Homework 2
COSE212, Fall 2024

Hakjoo Oh

Due: 10/6, 23:59

Problem 1 Let us design and implement a programming language called ML^-. ML^- is a small yet Turing-complete functional language that supports built-in lists and (mutually) recursive procedures.

Language Design The syntax of ML^- is defined inductively as follows:

$$
\begin{array}{rcll}
P & \rightarrow & E & \\
E & \rightarrow & \texttt{()} & \text{unit} \\
 & | & \texttt{true} \mid \texttt{false} & \text{booleans} \\
 & | & n & \text{integers} \\
 & | & x & \text{variables} \\
 & | & E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E & \text{arithmetic} \\
 & | & E \texttt{ = } E \mid E \texttt{ < } E & \text{comparison} \\
 & | & \texttt{not } E & \text{negation} \\
 & | & \texttt{nil} & \text{empty list} \\
 & | & E \texttt{ :: } E & \text{list cons} \\
 & | & E \texttt{ @ } E & \text{list append} \\
 & | & \texttt{head } E & \text{list head} \\
 & | & \texttt{tail } E & \text{list tail} \\
 & | & \texttt{isnil } E & \text{checking empty list} \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E & \text{if} \\
 & | & \texttt{let } x \texttt{ = } E \texttt{ in } E & \text{let} \\
 & | & \texttt{letrec } f(x) \texttt{ = } E \texttt{ in } E & \text{recursion} \\
 & | & \texttt{letrec } f(x_1) = E_1 \text{ and } g(x_2) = E_2 \texttt{ in } E & \text{mutual recursion} \\
 & | & \texttt{proc } x\ E & \text{function definition} \\
 & | & E\ E & \text{function application} \\
 & | & \texttt{print } E & \text{print} \\
 & | & E; E & \text{sequence}
\end{array}
$$

The semantics of the language is similar to that of OCaml. The set of values the language manipulate includes unit $(\cdot)$, integers $(\mathbb{Z})$, booleans (*Bool*), lists (*List*), non-recursive procedures (*Procedure*), recursive procedures (*RecProcedure*), and mutually recursive procedures (*MRecProcedure*):

$$\begin{array}{rcl}
v \in \mathit{Val} & = & \{\cdot\} + \mathbb{Z} + \mathit{Bool} + \mathit{List} + \\
& & \mathit{Procedure} + \mathit{RecProcedure} + \mathit{MRecProcedure} \\
n \in \mathbb{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in \mathit{Bool} & = & \{\mathit{true}, \mathit{false}\} \\
s \in \mathit{List} & = & \mathit{Val}^* \\
\mathit{Procedure} & = & \mathit{Var} \times E \times \mathit{Env} \\
\mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times E \times \mathit{Env} \\
\mathit{MRecProcedure} & = & (\mathit{Var} \times \mathit{Var} \times E) \times (\mathit{Var} \times \mathit{Var} \times E) \times \mathit{Env}
\end{array}$$

Notations for list values need explanation. We write Val^* for the set of ordered sequences of values. We write $[]$ for the empty sequence. Given a value v and a sequence s, $v :: s$ denotes the sequence that is obtained by inserting v into the front of s. Given two sequences s_1 and s_2, we write $s_1 @ s_2$ for the concatenation of s_1 and s_2.

Environments (*Env*) map program variables (*Var*) to values.

$$\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Val}$$

The semantics rules are defined inductively as inference rules. Rules for constant expressions:

$$\frac{}{\rho \vdash \texttt{()} \Rightarrow \cdot} \qquad \frac{}{\rho \vdash \texttt{true} \Rightarrow \mathit{true}} \qquad \frac{}{\rho \vdash \texttt{false} \Rightarrow \mathit{false}} \qquad \frac{}{\rho \vdash n \Rightarrow n}$$

The value of a variable can be found from the current environment:

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

Arithmetic operations produce integers:

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ / } E_2 \Rightarrow n_1 / n_2}\ n_2 \neq 0$$

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 and negation produce boolean values:

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}}\ n_1 = n_2 \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}}\ n_1 \neq n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}}\ b_1 = b_2 \qquad \frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}}\ b_1 \neq b_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{true}}\ n_1 < n_2 \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{false}}\ n_1 \geq n_2$$

$$\frac{\rho \vdash E \Rightarrow \mathit{true}}{\rho \vdash \texttt{not}\ E \Rightarrow \mathit{false}} \qquad \frac{\rho \vdash E \Rightarrow \mathit{false}}{\rho \vdash \texttt{not}\ E \Rightarrow \mathit{true}}$$

Note that equality (E_1 `=` E_2) is undefined for list and function values. We deliberately choose to disallow list values; this is our choice for simplicity of the language semantics. However, comparing functional values is undecidable and cannot be implemented in programming languages.

Lists can be constructed in three ways:

$$\frac{}{\rho \vdash \texttt{nil} \Rightarrow []} \qquad \frac{\rho \vdash E_1 \Rightarrow v \qquad \rho \vdash E_2 \Rightarrow s}{\rho \vdash E_1\ \texttt{::}\ E_2 \Rightarrow v :: s} \qquad \frac{\rho \vdash E_1 \Rightarrow s_1 \qquad \rho \vdash E_2 \Rightarrow s_2}{\rho \vdash E_1\ \texttt{@}\ E_2 \Rightarrow s_1 @ s_2}$$

where v and s denote an arbitrary value and a list value, respectively. Other list operations are defined as follows:

$$\frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{head}\ E \Rightarrow v} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{tail}\ E \Rightarrow s}$$

$$\frac{\rho \vdash E \Rightarrow []}{\rho \vdash \texttt{isnil}\ E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{isnil}\ E \Rightarrow \mathit{false}}$$

We defined the semantics of conditional, let, letrec, proc, and call expressions in class as follows:

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v} \qquad \frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

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

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

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

$$\frac{[f \mapsto (f, x, E_1, g, y, E_2, \rho), g \mapsto (g, y, E_2, f, x, E_1, \rho)]\rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{and}\ g(y) = E_2\ \texttt{in}\ E_3 \Rightarrow v}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E_f, g, y, E_g, \rho') \quad \rho \vdash E_2 \Rightarrow v \\ \left\{\begin{array}{l} x \mapsto v, \\ f \mapsto (f, x, E_f, g, y, E_g, \rho'), \\ g \mapsto (g, y, E_g, f, x, E_f, \rho') \end{array}\right\} \rho' \vdash E_f \Rightarrow v' \end{array}}{\rho \vdash E_1\ E_2 \Rightarrow}$$

The expression `print` E prints the value of E and then produces a unit value:

$$\frac{}{\rho \vdash \texttt{print}\ E \Rightarrow \cdot}$$

The sequence expression $E_1; E_2$ evaluates E_1 and E_2 while ignoring the value of E_1:

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1; E_2 \Rightarrow v_2}$$

Language Implementation Now, let us implement ML^{-}. In OCaml, the syntax is defined as datatype as follows:

```
type program = exp
and exp =
  | UNIT
  | TRUE
  | FALSE
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | EQUAL of exp * exp
  | LESS of exp * exp
  | NOT of exp
  | NIL
  | CONS of exp * exp
  | APPEND of exp * exp
  | HEAD of exp
  | TAIL of exp
  | ISNIL of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | LETMREC of (var * var * exp) * (var * var * exp) * exp
  | PROC of var * exp
  | CALL of exp * exp
  | PRINT of exp
  | SEQ of exp * exp
and var = string
```

The type of values and environments are defined as follows:

```
type value =
  | Unit
  | Int of int
  | Bool of bool
  | List of value list
  | Procedure of var * exp * env
  | RecProcedure of var * var * exp * env
```

```
  | MRecProcedure of var * var * exp *
                     var * var * exp * env
and env = (var * value) list
```

Implement the function `runml`:

```
runml : program -> value
```

which takes a program, evaluates it, and produces its value. Whenever the semantics is undefined, raise exception `UndefinedSemantics`.

Examples Check your implementation by running the following example programs (and more).

1. Evaluating the program

   ```
   let x = 1
   in let f = proc (y) (x + y)
     in let x = 2
        in let g = proc (y) (x + y)
           in  (f 1) + (g 1)
   ```

   represented by

   ```
     LET ("x", CONST 1,
      LET ("f", PROC ("y", ADD (VAR "x", VAR "y")),
       LET ("x", CONST 2,
        LET ("g", PROC ("y", ADD (VAR "x", VAR "y")),
         ADD (CALL (VAR "f", CONST 1), CALL (VAR "g", CONST 1))))))
   ```

   should produce the value `Int 5`.

2. Evaluating the program

   ```
   letrec double(x) = if (x = 0) then 0 else (double (x-1) + 2
   in (double 6)
   ```

   represented by

   ```
     LETREC ("double", "x",
      IF (EQUAL (VAR "x", CONST 0), CONST 0,
       ADD (CALL (VAR "double", SUB (VAR "x", CONST 1)), CONST 2)),
      CALL (VAR "double", CONST 6))
   ```

   should produce `Int 12`.

3. Evaluating the program

```
letrec even(x) = if (x = 0) then true else odd(x-1)
       odd(x) = if (x = 0) then false else even(x-1)
in (even 13)
```

represented by

```
LETMREC
 (("even", "x",
   IF (EQUAL (VAR "x", CONST 0), TRUE,
    CALL (VAR "odd", SUB (VAR "x", CONST 1)))),
 ("odd", "x",
  IF (EQUAL (VAR "x", CONST 0), FALSE,
   CALL (VAR "even", SUB (VAR "x", CONST 1)))),
 CALL (VAR "odd", CONST 13))
```

should produce `Bool true`.

4. Evaluating the program

```
letrec factorial(x) =
         if (x = 0) then 1
         else factorial(x-1) * x
in letrec loop n =
     if (n = 0) then ()
     else (print (factorial n); loop (n-1))
   in (loop 10)
```

represented by

```
LETREC ("factorial", "x",
 IF (EQUAL (VAR "x", CONST 0), CONST 1,
  MUL (CALL (VAR "factorial", SUB (VAR "x", CONST 1)), VAR "x")),
 LETREC ("loop", "n",
  IF (EQUAL (VAR "n", CONST 0), UNIT,
   SEQ (PRINT (CALL (VAR "factorial", VAR "n")),
    CALL (VAR "loop", SUB (VAR "n", CONST 1)))),
  CALL (VAR "loop", CONST 10)))
```

should produce `Unit` after printing out the following lines:

```
3628800
362880
40320
5040
720
120
```

```
24
6
2
1
```

5. Evaluating the program

   ```
   letrec range(n) =
         if (n = 1) then (cons 1 nil)
         else n::(range (n-1))
   in (range 10)
   ```

   represented by

   ```
     LETREC ("range", "n",
      IF (EQUAL (VAR "n", CONST 1), CONS (CONST 1, NIL),
       CONS (VAR "n", CALL (VAR "range", SUB (VAR "n", CONST 1)))),
      CALL (VAR "range", CONST 10))
   ```

   should produce `List [Int 10; Int 9; Int 8; Int 7; Int 6; Int 5; Int 4; Int 3; Int 2; Int 1]`.

6. Evaluating the program

   ```
   letrec reverse(l) =
     if (isnil l) then []
     else (reverse (tl l)) @ (cons hd l)
   in (reverse (cons (1, cons (2, cons (3, nil)))))
   ```

   represented by

   ```
     LETREC ("reverse", "l",
      IF (ISNIL (VAR "l"), NIL,
       APPEND (CALL (VAR "reverse", TAIL (VAR "l")), CONS (HEAD (VAR "l"), NIL))),
      CALL (VAR "reverse", CONS (CONST 1, CONS (CONST 2, CONS (CONST 3, NIL)))))
   ```

   should produce `List [Int 3; Int 2; Int 1]`.

7. An interesting fact in programming languages is that any recursive function can be defined in terms of non-recursive functions (i.e., `letrec` is syntactic sugar[1] in ML^-). Consider the following function:

   ```
   let fix = proc (f) ((proc (x) f (proc (y) ((x x) y)))
                       (proc (x) f (proc (y) ((x x) y))))
   ```

---

[1] `https://en.wikipedia.org/wiki/Syntactic_sugar`

which is called fixed-point-combinator (or Z-combinator).[2] Note that `fix` is a non-recursive function, although its structure is complex and repetitive. Any recursive function definition of the form:

```
letrec f(x) = <body of f> in ...
```

can be defined as follows using `fix`:

```
let f = fix (proc (f) (proc (x) (<body of f>))) in ...
```

For example, the factorial program

```
letrec f(x) = if (x = 0) then 1 else f(x-1) * x
in (f 10)
```

can be defined using `fix`:

```
let fix = proc (f) ((proc (x) f (proc (y) ((x x) y)))
                    (proc (x) f (proc (y) ((x x) y))))
  in let f = fix (proc (f) (proc (x) (if (x = 0) then 1 else f(x-1) * x)))
     in (f 10)
```

which is represented in our implementation as follows:

```
  LET ("fix",
   PROC ("f",
    CALL
     (PROC ("x",
       CALL (VAR "f", PROC ("y", CALL (CALL (VAR "x", VAR "x"), VAR "y")))),
      PROC ("x",
       CALL (VAR "f", PROC ("y", CALL (CALL (VAR "x", VAR "x"), VAR "y")))))),
   LET ("f",
    CALL (VAR "fix",
     PROC ("f",
      PROC ("x",
       IF (EQUAL (VAR "x", CONST 0), CONST 1,
        MUL (CALL (VAR "f", SUB (VAR "x", CONST 1)), VAR "x"))))),
    CALL (VAR "f", CONST 10)))
```

Evaluating this program with your interpreter should produce `Int 3628800`.

For another example, consider the function `range` defined above:

```
in letrec range(n) = if (n = 1) then (cons 1 nil)
                     else n::(range (n-1))
in (range 10)
```

---

[2] `https://en.wikipedia.org/wiki/Fixed-point_combinator`

We can translate it to a non-recursive version as follows:

```
let fix = proc (f) ((proc (x) f (proc (y) ((x x) y)))
                    (proc (x) f (proc (y) ((x x) y))))
  in let f = fix (proc (range)
                   (proc (n)
                     (if (n = 1) then (cons 1 nil)
                      else n::(range (n-1)))))
     in (f 10)
```

In OCaml:

```
  LET ("fix",
   PROC ("f",
    CALL
     (PROC ("x",
       CALL (VAR "f", PROC ("y", CALL (CALL (VAR "x", VAR "x"), VAR "y")))),
     PROC ("x",
      CALL (VAR "f", PROC ("y", CALL (CALL (VAR "x", VAR "x"), VAR "y")))))),
   LET ("f",
    CALL (VAR "fix",
     PROC ("range",
      PROC ("n",
       IF (EQUAL (VAR "n", CONST 1), CONS (CONST 1, NIL),
        CONS (VAR "n", CALL (VAR "range", SUB (VAR "n", CONST 1))))))),
    CALL (VAR "f", CONST 10)))
```

Evaluating this program should produce `List [Int 10; Int 9; Int 8; Int 7; Int 6; Int 5; Int 4; Int 3; Int 2; Int 1]`.