

Comparison operators produce boolean values as follows:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 = E_2 \Rightarrow \mathit{true}, \sigma_2} \quad n_1 = n_2$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 = E_2 \Rightarrow \mathit{false}, \sigma_2} \quad n_1 \neq n_2$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 < E_2 \Rightarrow \mathit{true}, \sigma_2} \quad n_1 < n_2$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 < E_2 \Rightarrow \mathit{false}, \sigma_2} \quad n_1 \geq n_2$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{true}, \sigma_1} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{false}, \sigma_1}\ n \neq 0$$

The semantics of conditional, let, letrec, proc, and call expressions are as follows:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{false}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_3 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto v_1]\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E), \sigma}$$

$$\frac{[f \mapsto (x, E_1)]\rho, \sigma_0 \vdash E_2 \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto v]\rho, \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

The semantics of dereference, assignment, and sequence expressions are as follows:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref}\ E \Rightarrow l, [l \mapsto v]\sigma_1} \quad l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash !\ E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, [l \mapsto v]\sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{begin}\ E\ \texttt{end} \Rightarrow v, \sigma_1}$$

Now let's implement the `miniML` interpreter with dynamic scoping in OCaml. In file `lang.ml`, the syntax is defined as OCaml datatype as follows:

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | EQ of exp * exp
  | LT of exp * exp
  | ISZERO of exp
  | READ
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
  | NEWREF of exp
  | DEREF of exp
  | SETREF of exp * exp
  | SEQ of exp * exp
  | BEGIN of exp
and var = string
```

The type of values, environments, and memory states are defined in the `interpreter_dynamic.ml` file as follows:

```
type value =
    Int of int
  | Bool of bool
  | Procedure of var * exp
  | Loc of loc
and loc = int
and env = (var * value) list
and mem = (loc * value) list
```

According to the above information, implement the function

```
eval :  exp -> env -> mem -> value * mem
```

in the `interpreter_dynamic.ml` file. The function takes a program along with initial environment and memory state, and produces a value and a (possibly modified) memory state. Raise an exception `UndefinedSemantics` (defined in `lang.ml`) whenever the semantics is undefined. Skeleton code will be provided (before you start, see `README.md`).

Exercise 2 Consider the following semantics definition with **static scoping**, and implement an interpreter of `miniML`.

The semantics is defined with the following domain. Recall that recursive functions require special mechanism in static scoping.

$$\begin{array}{rcl} \mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} + \mathit{Loc} \\ \mathit{Procedure} & = & \mathit{Var} \times E \times \mathit{Env} \\ \mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times E \times \mathit{Env} \\ \rho \in \mathit{Env} & = & \mathit{Var} \to \mathit{Val} \\ \sigma \in \mathit{Mem} & = & \mathit{Loc} \to \mathit{Val} \end{array}$$

The followings are evaluation rules (rules same as in the previous exercise are omitted):

$$\frac{[f \mapsto (f, x, E_1, \rho)]\rho, \sigma_0 \vdash E_2 \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_1}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto v]\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (f, x, E, \rho'), \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \quad [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

Now let's implement the `miniML` interpreter with static scoping in OCaml. We will use the syntax defined as OCaml datatype in file `lang.ml` as we do in the previous exercise. The type of values, environments, and memory states are defined in the
`interpreter_static.ml` file as follows:

```
type value =
    Int of int
  | Bool of bool
  | Procedure of var * exp * env
  | RecProcedure of var * var * exp * env
  | Loc of loc
and loc = int
and env = (var * value) list
and mem = (loc * value) list
```

According to the aforementioned evaluation rules, implement the function

```
eval :  exp -> env -> mem -> value * mem
```

in the `interpreter_static.ml` file. Raise an exception `UndefinedSemantics` whenever the semantics is undefined.