CSE-321 Assignment 6 (100 points)

Due at 11:59pm, April 21

In this assignment, we use TML (Typed ML) introduced in the previous assignment.

This assignment consists of three parts. First you will implement a function that translates a TML expression into an expression with de Bruijn indexes. Then you will implement an evaluator that follows the call-by-value strategy. Finally you will design and implement an abstract machine based on the call-by-need strategy which is a variant of the call-by-name strategy.

The abstract syntax for TML is shown below.

type	A	::=	$\mathsf{bool} \mid \mathsf{int} \mid A \to A \mid A \times A \mid \mathsf{unit} \mid A + A$
expression	e	::=	$x \mid \lambda x{:}A.\, e \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid$
			$\mathsf{inl}_A\ e \mid \mathsf{inr}_A\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\, e \mid \mathsf{inr}\ x.\, e \mid \mathsf{fix}\ x{:}A.\, e \mid$
			$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \widehat{n} \mid \mathsf{plus} \mid \mathsf{minus} \mid \mathsf{eq}$
typing context	Γ	::=	$\cdot \mid \Gamma, x : A$

The concrete syntax for TML is as follows. All entries in the right side of the definition below are arranged in the same order that their counterparts in the abstract syntax appear in the definition above:

type	A	::=	`bool` $\mid$ `int` $\mid A$ `->` $A \mid A$ `*` $A \mid$ `unit` $\mid A$ `+` $A \mid$ `(`A`)`
expression	e	::=	$x \mid$ `fn` x `:` A `=>` $e \mid e\ e \mid$ `(`e`,` e`)` $\mid$ `fst` $e \mid$ `snd` $e \mid$ `()` $\mid$
			`inl (`A`)` $e \mid$ `inr (`A`)` $e \mid$ `case` e `of inl` x `=>` e `|` `inr` x `=>` $e \mid$
			`fix` x `:` A `=>` $e \mid$
			`true` $\mid$ `false` $\mid$ `if` e `then` e `else` $e \mid \widehat{n} \mid$ `+` $\mid$ `-` $\mid$ `=` $\mid$
			`let` x`:`A `=` e `in` $e' \mid$ `let rec` x`:`A `=` e `in` $e' \mid$
			`(`e`)`
integer	$\widehat{n}$	::=	`0` $\mid$ `1` $\mid$ `2` $\mid \cdots$

We add two forms of syntactic sugar:

`let` x`:`A `=` e `in` e'	for	`(fn` x`:`A `=>` e'`)` e
`let rec` x`:`A `=` e `in` e'	for	`(fn` x`:`A `=>` e'`) (fix` x`:`A`.` e`)`

This assignment does not use the type system of TML because we are interested only in the evaluation part of TML.

1 Translating into de Bruijn indexes (20 pts)

The goal of this part is to implement a function that translates a TML expression into an expression with de Bruijn indexes. For example, it converts $\lambda x{:}A.\, \lambda y{:}B.\, x\ y$ into $\lambda.\lambda.1\ 0$. The abstract syntax for TML with de Bruijn indexes is shown below. Note that this syntax does not use types at all. In fact, we do not have to keep types because types play no role in evaluating.

expression	e	::=	$n \mid \lambda.e \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid$
			$\mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{fix}\ e \mid$
			$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \widehat{n} \mid \mathsf{plus} \mid \mathsf{minus} \mid \mathsf{eq}$

Be careful about distinguishing between n and $\widehat{n}$. n stands for a de Bruijn index, but $\widehat{n}$ stands for a mathematical integer n.

With only this syntax, however, we cannot handle expressions containing free variables. Thus we need a special method to handle it. Consider the following expression:

$$\lambda x{:}A.\, x\ a$$

In this expression, a is a free variable with no binder and cannot be assigned a de Bruijn index. To deal with such an expression, we first have to find free variables in an expression, and then assign de Bruijn indexes to free variables. In order to keep information about what index to assign to each free variable, we introduce a *naming context* which is similar to a typing context. A typing context informs us of the type of a given free variable. Similarly a naming context informs us of the de Bruijn index of a given free variable. Here is the definition of naming contexts where x stands for variables of TML and n stands for de Bruijn indexes:

$$\textsf{naming context} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x \to n$$

For the sake of consistency, we assume that the rightmost free variable in an expression is given the lowest index and the leftmost free variable the highest. For example, consider the following expression:

$$\lambda x{:}A.\, a\ x\ b\ c$$

where a, b, and c are free variables. Since c is the rightmost and a is the leftmost, the naming context of the above expression is $\Gamma = a \to 2, b \to 1, c \to 0$. Then we can translate the above expression as follows:

$$\lambda.3\ 0\ 2\ 1$$

In the same way, $\lambda x{:}A.\, \lambda y{:}B.\, a\ x\ b\ c$ is translated into $\lambda.\lambda.4\ 1\ 3\ 2$, and $\lambda x{:}A.\, \lambda y{:}B.\, a\ b$ is translated into $\lambda.\lambda.3\ 2$. In the case that a free variable appears more than twice, the position of the rightmost free variable determines its index. For example, $a\ b\ c\ a$ is translated into $0\ 2\ 1\ 0$ and $a\ b\ c\ c\ b$ is translated into $2\ 0\ 1\ 1\ 0$.

Programming instruction

Download `hw6.zip` from the course webpage or the handin directory, and unzip it on your working directory. It will create a bunch of files on the working directory.

First see the structure `Tml` in `tml.ml` which has signature `TML`.

```
type var = string
type index = int
type tp =
  ...
type texp =
    Tvar of var
  | Tlam of var * tp * texp
  ...
type exp =
    Ind of index
  | Lam of exp
  ...
```

The datatypes `tp` and `texp` correspond to the syntactic categories type and expression of the abstract syntax for TML, respectively. `exp` corresponds to the expression of the abstract syntax for TML with de Bruijn indexes. Read the comments in the file and make sure that you understand what each data constructor is for.

Next see `eval.mli` and `eval.ml`. The goal is to implement the function `texp2exp` in the structure `Eval`:

```
val texp2exp : Tml.texp -> Tml.exp
```

`texp2exp` takes a TML expression of type `Tml.texp` and returns its translated expression of type `Tml.exp`.

You have to think about how to represent the naming context which is required to implement this function. Think about it and create your own naming context. Then implement the function `texp2exp`

correctly. Otherwise never touch the other two parts of this assignment because the other parts depend on this function.

After implementing the function `step` in `eval.ml`, run the command `make` to compile the sources files.

```
$ make
ocamlc -thread  -c tml.ml -o tml.cmo
ocamlc -thread  -c heap.ml -o heap.cmo
ocamlyacc parser.mly
26 shift/reduce conflicts.
ocamlc -c parser.mli
ocamlc -c parser.ml
ocamllex lexer.mll
67 states, 4551 transitions, table size 18606 bytes
ocamlc -c lexer.ml
ocamlc -thread  -c inout.ml -o inout.cmo
ocamlc -thread  -c eval.mli -o eval.cmi
ocamlc -thread  -c eval.ml -o eval.cmo
ocamlc -thread  -c loop.ml -o loop.cmo
ocamlc -thread  -c hw6.ml -o hw6.cmo
ocamlc -o lib.cma -a tml.cmo heap.cmo parser.cmo lexer.cmo inout.cmo eval.cmo loop.cmo
ocamlc -o hw6 lib.cma hw6.cmo
```

There are two ways to test your code in `eval.ml`. First you can run `hw6`. At the TML prompt, enter a TML expression followed by the semicolon symbol `;`. (The syntax of TML will be given shortly.) Each time you press the return key, a reduced expression is displayed.

```
$ ./hw6
Tml>:(fn x : int => x y z);
(lam. ((0 2) 1))
Press return:
Tml>:if a then b else c a;
(if 0 then 2 else (1 0))
Press return:
```

Alternatively you can use those functions in `loop.ml` in the interactive mode of OCAML. (You don't actually need to read `loop.ml`.) At the OCAML prompt, type `#load ''lib.cma'';;` to load the library for this assignment. Then open the structure `Loop`:

```
$ ocaml
        OCaml version 4.05.0

# #load "lib.cma";;
# open Loop;;
#
```

You type `loop (step (wait show));;` at the OCAML prompt, and then enter a TML expression followed by the semicolon symbol `;`.

```
# loop (step1 (wait1 show1));;
Tml>:(fn x : int => x y z);
(lam. ((0 2) 1))
Press return:
Tml>:if a then b else c a;
(if 0 then 2 else (1 0))
Press return:
```

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \quad \frac{e_2 \mapsto e_2'}{(\lambda x{:}A.\, e)\ e_2 \mapsto (\lambda x{:}A.\, e)\ e_2'}\ \mathit{Arg} \quad \frac{}{(\lambda x{:}A.\, e)\ v \mapsto [v/x]e}\ \mathit{App}$$

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\ \mathit{Pair} \quad \frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')}\ \mathit{Pair}' \quad \frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto \mathsf{fst}\ e'}\ \mathit{Fst} \quad \frac{}{\mathsf{fst}\ (v_1, v_2) \mapsto v_1}\ \mathit{Fst}'$$

$$\frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto \mathsf{snd}\ e'}\ \mathit{Snd} \quad \frac{}{\mathsf{snd}\ (v_1, v_2) \mapsto v_2}\ \mathit{Snd}' \quad \frac{e \mapsto e'}{\mathsf{inl}_A\ e \mapsto \mathsf{inl}_A\ e'}\ \mathit{Inl} \quad \frac{e \mapsto e'}{\mathsf{inr}_A\ e \mapsto \mathsf{inr}_A\ e'}\ \mathit{Inr}$$

$$\frac{e \mapsto e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ \mathit{If}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ \mathit{If}_{\mathsf{true}} \quad \frac{}{\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ \mathit{If}_{\mathsf{false}}$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto \mathsf{case}\ e'\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2}\ \mathit{Case}$$

$$\frac{}{\mathsf{case\ inl}_A\ v\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_1]e_1}\ \mathit{Case}'$$

$$\frac{}{\mathsf{case\ inr}_A\ v\ \mathsf{of\ inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_2]e_2}\ \mathit{Case}''$$

$$\frac{e_1 = \mathsf{plus\ or\ minus\ or\ eq} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \mathit{Arith}$$

$$\frac{}{\mathsf{plus}\ (\widehat{n_1}, \widehat{n_2}) \mapsto \widehat{n_1 + n_2}}\ \mathit{Plus} \quad \frac{}{\mathsf{minus}\ (\widehat{n_1}, \widehat{n_2}) \mapsto \widehat{n_1 - n_2}}\ \mathit{Minus}$$

$$\frac{\widehat{n_1} = \widehat{n_2}}{\mathsf{eq}\ (\widehat{n_1}, \widehat{n_2}) \mapsto \mathsf{true}}\ \mathit{Eq} \quad \frac{\widehat{n_1} \neq \widehat{n_2}}{\mathsf{eq}\ (\widehat{n_1}, \widehat{n_2}) \mapsto \mathsf{false}}\ \mathit{Eq}' \quad \frac{}{\mathsf{fix}\ x{:}A.\, e \mapsto [\mathsf{fix}\ x{:}A.\, e/x]e}\ \mathit{Fix}$$

Figure 1: Call-by-value operational semantics of TML

2 Implementing the call-by-value operational semantics (30 pts)

In this part, you will implement the call-by-value operational semantics with a substitution. We already implemented the call-by-value operational semantics of untyped λ-calculus in a previous assignment. Similarly, in this assignment, you will implement the call-by-value operational semantics of TML with de Bruijn indexes. For your reference, we give the call-by-value operational semantics of TML. See Figure 1.

We do not give the definition of values for the abstract syntax with de Bruijn indexes. Thus you have to find the definition of values v. You also have to find substitution rules for TML. Since our expression uses de Bruijn indexes to represent variables, you do not need to implement α-conversion.

Programming instruction

See `eval.mli` and `eval.ml`. The goal is to implement the function `step1` in the structure `Eval`:

```
val step1 : Tml.exp -> Tml.exp
```

`step1` takes a de Bruijn expression e. If e reduces to another expression e', it returns e'. Otherwise it raises an exception `Stuck` which is defined in the structure `Eval`. In implementing the function `step1`, you will perhaps need to implement some auxiliary functions.

You can test your `step1` function as follows:

- `loop (step1 (wait1 show1));;` : Each time you press the return key, a reduced expression is displayed.
- `loop (step1 show1);;` : Without pressing the return key, the entire reduction sequence is displayed.
- `loop (eval1 show1);;` : Only the final result is displayed.
- Use `loopFile` to read in a file.

Here are examples:

```
# loop (step1 (wait1 show1));;
Tml>:(fn x : (int -> int) => x) (fn y : int => y z);
((lam. 0) (lam. (0 1)))
Press return:
(lam. (0 1))
Press return:

# loop (eval1 show1);;
Tml>:fst (((fn x : int => x) (+ (100,200))), 2);
<300>
```

3 Abstract machine N (50 pts)

The last part of this assignment is to design and implement an abstract machine N. The abstract machine N follows the *call-by-need* strategy which is a variant of the *call-by-name* strategy. In the call-by-name strategy, a function argument is substituted directly into the function body. The argument is evaluated when it is used in the evaluation of the function, otherwise it is never evaluated. If the argument is used several times, it is re-evaluated each time. Thus when a function argument is frequently used, the call-by-name strategy is significantly slower than the call-by-value strategy.

To overcome this weakness, we introduce a new evaluation strategy, call-by-need. In the call-by-need strategy, once a function argument is evaluated, the result of the evaluation is stored in some memory space. When the argument is used again, the stored value is returned without another evaluation. The call-by-need evaluation produces the same result as call-by-name, but if the function argument is used two or more times, call-by-need is always faster than call-by-name because the argument is not re-evaluated in the call-by-need strategy.

We want to design and implement the abstract machine N (call-by-Need) which is similar to the abstract machine E. In order to implement the call-by-need machine, we need a memory structure to store results of the evaluation. Fortunately, in Assignment 2, we already implemented an appropriate data structure for this machine, *heap*. We use the heap structure that stores two different kinds of values, delayed expressions and computed values. A delayed expression is allocated when the machine evaluates an application of the form $(\lambda.e)\ e'$. Since the evaluation of the argument e' has to be delayed until its computed value is actually needed, the argument is allocated in the heap as the delayed expression. The location of the heap cell containing the delayed expression is bound to de Bruijn index 0. When the machine evaluates a de Bruijn index that is bound to the location of a delayed expression, the machine first evaluates it, and then updates the heap cell with the computed value that is the result of evaluating the delayed expression. When the machine evaluates the same de Bruijn index again, the computed value is immediately returned without another evaluation. For example, consider the expression $(\lambda.0\ 0)\ ((\lambda.0)\ (\lambda.0))$. The argument $\lambda.0\ (\lambda.0)$ is allocated as a delayed expression and its location is bound to de Bruijn index 0. When evaluating the first 0 of the application 0 0, we first evaluate the delayed expression $(\lambda.0)\ (\lambda.0)$ whose location is bound to 0. After evaluating it, we update the heap cell with the computed value $\lambda.0$. Then, when evaluating the second 0, we immediately obtain the computed value $\lambda.0$.

We show a partial definition of the abstract machine N below:
Compared with the abstract machine E, stored value sv and heap h are added in the abstract machine N. Moreover the definitions of environment η and state s are slightly changed. A heap is a mapping from *locations* to stored values, where a location l is a value for a reference, or simply another name for a reference. In the definition of environments, $\cdot$ denotes an empty environment, and $n \hookrightarrow l$ means that de Bruijn index n is bound to location l. A state $h,\ \sigma \blacktriangleright e\ @\ \eta$ means that the machine is currently analyzing e under the heap h and environment η. $h,\ \sigma \blacktriangleleft V$ means that the machine is currently returning V to the stack σ under the heap h.

We give a state transition example in Figure 3. In this example, we show how the initial state $\cdot \parallel \square \blacktriangleright (\lambda.0\ 0)\ ((\lambda.0)\ (\lambda.0))\ @\ \eta_0$ makes a sequence of transitions into the final state (where η_0 can be an empty environment). Of course we define our own closed values and frames. Basically our definitions are similar to those of the abstract machine E, except for a new kind of frame $[l]$. The frame $[l]$ is generated

$$
\begin{array}{rrcl}
\text{expression} & e & ::= & n \mid \lambda.e \mid e\ e \mid (e,e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid \\
 & & & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \\
 & & & \mathsf{fix}\ e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
 & & & \widehat{n} \mid \mathsf{plus} \mid \mathsf{minus} \mid \mathsf{eq} \mid \\
\text{closed value} & V & ::= & \langle \textit{to be filled by students} \rangle \\
\text{stored value} & sv & ::= & \mathsf{computed}(V) \mid \mathsf{delayed}(e,\eta) \\
\text{heap} & h & ::= & \cdot \mid h, l \hookrightarrow sv \\
\text{environment} & \eta & ::= & \cdot \mid \eta, n \hookrightarrow l \\
\text{frame} & \phi & ::= & \langle \textit{to be filled by students} \rangle \\
\text{stack} & \sigma & ::= & \square \mid \sigma; \phi \\
\text{state} & s & ::= & h \parallel \sigma \blacktriangleright e \mathbin{@} \eta \mid h \parallel \sigma \blacktriangleleft V
\end{array}
$$

Figure 2: Abstract machine N for TML

when the machine evaluates a de Bruijn index n bound to the location l in the current environment where l points to a delayed expression in the current heap. It means that the machine is waiting for the result of evaluating the delayed expression. When the result is returned, the heap cell at location l is updated with that result.

Before starting to write code, you will have to complete the definition of the abstract machine N and design your own state transition rules.

- Complete the definition of closed values V.
- Complete the definition of frames ϕ.
- Give the state transition rules for the abstract machine N.

For your reference, we give the call-by-name operational semantics using evaluation contexts for TML. See Figure 4. It helps you design your own call-by-need machine.

Now it's time to have fun!

Programming instruction

First see the `heap.ml`.

```
exception InvalidLocation
type loc
type 'a heap
val empty : 'a heap
val allocate : 'a heap -> 'a -> 'a heap * loc
val deref : 'a heap -> loc -> 'a
val update : 'a heap -> loc -> 'a -> 'a heap
```

`loc` is the internal representation of location l in the heap. `empty` returns an empty heap. `allocate` h v stores a given value v in a fresh heap cell and returns the pair (h', l) of the updated heap h' and the location l of this cell. `deref` h l fetches the value v stored in the heap cell at location l. `InvalidLocation` is raised if the l is an invalid location. `update` h l v updates the heap cell at location l with the given value v and returns the updated heap h'. `InvalidLocation` is raised if the l is an invalid `loc`.

Next see the structure `Eval` which has signature `EVAL`.

```
open Tml
exception NotImplemented
exception Stuck

type stoval =
  ...
and stack =
  ...
```

$$
\begin{array}{ll}
 & \cdot \parallel \Box \blacktriangleright (\lambda.0\ 0)\ ((\lambda.0)\ (\lambda.0)) \mathbin{@} \eta_0 \\
\mapsto_{\mathsf{N}} & \cdot \parallel \Box; \Box_{\eta_0}\ ((\lambda.0)\ (\lambda.0)) \blacktriangleright \lambda.0\ 0 \mathbin{@} \eta_0 \\
\mapsto_{\mathsf{N}} & \cdot \parallel \Box; \Box_{\eta_0}\ ((\lambda.0)\ (\lambda.0)) \blacktriangleleft [\eta_0, \lambda.0\ 0] \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0) \parallel \Box \blacktriangleright 0\ 0 \mathbin{@} \eta_0, 0 \hookrightarrow l_0 & (l_0 \text{ allocated}) \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0 \blacktriangleright 0 \mathbin{@} \eta_0, 0 \hookrightarrow l_0 \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0] \blacktriangleright (\lambda.0)\ (\lambda.0) \mathbin{@} \eta_0 \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0]; \Box_{\eta_0}\ \lambda.0 \blacktriangleright \lambda.0 \mathbin{@} \eta_0 \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0]; \Box_{\eta_0}\ \lambda.0 \blacktriangleleft [\eta_0, \lambda.0] \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0), l_1 \hookrightarrow \mathsf{delayed}(\lambda.0, \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0] \blacktriangleright 0 \mathbin{@} \eta_0, 0 \hookrightarrow l_1 & (l_1 \text{ allocated}) \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0), l_1 \hookrightarrow \mathsf{delayed}(\lambda.0, \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0]; [l_1] \blacktriangleright \lambda.0 \mathbin{@} \eta_0 \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0), l_1 \hookrightarrow \mathsf{delayed}(\lambda.0, \eta_0) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0]; [l_1] \blacktriangleleft [\eta_0, \lambda.0] \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{delayed}((\lambda.0)\ (\lambda.0), \eta_0), l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0; [l_0] \blacktriangleleft [\eta_0, \lambda.0] & (l_1 \text{ updated}) \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]) \parallel \Box; \Box_{\eta_0, 0 \hookrightarrow l_0}\ 0 \blacktriangleleft [\eta_0, \lambda.0] & (l_0 \text{ updated}) \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), \\
 & \quad l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), l_2 \hookrightarrow \mathsf{delayed}(0, \eta_0, 0 \hookrightarrow l_0) \parallel \Box \blacktriangleright 0 \mathbin{@} \eta_0, 0 \hookrightarrow l_2 & (l_2 \text{ allocated}) \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), \\
 & \quad l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), l_2 \hookrightarrow \mathsf{delayed}(0, \eta_0, 0 \hookrightarrow l_0) \parallel \Box; [l_2] \blacktriangleright 0 \mathbin{@} \eta_0, 0 \hookrightarrow l_0 \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), \\
 & \quad l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), l_2 \hookrightarrow \mathsf{delayed}(0, \eta_0, 0 \hookrightarrow l_0) \parallel \Box; [l_2] \blacktriangleleft [\eta_0, \lambda.0] \\
\mapsto_{\mathsf{N}} & \cdot, l_0 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), \\
 & \quad l_1 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]), l_2 \hookrightarrow \mathsf{computed}([\eta_0, \lambda.0]) \parallel \Box \blacktriangleleft [\eta_0, \lambda.0] & (l_2 \text{ updated})
\end{array}
$$

Figure 3: An example of state transitions

```
and state =
  ...
and env = NOT_IMPLEMENT_ENV

and value = NOT_IMPLEMENT_VALUE

and frame = NOT_IMPLEMENT_FRAME

let emptyEnv = NOT_IMPLEMENT_ENV

let value2exp _ = raise NotImplemented

let texp2exp _ = ....

let step1 _ = .....

let step2 _ = raise Stuck

let exp2string exp =
....

let state2string st =
...
...
```

The datatypes `env`, `value`, and `frame` correspond to the syntactic categories environment, closed value, and frame, respectively. The datatypes `stoval`, `stack` and `state` correspond to the syntactic categories stored value, stack and state, respectively, and are already defined in `eval.ml`.

- Define the datatypes `env`, `value`, and `frame`.

$$
\begin{array}{rrcl}
\text{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid \\
& & & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\, e \mid \mathsf{inr}\ x.\, e \mid \\
& & & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
& & & \widehat{n} \mid \mathsf{plus} \mid \mathsf{minus} \mid \mathsf{eq} \mid \mathsf{fix}\ x.\, e \mid \\
\text{value} & v & ::= & \lambda x.\, e \mid (e, e) \mid () \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \\
& & & \mathsf{true} \mid \mathsf{false} \mid \widehat{n} \mid \mathsf{plus} \mid \mathsf{minus} \mid \mathsf{eq} \mid \\
\text{evaluation context} & \kappa & ::= & \Box \mid \kappa\ e \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \mid \\
& & & \mathsf{case}\ \kappa\ \mathsf{of}\ \mathsf{inl}\ x.\, e \mid \mathsf{inr}\ x.\, e \mid \mathsf{if}\ \kappa\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
& & & \mathsf{plus}\ \kappa \mid \mathsf{plus}\ (\kappa, e) \mid \mathsf{plus}\ (v, \kappa) \mid \\
& & & \mathsf{minus}\ \kappa \mid \mathsf{minus}\ (\kappa, e) \mid \mathsf{minus}\ (v, \kappa) \mid \\
& & & \mathsf{eq}\ \kappa \mid \mathsf{eq}\ (\kappa, e) \mid \mathsf{eq}\ (v, \kappa) \mid
\end{array}
$$

$$
\begin{array}{rcll}
(\lambda x.\, e)\ e' & \mapsto_\beta & [e'/x]e & \\
\mathsf{fst}\ (e_1, e_2) & \mapsto_\beta & e_1 & \\
\mathsf{snd}\ (e_1, e_2) & \mapsto_\beta & e_2 & \\
\mathsf{case}\ \mathsf{inl}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [e/x_1]e_1 & \\
\mathsf{case}\ \mathsf{inr}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 & \mapsto_\beta & [e/x_2]e_2 & \\
(\mathsf{fix}\ x.\, e) & \mapsto_\beta & [\mathsf{fix}\ x.\, e/x]e & \\
\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_1 & \\
\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \mapsto_\beta & e_2 & \\
\mathsf{plus}\ (\widehat{n_1}, \widehat{n_2}) & \mapsto_\beta & \widehat{n} & \textit{where } n = n_1 + n_2 \\
\mathsf{minus}\ (\widehat{n_1}, \widehat{n_2}) & \mapsto_\beta & \widehat{n} & \textit{where } n = n_1 - n_2 \\
\mathsf{eq}\ (\widehat{n_1}, \widehat{n_2}) & \mapsto_\beta & \mathsf{true} & \textit{if } n_1 = n_2 \\
\mathsf{eq}\ (\widehat{n_1}, \widehat{n_2}) & \mapsto_\beta & \mathsf{false} & \textit{if } n_1 \neq n_2
\end{array}
$$

$$
\frac{e \mapsto_\beta e'}{\kappa[\![e]\!] \mapsto \kappa[\![e']\!]}\ \mathit{Red}_\beta
$$

Figure 4: Call-by-name operational semantics using evaluation contexts

- Define the value `emptyEnv` that means $\cdot$ in the syntactic category environment.
- Complete the function `value2exp : value -> Tml.exp`. Given a value V_{prim} as defined below, `value2exp` returns a corresponding expression of type `Tml.exp`. For all other kinds of values, `value2exp` raises an exception `NotConvertible`. **It is absolutely important to give a correct implementation of this function, since our test module assumes the correctness of your implementation of `value2exp`.** In other words, if you give a wrong implementation of `value2exp`, you will receive no credit for the entire third part because the test program will judge that your `step2` function (see below) is faulty!

$$
\text{primitive value} \qquad V_{\mathsf{prim}} \quad ::= \quad () \mid \mathsf{true} \mid \mathsf{false} \mid \widehat{n}
$$

- Complete the function `state2string`. There is no strict specification for this function, but you might find this function useful for debugging your code. Feel free to choose whatever specification you like, as we will not test this function. You may choose to return just an empty string if you want!

The final goal of this part is to implement the function `step`:

- Implement the function `step2 : state -> state` which takes a state of the abstract machine N and returns the next state. It raises an exception `Stuck` if no progress can be made.

We will test your implementation of `step2` by examining the result V in the final state using *your* implementation of `Eval.value2exp`.

You can test your `step2` function as follows:

- `loop (step2 (wait2 show2));;` : Each time you press the return key, the next state is displayed.

- `loop (step2 show2);;` : The entire translation sequence is displayed.
- `loop (eval2 show2);;` : Only the final state is displayed.
- Use `loopFile` to read in a file.

Submission instruction

1. Make sure that you can compile `eval.ml` by running `make`.
2. When you have your file `eval.ml` ready for submission, copy it to your hand-in directory on `programming2.postech.ac.kr`. For example, if your Hemos ID is `foo`, copy it to:

   `/home/class/cs321/handin/foo/`