Homework 3
COSE212, Fall 2015

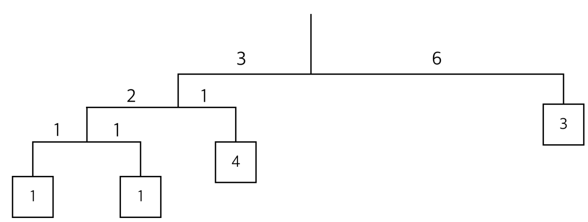
Hakjoo Oh

Due: 10/23, 24:00

Problem 1 A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. In OCaml datatype, a binary mobile can be defined as follows:

```
type mobile = branch * branch     (* left and rigth branches *)
and branch = SimpleBranch of length * weight
           | CompoundBranch of length * mobile
and length = int
and weight = int
```

A branch is either a simple branch, which is constructed from a length together with a weight, or a compound branch, which is constructed from a length together with another mobile. For instance, the mobile



is represented by the following:

```
(CompoundBranch (3,
  (CompoundBranch (2, (SimpleBranch (1, 1), SimpleBranch (1, 1))),
   SimpleBranch (1, 4))),
 SimpleBranch (6, 3))
```

Define the function

```
balanced : mobile -> bool
```

that tests whether a binary mobile is balanced. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. For example, the example mobile above is balanced.

Problem 2 Consider the following language:

```
type exp = V of var
         | P of var * exp
         | C of exp * exp
and var = string
```

In this language[1], a program is simply a variable, a procedure, or a procedure call.

Write a checker function

`check : exp -> bool`

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- `P ("a", V "a")`
- `P ("a", P ("a", V "a"))`
- `P ("a", P ("b", C (V "a", V "b")))`
- `P ("a", C (V "a", P ("b", V "a")))`

Ill-formed ones are:

- `P ("a", V "b")`
- `P ("a", C (V "a", P ("b", V "c")))`
- `P ("a", P ("b", C (V "a", V "c")))`

[1] This language is called "lambda calculus". Even though it is very simple, the language is Turing-complete and all OCaml programs can be reduced to an equivalent program in lambda calculus.

Problem 3 Write an interpreter for the LETREC language:

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & \texttt{zero?}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{letrec}\ f(x) = E\ \texttt{in}\ E \\
& | & \texttt{proc}\ x\ E \\
& | & E\ E
\end{array}
$$

The semantics of the language is defined in Figure 1. In OCaml, the program syntax and the semantic domain for the language can be defined as follows:

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
type value = Int of int | Bool of bool
           | Procedure of var * exp * env
           | RecProcedure of var * var * exp * env
and env = var -> value
```

Define the function `run`:

```
run : program -> value
```

For instance, the program

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

is represented by the OCaml representation:

Domain:

$$
\begin{array}{rcl}
\mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\
\mathit{Procedure} & = & \mathit{Var} \times E \times \mathit{Env} \\
\mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times E \times \mathit{Env} \\
\mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Val}
\end{array}
$$

Semantics rules:

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

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

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

$$
\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{zero?}\ E \Rightarrow \mathit{true}}
$$

$$
\frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{zero?}\ E \Rightarrow \mathit{false}}\ n \neq 0
$$

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

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

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

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

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

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

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

Figure 1: Semantics of the LETREC language.

```
let pgm =
  LETREC ("double", "x", IF (ISZERO (VAR "x"),
                             CONST 0,
                             ADD (CALL (VAR "double", SUB (VAR "x", CONST 1)) ,
                                  CONST 2)),
    CALL (VAR "double", CONST 6))
```

The result of `run pgm` should be `Int 12`.

Problem 4 Consider the Proc language:

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

The nameless representation of the language ("Nameless Proc") can be defined as follows:

```
type nl_program = nl_exp
and nl_exp =
  | NL_CONST of int
  | NL_VAR of int
  | NL_ADD of nl_exp * nl_exp
  | NL_SUB of nl_exp * nl_exp
  | NL_ISZERO of nl_exp
  | NL_IF of nl_exp * nl_exp * nl_exp
  | NL_LET of nl_exp * nl_exp
  | NL_PROC of nl_exp
  | NL_CALL of nl_exp * nl_exp
```

Define the function

```
translate : program -> nl_program
```

that transforms a given Proc program into its nameless representation. For instance, the program

```
LET ("x", CONST 37,
  PROC ("y", LET ("z", SUB (VAR "y", VAR "x"),
    SUB (VAR "x", VAR "y"))))
```

should be translated to the nameless program:

```
NL_LET (NL_CONST 37,
  NL_PROC (NL_LET (NL_SUB (NL_VAR 0, NL_VAR 1),
    NL_SUB (NL_VAR 2, NL_VAR 1))))
```

Problem 5 Write an interpreter for "Nameless Proc":

```
nl_run : nl_program -> nl_value
```

The values and environments are defined as follows:

```
type nl_value = NL_Int of int
              | NL_Bool of bool
              | NL_Procedure of nl_exp * nl_env
and nl_env = nl_value list
```

The semantics of Nameless Proc is formalized in lecture slides.

With the interpreter for Nameless Proc, you can run an ordinary Proc program by first translating it into an equivalent nameless program and then running the resulting program using `nl_run`. For instance, the Proc program

```
let pgm = LET ("x", CONST 1, VAR "x")
```

can be evaluated as follows:

```
nl_run (translate pgm)
```

which should produce `NL_Int 1`.

How to submit

1. Download the homework 3 template file (`hw3.ml`) from the course webpage: `http://prl.korea.ac.kr/~hakjoo/home/courses/cose212/2015`
2. Replace all `(* TODO *)` in `hw3.ml` by your own code. You can define any helper functions in `hw3.ml`.
3. Submit the single file `hw3.ml` via Blackboard.