Exercise 3, CS 555

By: Sauce Code Team (Dandan Mo, Qi Lu, Yiming Yang)

Due: April 16th, 2012

1 De Bruijn Notation

1.1 nameless representation

Formally convert the ordinary lambda terms to nameless terms. Using nature numbers to replace the variable names.

Here, we use the list of variables to keep track of the abstract depth of each one, which is just the de Bruijn indices for them. Besides, in order to go around the presumable error of generating hybrid term (S.Term with some DB.Term inside), we first convert the de Bruijn indices into **String** as the new names of the variables, and when we finish the variable name substitution, we call *finalProcess* function to convert these names back into **Int** and remove the variable names in their definitions.

1.2 Haskell Implementation

```haskell
module DeBruijn where

import qualified AbstractSyntax as S
import Data.List
import Data.Char

type Type = S.Type
data Term = Var Int
          | IntConst Integer
          | Tru
          | Fls
          | Abs Type Term
          | App Term Term
          | If Term Term Term
          | Fix Term
          | Let Term Term
          | Bop BOP Term Term
          | Bpr BPR Term Term

instance Show Term where
        show (Var i) = "(Index " ++ show i ++ ")"
        show (IntConst i) = show i
        show Tru = "true"
        show Fls = "false"
        show (Abs τ t) = "abs(" ++ ":" ++ show τ ++ "." ++ show t ++ ")"
        show (App t1 t2) = "app(" ++ show t1 ++ "," ++ show t2 ++ ")"
        show (If t1 t2 t3) = "if " ++ show t1 ++ " then " ++ show t2 ++ " else " ++ show t3 ++ " fi"
        show (Fix t) = "fix " ++ show t
        show (Let t1 t2) = "let " ++ show t1 ++ " " ++ show t2
        show (Bop op t1 t2) = show op ++ "(" ++ show t1 ++ ", " ++ show t2 ++ ")"
        show (Bpr p t1 t2) = show p ++ "(" ++ show t1 ++ "," ++ show t2 ++ ")"

    -- Binary Operator
data BOP = Add | Sub | Mul | Div | Nand
```

```haskell
instance Show BOP where
          show Add = "+"
          show Sub = "-"
          show Mul = "*"
          show Div = "/"
          show Nand = "^"

    -- Binary Predicate
data BPR = Eq | Lt
instance Show BPR where
  show Eq = "="
  show Lt = "<"

newtype Environment = Env [S.Var]
                      deriving Show

lookupEnv :: Environment → S.Var → Int
lookupEnv (e@(Env [ ])) x = error ("variable " ++ x ++ " not bound in environment " ++ show e)
lookupEnv (Env es) x =
  case elemIndices x es of
    [ ] → error "This term has free variables!"
    _ → head $ elemIndices x es

finalProcess :: S.Term → Term
finalProcess (S.Var x) = Var (read x :: Int)
finalProcess (S.Abs x τ t1) = Abs τ (finalProcess t1)
finalProcess (S.App t1 t2) = App (finalProcess t1) (finalProcess t2)
finalProcess S.Tru = Tru
finalProcess S.Fls = Fls
finalProcess (S.If t1 t2 t3) = If (finalProcess t1) (finalProcess t2) (finalProcess t3)
finalProcess (S.IntConst n) = IntConst n
finalProcess (S.IntAdd t1 t2) = Bop Add (finalProcess t1) (finalProcess t2)
finalProcess (S.IntSub t1 t2) = Bop Sub (finalProcess t1) (finalProcess t2)
finalProcess (S.IntMul t1 t2) = Bop Mul (finalProcess t1) (finalProcess t2)
finalProcess (S.IntDiv t1 t2) = Bop Div (finalProcess t1) (finalProcess t2)
finalProcess (S.IntNand t1 t2) = Bop Nand (finalProcess t1) (finalProcess t2)
finalProcess (S.IntEq t1 t2) = Bpr Eq (finalProcess t1) (finalProcess t2)
finalProcess (S.IntLt t1 t2) = Bpr Lt (finalProcess t1) (finalProcess t2)
finalProcess (S.Fix t) = Fix (finalProcess t)
finalProcess (S.Let x t1 t2) = Let (finalProcess t1) (finalProcess t2)


toDeBruijn :: S.Term → Term
toDeBruijn t =
  finalProcess $ f (t, Env [ ])
  where
    f :: (S.Term, Environment) → S.Term
    f (S.Abs x τ t1, Env es) = S.Abs x τ (f (t1, Env (x : es)))
    f (S.App t1 t2, e) = S.App (f (t1, e)) (f (t2, e))
    f (S.Var x, e) = S.Var (show (lookupEnv e x))
    f (S.Let x t1 t2, Env es) = S.Let x (f (t1, Env es)) (f (t2, Env (x : es)))
```

$f\ (S.If\ t_1\ t_2\ t_3, e) = S.If\ (f\ (t_1, e))\ (f\ (t_2, e))\ (f\ (t_3, e))$
$f\ (S.IntAdd\ t_1\ t_2, e) = S.IntAdd\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntSub\ t_1\ t_2, e) = S.IntSub\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntMul\ t_1\ t_2, e) = S.IntMul\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntDiv\ t_1\ t_2, e) = S.IntDiv\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntNand\ t_1\ t_2, e) = S.IntNand\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntEq\ t_1\ t_2, e) = S.IntEq\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.IntLt\ t_1\ t_2, e) = S.IntLt\ (f\ (t_1, e))\ (f\ (t_2, e))$
$f\ (S.Fix\ t, e) = S.Fix\ (f\ (t, e))$
$f\ (t, e) = t$

2 Natural Semantics with Nameless Terms

We modify the Natural Semantics evaluation rules by adding environment and closures. The rules we modified are as the followings:

$$e \vdash Tru \Rightarrow True$$

$$e \vdash Fls \Rightarrow False$$

$$e \vdash \text{IntConst } n \Rightarrow n$$

$$\frac{e \vdash t_1 \Rightarrow True, e \vdash t_2 \Rightarrow \alpha}{e \vdash \text{If } t_1\ t_2\ t_3 \Rightarrow \alpha}$$

$$\frac{e \vdash t_1 \Rightarrow False, e \vdash t_3 \Rightarrow \beta}{e \vdash \text{If } t_1\ t_2\ t_3 \Rightarrow \beta}$$

$$e \vdash \text{Var } i \Rightarrow e[i]$$

$$\frac{e \vdash t_1 \Rightarrow \text{Clo } \lambda.t'\ e', e \vdash t_2 \Rightarrow v, v : e' \vdash t' \Rightarrow v'}{e \vdash t_1\ t_2 \Rightarrow v'}$$

$$\frac{e \vdash t_1 \Rightarrow \text{Clo Fix } t'\ e', e' \vdash \text{Fix } t' \Rightarrow \text{Clo } \lambda.tt\ ee, e \vdash t_2 \Rightarrow v, v : ee \vdash tt \Rightarrow v'}{e \vdash t_1\ t_2 \Rightarrow v'}$$

$$e \vdash \lambda.t_1 \Rightarrow \text{Clo } \lambda.t_1 \text{ e}$$

$$\frac{e \vdash t_1 \Rightarrow v_1, e \vdash t_2 \Rightarrow v_2, \overline{op}(v_1, v_2) = v}{e \vdash op(t_1, t_2) \Rightarrow v}$$

where op are binary arithmetical operations: Add, Sub,Mul,Div,Nand, $\overline{op}$ indicates the real arithmetical functions.

$$\frac{e \vdash t_1 \Rightarrow v_1, e \vdash t_2 \Rightarrow v_2, \overline{rp}(v_1, v_2) = v}{e \vdash rp(t_1, t_2) \Rightarrow v}$$

where rp are binary relational operations: Eq, Lt, $\overline{rp}$ indicates the real relational functions.

$$\frac{e \vdash t_1 \Rightarrow \alpha, \alpha : e \vdash t_2 \Rightarrow \beta}{e \vdash \text{Let } t_1\ t_2 \Rightarrow \beta}$$

$$\frac{e \vdash t_1 \Rightarrow \text{Clo } \lambda.t'\ e', (\text{Clo Fix } \lambda.t'\ \text{e}') : e' \vdash t' \Rightarrow v}{e \vdash \text{Fix } t_1 \Rightarrow v}$$

The followings are the implemented codes:

```
module NSWCAD where
import Data.Maybe
import qualified DeBruijn as S
import qualified IntegerArithmetic as I
import Debug.Trace

data Value = BoolVal Bool | IntVal Integer | Clo S.Term Env
deriving Show

type Env = [Value]

evalInEnv :: Env -> S.Term -> Maybe Value
evalInEnv e t = case t of
          -- true,false
            S.Tru -> Just (BoolVal True)
            S.Fls -> Just (BoolVal False)
          -- integer
            S.IntConst n -> Just (IntVal n)
          -- if
            S.If t1 t2 t3 -> case evalInEnv e t1 of
                                 Just (BoolVal True) -> case evalInEnv e t2 of
                                                          Just a -> Just a
                                                          _ -> error "if-t2"
                                 Just (BoolVal False) -> case evalInEnv e t3 of
                                                          Just b -> Just b
                                                          _ -> error "if-t3"
                                 _ -> error "if-t1"
          -- var
            S.Var i -> Just (e !! i)
          -- app
            S.App t1 t2 -> case evalInEnv e t1 of
                  Just (Clo (S.Abs τ t') e') -> case evalInEnv e t2 of
                                                 Just v' -> case evalInEnv ([v'] ++ e') t' of
                                                               Just vv -> Just vv
                                                               _ -> error "app-replacement"
                                                 _ -> error "app-t2 is not a value"
                  Just (Clo (S.Fix t') e') -> case evalInEnv e' (S.Fix t') of
                                                 Just (Clo (S.Abs tau' tt) ee) -> case evalInEnv e t2 of
```

```
                                                                  Just v' -> case evalInEnv ([v'] ++ ee) tt of
                                                                                 Just vv -> Just vv
                                                                                 _ -> Nothing
                                                                  _ -> Nothing
                                               _ -> Nothing
                    _ -> error "app-t1 is not an abstraction"
    -- abs
      S.Abs τ t1 -> Just (Clo (S.Abs τ t1) e)
    -- add, sub,mul,div,nand
      S.Bop op t1 t2 -> case evalInEnv e t1 of
                          Just (IntVal v1) -> case evalInEnv e t2 of
                                                Just (IntVal v2) -> case op of
                                                                      S.Add -> Just (IntVal (I.intAdd v1 v2))
                                                                      S.Sub -> Just (IntVal (I.intSub v1 v2))
                                                                      S.Mul -> Just (IntVal (I.intMul v1 v2))
                                                                      S.Div -> Just (IntVal (I.intDiv v1 v2))
                                                                      S.Nand -> Just (IntVal (I.intNand v1 v2))

                                                _ -> error "BOP t2 is not a value"
                          _ -> error "BOP t1 is not a value"
    -- eq,lt
      S.Bpr pr t1 t2 -> case evalInEnv e t1 of
                          Just (IntVal v1) -> case evalInEnv e t2 of
                                                Just (IntVal v2) -> case pr of
                                                                      S.Eq -> case I.intEq v1 v2 of
                                                                                True -> Just (BoolVal True)
                                                                                False -> Just (BoolVal False)

                                                                      S.Lt -> case I.intLt v1 v2 of
                                                                                True -> Just (BoolVal True)
                                                                                False -> Just (BoolVal False)

                                                _ -> error "BRP t2 is not a value"
                          _ -> error "BRP t1 is not a value"
    -- let
      S.Let t1 t2 -> case evalInEnv e t1 of
                          Just a -> case evalInEnv ([a] ++ e) t2 of
                                      Just b -> Just b
                                      _ -> error "let-t2 is not a value"
                          _ -> error "let t1 is not a value"
    -- fix
      S.Fix t1 -> case evalInEnv e t1 of
                    Just (Clo (S.Abs τ t') e') -> case evalInEnv ([Clo (S.Fix (S.Abs τ t')) e'] ++ e') t' of
                                                    Just b -> Just b
                                                    _ -> error "fix-point error"
                    _ -> error "fix-t1 is not an abstraction"


eval :: S.Term -> Value
eval t = fromJust (evalInEnv [ ] t)
```

3 C-E-S Compiler and Virtual Machine

3.1 Formal Rules

Foramally statting the compilation rules of C-E-S compiler and virtual machine:

$$C[\text{IntConst n}] = \text{CONST } n$$

$$C[\text{Var i}] = \text{ACCESS } i$$

$$C[\text{Abs t}] = \text{CLOSE } C[t];\ \text{RETURN}$$

$$C[\text{App t1 t2}] = C[t1]; C[t2];\ \text{APPLY}$$

$$C[\text{If t0 t1 t2}] = C[t0];\ \text{IF}; [CloseC[t1]]; [CloseC[t2]]$$

$$C[\text{True}] = \ \text{True}$$

$$C[\text{False}] = \ \text{False}$$

$$C[\text{Fix t}] = C[t];\ \text{Fix}$$

$$C[\text{Let t1 t2}] = C[t1];\ \text{Let };C[t2];\ \text{EndLet}$$

$$C[\text{Bop t1 t2}] = C[t1]; C[t2];\ \text{Bop} \quad \text{-Binary operations: } +, -, *, /$$

$$C[\text{Bpr t1 t2}] = C[t1]; C[t2];\ \text{Bpr} \quad \text{-Binary predicates: } <; ==$$

Foramally statting the transitions of C-E-S compiler and virtual machine:

(Access i : c, e, s) $\rightarrow$ (c, e, (e !! i) : s)

(If:(Close t1):(Close t2):c, e, (Value True):s) $\rightarrow$ (t1++c, e, s)

(If:(Close t1):(Close t2):c, e, (Value False):s) $\rightarrow$ (t2++c, e, s)

(Close code':code, env, s) $\rightarrow$ (code, env, Env [Clo code' env]:s)

(Apply:code, env, (Value v):(Env [Clo code' env']):s) $\rightarrow$ (code', v:env', (Code code):(Env env):s)

(Apply:code, env, (Value v):(Value (Clo code' env')):s) $\rightarrow$ $C[$(code', v:env', (Code code):(Env env):s)

(Apply:code, env, (Value v):(Value (CloFix [Close [Close code', Return], Fix] env')):s)$]$
$\rightarrow$ (code', v:(CloFix [Close [Close code', Return], Fix] env'):env', (Code code):(Env env):s)

$(Return : c, e, s' : (Codec') : (Enve') : s)$ $\rightarrow$ (c', e', s':s)

(Int n:c, e, s) $\rightarrow$ (c, e, (Value n):s)

(Bool b:c, e, s) $\rightarrow$ (c, e, (Value b):s)

(bop:c, e, (Value v2):(Value v1):s) $\rightarrow$ (c, e, (Value (bop v1 v2)):s)

(bpr:c, e, (Value v2):(Value v1):s) $\rightarrow$ (c, e, (Value (bpr v1 v2)):s)

(Let:code, env, (Value v):s) $\rightarrow$ (code, v:env, s)

(EndLet:code, v:env, s) → (code, env, s)

(Fix:code, env, (Env [Clo code' env']):s) → (code', (CloFix [Close code', Fix] []):env, (Code code):(Env env):s)

3.1.1 Haskell Implementation

```
module CESMachine where
import Debug.Trace
import qualified IntegerArithmetic as I
import qualified DeBruijn as DB

data Inst = Int Integer
          | Bool Bool
          | Bop BOP
          | Bpr BPR
          | Access Int
          | Close Code
          | Let
          | EndLet
          | Apply
          | Return
          | If
          | Fix
          deriving Show

data BOP = Add | Sub | Mul | Div | Nand
instance Show BOP where
            show Add = "+"
            show Sub = "-"
            show Mul = "*"
            show Div = "/"
            show Nand = "^"

data BPR = Eq | Lt
instance Show BPR where
  show Eq = "="
  show Lt = "<"

type Code = [Inst]
data Value = BoolVal Bool | IntVal Integer | Clo Code Env | CloFix Code Env
          deriving Show
type Env = [Value]
data Slot = Value Value | Code Code | Env Env
          deriving Show
type Stack = [Slot]
type State = (Code, Env, Stack)
```

```haskell
compile :: DB.Term -> Code
compile t = case t of
    DB.Var n -> [Access n]
    DB.IntConst n -> [Int n]
    DB.Abs tp t0 -> case compile t0 of t1 -> [Close (t1 ++ [Return])]
    DB.App t1 t2 -> case compile t1 of
                      t1' -> case compile t2 of
                        t2' -> t1' ++ t2' ++ [Apply]
    DB.If t0 t1 t2 -> compile t0 ++ [If] ++ [Close (compile t1)] ++ [Close (compile t2)]
    DB.Tru -> [Bool True]
    DB.Fls -> [Bool False]
    DB.Fix t0 -> case compile t0 of t0' -> t0' ++ [Fix]
    DB.Let t1 t2 -> case compile t1 of
                      t1' -> case compile t2 of
                        t2' -> t1' ++ [Let] ++ t2' ++ [EndLet]

    DB.Bop bop t1 t2 -> case compile t1 of
                          t1' -> case compile t2 of
                            t2' -> case bop of
                              DB.Add -> t1' ++ t2' ++ [Bop Add]
                              DB.Sub -> t1' ++ t2' ++ [Bop Sub]
                              DB.Mul -> t1' ++ t2' ++ [Bop Mul]
                              DB.Div -> t1' ++ t2' ++ [Bop Div]
                              DB.Nand -> t1' ++ t2' ++ [Bop Nand]

    DB.Bpr bpr t1 t2 -> case compile t1 of
                          t1' -> case compile t2 of
                            t2' -> case bpr of
                              DB.Eq -> t1' ++ t2' ++ [Bpr Eq]
                              DB.Lt -> t1' ++ t2' ++ [Bpr Lt]
step :: State -> Maybe State
step state = case state of
     (Access i : c, e, s) -> Just (c, e, Value (e !! i) : s)
     (If : (Close t) : (Close f) : c, e, (Value (BoolVal v0)) : s) -> case v0 of
                                                    True -> Just (t ++ c, e, s)
                                                    False -> Just (f ++ c, e, s)
     (Close code' : code, env, s) -> Just (code, env, (Value (Clo code' env)) : s)
     (Apply : code, env, (Value v) : (Value (Clo code' env')) : s) ->
         Just (code', v : env', (Code code) : (Env env) : s)
     (Apply : code, env, (Value v) : (Value (CloFix [Close [Close code', Return], Fix] env')) : s) ->
         Just (code', v : (CloFix [Close [Close code', Return], Fix] env') : env', (Code code) : (Env env) : s)
     (Return : c, e, s' : (Code c') : (Env e') : s) -> Just (c', e', s' : s)
     (Int n : c, e, s) -> Just (c, e, (Value (IntVal n)) : s)
     (Bool b : c, e, s) -> Just (c, e, (Value (BoolVal b)) : s)
     ((Bop bop) : c, e, (Value (IntVal v2)) : (Value (IntVal v1)) : s) -> case bop of
                              Add -> Just (c, e, (Value (IntVal (I.intAdd v1 v2))) : s)
                              Sub -> Just (c, e, (Value (IntVal (I.intSub v1 v2))) : s)
                              Mul -> Just (c, e, (Value (IntVal (I.intMul v1 v2))) : s)
                              Div -> Just (c, e, (Value (IntVal (I.intDiv v1 v2))) : s)
                              Nand -> Just (c, e, (Value (IntVal (I.intNand v1 v2))) : s)
     ((Bpr bpr) : c, e, (Value (IntVal v2)) : (Value (IntVal v1)) : s) -> case bpr of
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad Eq \rightarrow Just\ (c, e, (Value\ (BoolVal\ (I.intEq\ v1\ v2))) : s) \\
\qquad\qquad\qquad\qquad Lt \rightarrow Just\ (c, e, (Value\ (BoolVal\ (I.intLt\ v1\ v2))) : s) \\
\qquad (Let : code, env, (Value\ v) : s) \rightarrow Just\ (code, v : env, s) \\
\qquad (EndLet : code, v : env, s) \rightarrow Just\ (code, env, s) \\
\qquad (Fix : code, env, (Value\ (Clo\ code'\ env')) : s) \rightarrow \\
\qquad\quad Just\ (code', (CloFix\ [Close\ code', Fix]\ [\,]) : env, (Code\ code) : (Env\ env) : s) \\
\qquad \_ \rightarrow Nothing \\
loop :: State \rightarrow State \\
loop\ state = \\
\qquad \mathbf{case}\ step\ state\ \mathbf{of} \\
\qquad\qquad Just\ state' \rightarrow loop\ state' \\
\qquad\qquad Nothing \rightarrow state \\
\\
eval :: DB.Term \rightarrow Value \\
eval\ t = \mathbf{case}\ loop\ (compile\ t, [\,], [\,])\ \mathbf{of} \\
\qquad\qquad (\_, \_, Value\ v : \_) \rightarrow v \\
\qquad\qquad \_ \rightarrow error\ \texttt{"not a value"}
\end{array}
$$

4 CPS

Implement CPS for the core lambda-language consisting of variables, abstractions, applications, primitive constants, primitive operations (+, -, etc.), if, let, and fix.

4.1 CPS Transformation Rules

The CPS transformation scheme to be used is the one shown in class (the original Fischer-Plotkin CPS transformation). Here it is:

$$
\begin{aligned}
\text{variables:} \quad & [\![x]\!] = \lambda\kappa.\ \kappa\ x \\
\text{abstractions:} \quad & [\![\lambda x.t_1]\!] = \lambda\kappa.\ \kappa\ (\lambda x.[\![t_1]\!]) \\
\text{applications:} \quad & [\![t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!](\lambda v_1.\ [\![t_2]\!](\lambda v_2.\ v_1\ v_2\ \kappa)) \\
\text{constants:} \quad & [\![c]\!] = \lambda\kappa.\ \kappa\ c
\end{aligned}
$$

The constants include boolean and integer constants. Besides, we also implemented the following rules:

$$
\begin{aligned}
\text{conditional terms:} \quad & [\![\ \text{If}\ t_1\ t_2\ t_3]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v.\ \text{If}\ v\ ([\![t_2]\!]\ \kappa)\ ([\![t_3]\!]\ \kappa)) \\
\text{let bindings [1]:} \quad & [\![\ \text{Let}\ x\ t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v.\ \text{Let}\ x\ v\ ([\![t_2]\!]\ \kappa)) \\
\text{binary operators:} \quad & [\![bop\ t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v_1.\ [\![t_2]\!]\ (\lambda v_2.\ (\kappa\ (bop\ v_1\ v_2)))) \\
\text{fix terms:} \quad & [\![\ \text{Fix}\ t_1\ ]\!] = \begin{cases} \lambda\kappa.\ \kappa\ (\text{Fix}\ \lambda f.\ \lambda x.\ [\![t_{11}]\!]) & \text{if } t_1 = \lambda f.\ \lambda x.\ t_{11}, \\ \lambda\kappa.\ [\![t_1]\!]\ (\lambda v_1.\ (\text{Fix}\ v_1)\ \kappa) & \text{otherwise.} \end{cases}
\end{aligned}
$$

4.2 Type-Preserved CPS Transformation

In addition, we also implement the type-preserved CPS transformation using a function to transform the types into CPS form, i.e. *toCPSType* function in CPS module. Formally [2], given a primitive type σ (TypeInt, TypeBool), its CPS form type σ' is:

$$\sigma' = \sigma$$

And for function type $\alpha \to \beta$, the associated CPS form $(\alpha \to \beta)'$ is:

$$(\alpha \to \beta)' = \alpha' \to (\beta' \to o) \to o$$

where o is a pseudo continuation return "type". Since continuations actually don't return values, we can set this o to any type. In our implementation, we just add one parameter named *answerType* for all CPS transformation functions, leaving it as a user option. But we also add "**TypeNull**" in **Type** data structure of *AbstractSyntax* module for debugging. Besides, for the generated continuation types, for example, given the transformation:

$$[\![t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!](\lambda v_1.\ [\![t_2]\!](\lambda v_2.\ v_1\ v_2\ \kappa))$$

if t_1 is of type $\alpha \to \beta$, then v_1 must be of type $(\alpha \to \beta)' = \alpha' \to (\beta' \to o) \to o$, v_2 of type α', and κ of type $\beta' \to o$. And it is similar for all the other transformation rules.

4.3 Get Type With Context

At first, we use **typeCheck** function in *Typing* module to calculate the type for each subterm. However, for the case of variables and the subterms within abstractions and fix terms, since the type information of some variables are out of the scope, this method will fail.

So instead, we use **typing** function in *Typing* module, which takes one argument of the current typing context Γ explicitly. And in *CPS* module, we also write a function **toCPSWithContext** to take this Γ as an explicit argument. Then, we can just make the function **toCPS** to call it, starting with an empty typing context.

With this implementation, we can know that the only cases of adding typing pairs into Γ are the **Let** bindings and the abstractions. For all the other cases, we can recursively call the **toCPSWithContext** function for the subterms with the current Γ.

The implementation of CPS transformation is as follows:

```
module CPS where

import Data.Maybe

import qualified AbstractSyntax as S
import qualified Typing as T

toCPSType :: S.Type → S.Type → S.Type
toCPSType S.TypeBool _ = S.TypeBool
toCPSType S.TypeInt _ = S.TypeInt
toCPSType (S.TypeArrow τ1 τ2) answerType =
  S.TypeArrow (toCPSType τ1 answerType) (S.TypeArrow (S.TypeArrow (toCPSType τ2 answerType) answerType)
```

```
                  answerType)

toCPSWithContext :: S.Type → S.Term → T.Context → S.Term
toCPSWithContext answerType t Γ =
  case t of
    S.IntConst n → S.Abs "κ" (S.TypeArrow S.TypeInt answerType) (S.App (S.Var "κ") t)
    S.Tru → S.Abs "κ" (S.TypeArrow S.TypeBool answerType) (S.App (S.Var "κ") t)
    S.Fls → S.Abs "κ" (S.TypeArrow S.TypeBool answerType) (S.App (S.Var "κ") t)
    S.Var x → S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                        (S.App (S.Var "κ") t)
    S.App t1 t2 → S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                            (S.App (toCPSWithContext answerType t1 Γ)
                                   (S.Abs "v1" (toCPSType (fromJust $ T.typing Γ t1) answerType)
                                           (S.App (toCPSWithContext answerType t2 Γ)
                                                  (S.Abs "v2" (toCPSType (fromJust $ T.typing Γ t2)
                                                                          answerType)
                                                          (S.App (S.App (S.Var "v1") (S.Var "v2"))
                                                                 (S.Var "κ"))))))
    S.Abs x τ1 t1 → S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                              (S.App (S.Var "κ")
                                     (S.Abs x (toCPSType τ1 answerType)
                                            (toCPSWithContext answerType t1 (T.Bind Γ x τ1))))
    S.If t1 t2 t3 → S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                              (S.App (toCPSWithContext answerType t1 Γ)
                                     (S.Abs "v" (toCPSType (fromJust $ T.typing Γ t1) answerType)
                                            (S.If (S.Var "v")
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Var "κ"))
                                              (S.App (toCPSWithContext answerType t3 Γ)
                                                     (S.Var "κ")))))
    S.Let x t1 t2 → S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                              (S.App (toCPSWithContext answerType t1 Γ)
                                     (S.Abs "v" (toCPSType (fromJust $ T.typing Γ t1) answerType)
                                            (S.Let x (S.Var "v")
                                                   (S.App (toCPSWithContext answerType t2
                                                                  (T.Bind Γ x (fromJust $ T.typing
                                                                                  Γ t1)))
                                                          (S.Var "κ")))))
    S.Fix t1 →
      case t1 of
        S.Abs f τ1 (S.Abs x τ2 t11) →
            S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                      (S.App (S.Var "κ")
                             (S.Fix
                               (S.Abs f (toCPSType τ1 answerType)
                                      (S.Abs x (toCPSType τ2 answerType)
                                             (toCPSWithContext answerType t11
                                                  (T.Bind (T.Bind Γ f τ1) x τ2))))))
        _ →
            S.Abs "κ" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                      (S.App (toCPSWithContext answerType t1 Γ)
```

```
                                (S.Abs "v1" (toCPSType (fromJust $ T.typing Γ t1) answerType)
                                            (S.App (S.Fix (S.Var "v1"))
                                                   (S.Var "κ"))))
S.IntAdd t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeInt answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntAdd (S.Var "v1")
                                                                             (S.Var "v2")))))))
S.IntSub t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeInt answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntSub (S.Var "v1")
                                                                             (S.Var "v2")))))))
S.IntMul t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeInt answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntMul (S.Var "v1")
                                                                             (S.Var "v2")))))))
S.IntDiv t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeInt answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntDiv (S.Var "v1")
                                                                             (S.Var "v2")))))))
S.IntNand t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeInt answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntNand (S.Var "v1")
                                                                              (S.Var "v2")))))))
S.IntEq t1 t2 → S.Abs "κ" (S.TypeArrow S.TypeBool answerType)
                                (S.App (toCPSWithContext answerType t1 Γ)
                                       (S.Abs "v1" S.TypeInt
                                              (S.App (toCPSWithContext answerType t2 Γ)
                                                     (S.Abs "v2" S.TypeInt
                                                            (S.App (S.Var "κ")
                                                                   (S.IntEq (S.Var "v1")
                                                                            (S.Var "v2")))))))
```

$$
\begin{array}{l}
\mathit{S.IntLt}\ t_1\ t_2 \to \mathit{S.Abs}\ \text{“}\kappa\text{“}\ (\mathit{S.TypeArrow}\ \mathit{S.TypeBool}\ \mathit{answerType}) \\
\qquad (\mathit{S.App}\ (\mathit{toCPSWithContext}\ \mathit{answerType}\ t_1\ \Gamma) \\
\qquad\quad (\mathit{S.Abs}\ \texttt{"v1"}\ \mathit{S.TypeInt} \\
\qquad\qquad (\mathit{S.App}\ (\mathit{toCPSWithContext}\ \mathit{answerType}\ t_2\ \Gamma) \\
\qquad\qquad\quad (\mathit{S.Abs}\ \texttt{"v2"}\ \mathit{S.TypeInt} \\
\qquad\qquad\qquad (\mathit{S.App}\ (\mathit{S.Var}\ \text{“}\kappa\text{“}) \\
\qquad\qquad\qquad\quad (\mathit{S.IntLt}\ (\mathit{S.Var}\ \texttt{"v1"}) \\
\qquad\qquad\qquad\qquad (\mathit{S.Var}\ \texttt{"v2"})))))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{toCPS} :: \mathit{S.Type} \to \mathit{S.Term} \to \mathit{S.Term} \\
\mathit{toCPS}\ \mathit{answerType}\ t = \\
\quad \mathit{toCPSWithContext}\ \mathit{answerType}\ t\ \mathit{T.Empty}
\end{array}
$$

5 C-E-3R Compiler and Virtual Machine

The input of the C-E-3R machine are the Debruijn terms. The machine will compile the Debruijn terms into the instructions, which together with the environment(Env) and three registers(r_1,r_2,r_3) form the initial state of the machine.We design the transition rules for this machine, and apply them to the initial state until we finish process all the instructions. If error occurred, the machine will stop and output the last valid value stored in r_1.

C-E-3R machine is consisted of a compiler and a set of transition rules.The compiler takes the restricted forms of λ terms as its input and transform them into the instructions we design. The restricted forms categorizes the DeBruijn terms into “body” and “atom”, which are also DeBruijn terms but have another name. So we need to define which terms are “body” and which terms are “atom””.

And then we implement the compilers consisted of two kinds of compiling functions, acompile is for compiling “atom” terms and bcompile is for compiling “body” terms.

The “atom” and “body”, compilers, instruction set and transition rules form a complete C-E-3-R machine.

5.1 Definition of “body” and “atom”

1): “body”

$$
\begin{array}{lll}
\text{body} := & t_1\ t_2\ t_3 & \\
& t_1\ t_2 & \\
& n & n \text{ is an integer constant} \\
& i & i \text{ is an index for the variable} \\
& true & \\
& false & \\
& \text{if } t_1\ t_2\ t_3 & \\
& \text{let i t} & \text{i is an index}
\end{array}
$$

Table 1: Instructions

Instruction	Explanation
ACCESS_i j	take the j-th element from environment into r_i
CONST_i n	put integer constant n into r_i
BOOL_i b	put boolean constant b into r_i
CLOSE_i c	create a closure with Code c and the current environment, put it into r_i
$\text{BOP}_i\ v_1\ v_2$	v_1,v_2 are indexes, get their values from the environment, do arithmetic calculation put the result into r_i BOP are ADD, SUB, MUL, DIV, NAND, EQ, LT
IF $v\ c_1\ c_2$	v is an index, it points to the value of the condition in the environment c_1,c_2 are Code, they are branches we go to c_1 branch if the condition is true,otherwise go to c_2 branch
LET $v\ c$	v is an index pointing to substitutive value,we put this value into the environment
ENDLET	remove the head of the environment
$TAILAPPLY1$	extract code in r_1, put r_2's value into environment, execute the code
$TAILAPPLY2$	extract code in r_1, put r_2 and r_3 value into environment, execute the code

2):"atom"

$$
\begin{array}{llr}
\text{atom} := & i & \text{index} \\
& n & \text{integer constant} \\
& true & \\
& false & \\
& \lambda.\lambda.t & \\
& \lambda.t & \\
& bop\ v_1\ v_2 & bop \text{ refers to binary operation} \\
& & (+, -, *, /,' =, <), v_1 \text{ and } v_2 \text{ are indexes} \\
& \text{fix } i & i \text{ is an index} \\
& \text{fix } \lambda.\lambda.\lambda.t &
\end{array}
$$

5.2 Instruction sets

The i appearing in the instructions is $i \in \{1, 2, 3\}$,r_i is register. Please refer to Table 1.

5.3 Compilers

B: compile the body into codes
A: compile the atom into instructions

1): body compiler(Table 2)

2):atom compier (Table 3), A_j's subindex is $j \in \{1, 2, 3\}$
For the fix term, in the compiling process, we just close the instructions code, which is FIX Code.

Table 2: body compiler: B

B(DeBruijn term)	Instructions
B(t_1 t_2 t_3)	A1(t_1); A2(t_2); A3(t_3); TAILAPPLY2;
B(t_1 t_2)	A1(t_1); A2(t_2); TAILAPPLY1;
B(n)	A1(n)
B(i)	A1(i)
B(true)	A1(true)
B(false)	A1(false)
B(if v t_2 t_3)	IF v B(t_2) B(t_3)
B(let i lst)	LET i B(t); ENDLET

Table 3: atom compiler:A

A(DeBruijn term)	Instructions
A_j(i)	$ACCEESS_j$ i
A_j(n)	$CONST_j$ n
A_j(true)	$BOOL_j$ True
A_j(false)	$BOOL_j$ False
A_j($\lambda.\lambda.t$)	$CLOSE_j$ B(t)
A_j($\lambda.t$)	$CLOSE_j$ B(t)
A_j(bop v_1 v_2)	BOP_j v_1 v_2
bop is $+, -, *, /, \prime =, <$	BOP is ADD, SUB, MUL, DIV, NAND, EQ, LT
A_j(fix i)	$CLOSE_j$ (FIX B(i))
A_j(fix $\lambda.\lambda.\lambda.t$)	$CLOSE_j$ (FIX B(t))

5.4 Transition rules

For integer constants,boolean constants,index and "let" term, the transition rules are as described in class;
For binary operation, we look up the values in the environment according to the indexes provided by the instruction, and do the arithmetic calculation;
For "if" terms, we look up the value of the condition part in the environment according to the index provided by the instruction, and choose the correct branch;

Fix term is the most difficult part, in order to achieve the recursive operation, we modify the transition rules for **CLOSE** and **TAILAPPLY1**,**TAILAPPLY2**:

1): if we are closing a FIX instruction with environment e, e.g, $CLOSE_1$(FIX c′), we need to create a closure with c' and a new environment e′, where $e' = (Clo\ [FIX\ c']\ e) : e$, so the closure will be Clo c' ((Clo [FIX c'] e):e). In this way, the machine knows that fix itself is also an argument for fix.

2): if we are applying a FIX closure, e.g Clo [Fix c′] e, to the other argument, we must put itself into its environment after the other argument. For example:
((TAILAPPLY1):c, e, (Clo [Fix c′] e′),v,_) ⇒ (c′,v:(Clo [FIX c′] e′):e′,(UNCARE,UNCARE,UNCARE)).
In this way, fix term itself are added into the environment as an argument.

Table 4: Transition Rules

current state	next state
((CONST$_1$ n):c,e,(_,v_2,v_3))	(c,e,(n,v_2,v_3))
((CONST$_2$ n):c,e,(v_1,_,v_3))	(c,e,(v_1,n,v_3))
((CONST$_3$ n):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,n))
((BOOL$_1$ b):c,e,(_,v_2,v_3))	(c,e,(b,v_2,v_3))
((BOOL$_2$ b):c,e,(v_1,_,v_3))	(c,e,(v_1,b,v_3))
((BOOL$_3$ b):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,b))
((ACCESS$_1$ i):c,e,(_,v_2,v_3))	(c,e,(e[i],v_2,v_3))
((ACCESS$_2$ i):c,e,(v_1,_,v_3))	(c,e,(v_1,e[i],v_3))
((ACCESS$_3$ i):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,e[i]))
((CLOSE$_1$ [FIX c′]):c,e,(_,v_2,v_3))	(c,e,(Clo c′ ((Clo [FIX c′] e):e),v_2,v_3))
((CLOSE$_2$ [FIX c′]):c,e,(v_1,_,v_3))	(c,e,(v_1,Clo c′ ((Clo [FIX c′] e):e),v_3))
((CLOSE$_3$ [FIX c′]):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,Clo c′ ((Clo [FIX c′] e):e)))
((CLOSE$_1$ c′):c,e,(_,v_2,v_3))	(c,e,(Clo c′ e,v_2,v_3))
((CLOSE$_2$ c′):c,e,(v_1,_,v_3))	(c,e,(v_1,Clo c′ e,v_3))
((CLOSE$_3$ c′):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,Clo c′ e))
((BOP$_1$ x_1,x_2):c,e,(_,v_2,v_3))	(c,e,(bop($e[x_1]$,$e[x_2]$),v_2,v_3))
((BOP$_2$ x_1,x_2):c,e,(v_1,_,v_3))	(c,e,(v_1,bop($e[x_1]$,$e[x_2]$),v_3))
((BOP$_3$ x_1,x_2):c,e,(v_1,v_2,_))	(c,e,(v_1,v_2,bop($e[x_1]$,$e[x_2]$)))
((IF v c_1 c_2):c,e,(v_1,v_2,v_3))	(c_1++ c,e,(v_1,v_2,v_3))———$e[v] = true$
((IF v c_1 c_2):c,e,(v_1,v_2,v_3))	(c_2++ c,e,(v_1,v_2,v_3))———$e[v] = false$
((LET v c′):c,e,(v_1,v_2,v_3))	(c′++c,e[v]:e,(v_1,v_2,v_3))
((ENDLET):c,v:e,(v_1,v_2,v_3))	(c,e,(v_1,v_2,v_3))
((TAILAPPLY1:c,e,(Clo [FIX c′] e′,v,_)))	(c′,v:(Clo [FIX c′] e′):e′,(_,_,_))
((TAILAPPLY2):c,e,(Clo [FIX c′] e′,v_1,v_2))	(c′,v_2:v_1:(Clo [FIX c′] e′):e′,(_,_,_))
((TAILAPPLY1:c,e,(Clo c′ e′,v,_)))	(c′,v:e′,(_,_,_))
((TAILAPPLY2):c,e,(Clo c′ e′,v_1,v_2))	(c′,v_2:v_1:e′,(_,_,_))

The FIX CLOSE and FIX APPLY rules guarantee the recursive functions. For details, please refer to Table 4

5.5 Implementation and testing

The codes are as followings. We test factorial functions and all the 6 more complicated testcases given in exercise2, the C-E-3R machine passes all theses test cases.

```
module CE3RMachine where

import qualified AbstractSyntax as S
import qualified DeBruijn as DB
import qualified CPS as CPS
import qualified IntegerArithmetic as I
import Debug.Trace

    -- instructions
data Inst = ACCESS1 Int
```

```
  | ACCESS2 Int
  | ACCESS3 Int
  | CONST1 Integer
  | CONST2 Integer
  | CONST3 Integer
  | BOOL1 Bool
  | BOOL2 Bool
  | BOOL3 Bool
  | CLOSE1 Code
  | CLOSE2 Code
  | CLOSE3 Code
      -- ADD
  | ADD1 Int Int
  | ADD2 Int Int
  | ADD3 Int Int
      -- SUB
  | SUB1 Int Int
  | SUB2 Int Int
  | SUB3 Int Int
      -- MUL
  | MUL1 Int Int
  | MUL2 Int Int
  | MUL3 Int Int
      -- DIV
  | DIV1 Int Int
  | DIV2 Int Int
  | DIV3 Int Int
      -- NAND
  | NAND1 Int Int
  | NAND2 Int Int
  | NAND3 Int Int
      -- EQ
  | EQ1 Int Int
  | EQ2 Int Int
  | EQ3 Int Int
      -- LT
  | LT1 Int Int
  | LT2 Int Int
  | LT3 Int Int
      -- TailApply1, TailApply2
  | TAILAPPLY1
  | TAILAPPLY2
      -- IF
  | IF Int Code Code
      -- LET
  | LET Int Code
  | ENDLET
      -- FIX
  | FIX Code
deriving Show
```

```
-- define the nameless body and atom
type Type = S.Type

-- code,environment,values,regs, state
type Code = [Inst]

type Env = [Value]

data Value = BoolVal Bool
           | IntVal Integer
           | Clo Code Env
           | UNCARE
           deriving Show

type Regs = (Value, Value, Value)

type State = (Code, Env, Regs)


-- compile the nameless body to the machine code

bcompile :: DB.Term → Code
bcompile t =
  case t of
    DB.App (DB.App t1 t2) t3 → [acompile 1 t1] ++ [acompile 2 t2] ++ [acompile 3 t3] ++ [TAILAPPLY2]
    DB.App t1 t2 → [acompile 1 t1] ++ [acompile 2 t2] ++ [TAILAPPLY1]
    DB.Var i → [acompile 1 t]
    DB.IntConst n → [acompile 1 t]
    DB.Tru → [acompile 1 t]
    DB.Fls → [acompile 1 t]
    DB.If (DB.Var v) t2 t3 → [IF v (bcompile t2) (bcompile t3)]
    DB.Let (DB.Var v) t2 → [LET v (bcompile t2)] ++ [ENDLET]
    _ → trace ("bcompile unsupported term:" ++ show t ++ "\n") error "bcompile:Unsupported term"

-- compile the nameless axiom to the machine code instructions
acompile :: Int → DB.Term → Inst
acompile j t =
  case t of
    DB.Var i → case j of
                        1 → ACCESS1 i
                        2 → ACCESS2 i
                        3 → ACCESS3 i
                        _ → error "Var i:Code Generating Error"
    DB.IntConst n → case j of
                        1 → CONST1 n
                        2 → CONST2 n
                        3 → CONST3 n
                        _ → error "IntConst n:Code Generating Error"
    DB.Tru → case j of
                        1 → BOOL1 True
```

```
                        2 → BOOL2 True
                        3 → BOOL3 True
                        _ → error "Tru:Code Generating Error"
DB.Fls → case j of
                        1 → BOOL1 False
                        2 → BOOL2 False
                        3 → BOOL3 False
                        _ → error "Fls:Code Generating Error"
DB.Abs τ1 (DB.Abs τ2 t2) → case j of
                                        1 → CLOSE1 (bcompile t2)
                                        2 → CLOSE2 (bcompile t2)
                                        3 → CLOSE3 (bcompile t2)
                                        _ → error "Abs Abs:Code Generating Error"
DB.Abs τ1 t1 → case j of
                        1 → CLOSE1 (bcompile t1)
                        2 → CLOSE2 (bcompile t1)
                        3 → CLOSE3 (bcompile t1)
                        _ → error "Abs:Code Generating Error"
   -- add
DB.Bop DB.Add (DB.Var i1) (DB.Var i2) → case j of
                        1 → ADD1 i1 i2
                        2 → ADD2 i1 i2
                        3 → ADD3 i1 i2
                        _ → error "add:Code Generating Error"
   -- sub
DB.Bop DB.Sub (DB.Var i1) (DB.Var i2) → case j of
                        1 → SUB1 i1 i2
                        2 → SUB2 i1 i2
                        3 → SUB3 i1 i2
                        _ → error "sub:Code Generating Error"
   -- mul
DB.Bop DB.Mul (DB.Var i1) (DB.Var i2) → case j of
                        1 → MUL1 i1 i2
                        2 → MUL2 i1 i2
                        3 → MUL3 i1 i2
                        _ → error "mul:Code Generating Error"
   -- div
DB.Bop DB.Div (DB.Var i1) (DB.Var i2) → case j of
                        1 → DIV1 i1 i2
                        2 → DIV2 i1 i2
                        3 → DIV3 i1 i2
                        _ → error "div:Code Generating Error"
   -- nand
DB.Bop DB.Nand (DB.Var i1) (DB.Var i2) → case j of
                        1 → NAND1 i1 i2
                        2 → NAND2 i1 i2
                        3 → NAND3 i1 i2
                        _ → error "nand:Code Generating Error"
   -- eq
DB.Bpr DB.Eq (DB.Var i1) (DB.Var i2) → case j of
                        1 → EQ1 i1 i2
```

```
                         2 → EQ2 i1 i2
                         3 → EQ3 i1 i2
                         _ → error "eq:Code Generating Error"
      -- lt
    DB.Bpr DB.Lt (DB.Var i1) (DB.Var i2) → case j of
                         1 → LT1 i1 i2
                         2 → LT2 i1 i2
                         3 → LT3 i1 i2
                         _ → error "lt:Code Generating Error"
      -- fix
    DB.Fix (DB.Var v) → case j of
                         1 → CLOSE1 [FIX (bcompile (DB.Var v))]
                         2 → CLOSE2 [FIX (bcompile (DB.Var v))]
                         3 → CLOSE3 [FIX (bcompile (DB.Var v))]
                         _ → error "fix var:Code Generating Error"
    DB.Fix (DB.Abs τ1 (DB.Abs τ2 (DB.Abs tau3 t3))) → case j of
                       1 → CLOSE1 [FIX (bcompile t3)]
                       2 → CLOSE2 [FIX (bcompile t3)]
                       3 → CLOSE3 [FIX (bcompile t3)]
                       _ → error "fix t:Code Generating Error"
    _ → trace ("unsupported term is " ++ show t) error "acompile:Unsupported term"
```

```
    -- transition rules
ce3rMachineStep :: State → Maybe State
ce3rMachineStep ((CONST1 n) : c, e, (_, v2, v3)) = Just (c, e, (IntVal n, v2, v3))
ce3rMachineStep ((CONST2 n) : c, e, (v1, _, v3)) = Just (c, e, (v1, IntVal n, v3))
ce3rMachineStep ((CONST3 n) : c, e, (v1, v2, _)) = Just (c, e, (v1, v2, IntVal n))
ce3rMachineStep ((BOOL1 b) : c, e, (_, v2, v3)) = Just (c, e, (BoolVal b, v2, v3))
ce3rMachineStep ((BOOL2 b) : c, e, (v1, _, v3)) = Just (c, e, (v1, BoolVal b, v3))
ce3rMachineStep ((BOOL3 b) : c, e, (v1, v2, _)) = Just (c, e, (v1, v2, BoolVal b))
ce3rMachineStep ((ACCESS1 i) : c, e, (_, v2, v3)) = Just (c, e, (e !! i, v2, v3))
ce3rMachineStep ((ACCESS2 i) : c, e, (v1, _, v3)) = Just (c, e, (v1, e !! i, v3))
ce3rMachineStep ((ACCESS3 i) : c, e, (v1, v2, _)) = Just (c, e, (v1, v2, e !! i))
    -- close
ce3rMachineStep ((CLOSE1 [FIX c']) : c, e, (_, v2, v3)) = Just (c, e, (Clo c' ((Clo [FIX c'] e) : e), v2, v3))
ce3rMachineStep ((CLOSE2 [FIX c']) : c, e, (v1, _, v3)) = Just (c, e, (v1, Clo c' ((Clo [FIX c'] e) : e), v3))
ce3rMachineStep ((CLOSE3 [FIX c']) : c, e, (v1, v2, _)) = Just (c, e, (v1, v2, Clo c' ((Clo [FIX c'] e) : e)))
ce3rMachineStep ((CLOSE1 c') : c, e, (_, v2, v3)) = Just (c, e, (Clo c' e, v2, v3))
ce3rMachineStep ((CLOSE2 c') : c, e, (v1, _, v3)) = Just (c, e, (v1, Clo c' e, v3))
ce3rMachineStep ((CLOSE3 c') : c, e, (v1, v2, _)) = Just (c, e, (v1, v2, Clo c' e))

    -- add
ce3rMachineStep ((ADD1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (IntVal (I.intAdd a b), v2, v3))
                 _ → Nothing
    _ → Nothing
```

```
ce3rMachineStep ((ADD2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, IntVal (I.intAdd a b), v3))
                 _ → Nothing
    _ → Nothing
ce3rMachineStep ((ADD3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, v2, IntVal (I.intAdd a b)))
                 _ → Nothing
    _ → Nothing

    -- sub
ce3rMachineStep ((SUB1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (IntVal (I.intSub a b), v2, v3))
                 _ → Nothing
    _ → Nothing
ce3rMachineStep ((SUB2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, IntVal (I.intSub a b), v3))
                 _ → Nothing
    _ → Nothing
ce3rMachineStep ((SUB3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, v2, IntVal (I.intSub a b)))
                 _ → Nothing
    _ → Nothing
    -- mul
ce3rMachineStep ((MUL1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (IntVal (I.intMul a b), v2, v3))
                 _ → Nothing
    _ → Nothing
ce3rMachineStep ((MUL2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, IntVal (I.intMul a b), v3))
                 _ → Nothing
    _ → Nothing
ce3rMachineStep ((MUL3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                 IntVal b → Just (c, e, (v1, v2, IntVal (I.intMul a b)))
                 _ → Nothing
    _ → Nothing
```

```
    -- div
ce3rMachineStep ((DIV1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (IntVal (I.intDiv a b), v2, v3))
                    _ → Nothing
    _ → Nothing
ce3rMachineStep ((DIV2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (v1, IntVal (I.intDiv a b), v3))
                    _ → Nothing
    _ → Nothing
ce3rMachineStep ((DIV3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (v1, v2, IntVal (I.intDiv a b)))
                    _ → Nothing
    _ → Nothing
    -- nand
ce3rMachineStep ((NAND1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (IntVal (I.intNand a b), v2, v3))
                    _ → Nothing
    _ → Nothing
ce3rMachineStep ((NAND2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (v1, IntVal (I.intNand a b), v3))
                    _ → Nothing
    _ → Nothing
ce3rMachineStep ((NAND3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (v1, v2, IntVal (I.intNand a b)))
                    _ → Nothing
    _ → Nothing
    -- eq
ce3rMachineStep ((EQ1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (BoolVal (I.intEq a b), v2, v3))
                    _ → Nothing
    _ → Nothing
ce3rMachineStep ((EQ2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                    IntVal b → Just (c, e, (v1, BoolVal (I.intEq a b), v3))
                    _ → trace ("eq2 i2:" ++ show (e !! i2) ++ "\n") Nothing
    _ → trace ("eq t1:" ++ show (e !! i1) ++ "\n") Nothing
```

```
ce3rMachineStep ((EQ3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                  IntVal b → Just (c, e, (v1, v2, BoolVal (I.intEq a b)))
                  _ → trace ("eq i2:" ++ show (e !! i2) ++ "\n") Nothing
    _ → trace ("eq t1:" ++ show (e !! i1) ++ "\n") Nothing
    -- lt
ce3rMachineStep ((LT1 i1 i2) : c, e, (_, v2, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                  IntVal b → Just (c, e, (BoolVal (I.intLt a b), v2, v3))
                  _ → Nothing
    _ → Nothing
ce3rMachineStep ((LT2 i1 i2) : c, e, (v1, _, v3)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                  IntVal b → Just (c, e, (v1, BoolVal (I.intLt a b), v3))
                  _ → Nothing
    _ → Nothing
ce3rMachineStep ((LT3 i1 i2) : c, e, (v1, v2, _)) =
  case (e !! i1) of
    IntVal a → case (e !! i2) of
                  IntVal b → Just (c, e, (v1, v2, BoolVal (I.intLt a b)))
                  _ → Nothing
    _ → Nothing
    -- if
ce3rMachineStep ((IF v c1 c2) : c, e, (v1, v2, v3)) =
  case (e !! v) of
    BoolVal True → Just (c1 ++ c, e, (v1, v2, v3))
    BoolVal False → Just (c2 ++ c, e, (v1, v2, v3))
    _ → Nothing
    -- let
ce3rMachineStep ((LET v c') : c, e, (v1, v2, v3)) = Just (c' ++ c, (e !! v) : e, (v1, v2, v3))
ce3rMachineStep ((ENDLET) : c, v : e, (v1, v2, v3)) = Just (c, e, (v1, v2, v3))


ce3rMachineStep ((TAILAPPLY1) : c, e, (Clo [FIX c'] e', v, _)) =
  Just (c', v : (Clo [FIX c'] e') : e', (UNCARE, UNCARE, UNCARE))
ce3rMachineStep ((TAILAPPLY2) : c, e, (Clo [FIX c'] e', v1, v2)) =
  Just (c', v2 : v1 : (Clo [FIX c'] e') : e', (UNCARE, UNCARE, UNCARE))


ce3rMachineStep ((TAILAPPLY1) : c, e, (Clo c' e', v, _)) = Just (c', v : e', (UNCARE, UNCARE, UNCARE))
ce3rMachineStep ((TAILAPPLY2) : c, e, (Clo c' e', v1, v2)) = Just (c', v2 : v1 : e', (UNCARE, UNCARE, UNCARE))


ce3rMachineStep t = Nothing


    -- apply transition rules
```

```
loop :: State → State
loop state =
        case ce3rMachineStep state of
                Just state' → loop state'
                Nothing → state

    -- evaluation
eval :: DB.Term → Value
eval t = case loop (bcompile t, [ ], (UNCARE, UNCARE, UNCARE)) of
                (_, _, (v1, UNCARE, UNCARE)) → v1
                s → error "Evaluation Error Occurred!"
```

6 Main

In module *Main*, we set **answerType** to be **TypeNull**, which will be printed out as "**0**". With this pseudo-type, we can see the type of the CPS term more clearly. However, when putting it into practice, we can set it to any concrete type we want instead.

```
module Main where

import qualified System.Environment
import Data.List
import System.IO

import qualified AbstractSyntax as S
import qualified StructuralOperationalSemantics as E
import qualified IntegerArithmetic as I
import qualified Typing as T


import qualified CESMachine as CES
import qualified DeBruijn as DB
import qualified NSWCAD as NDB

import qualified CPS as CPS
import qualified CE3RMachine as CE3R

main :: IO ()
main =
    do
      args ← System.Environment.getArgs
      let [sourceFile] = args
      source ← readFile sourceFile
      let tokens = S.scan source
      let term = S.parse tokens
      let dbterm = DB.toDeBruijn term
      putStrLn ("----Term----")
      putStrLn (show term)

      putStrLn ("----Type----")
```

```
putStrLn (show (T.typeCheck term))

putStrLn ("----Normal Form----")
putStrLn (show (E.eval term))

putStrLn ("----DBTerm----")
putStrLn (show dbterm)
putStrLn ("----Natural Semantics with Clo,Env and DB Term----")
putStrLn (show (NDB.eval dbterm))

putStrLn ("----CES Machine Code----")
putStrLn (show (CES.compile dbterm) ++ "\n")

putStrLn ("----CES Eval----")
putStrLn (show (CES.eval dbterm) ++ "\n")



let answerType = S.TypeNull

let cpsterm = CPS.toCPS answerType term
putStrLn ("----CPS Form----")
putStrLn (show cpsterm)
putStrLn ("----CPS Type----")
putStrLn (show (T.typeCheck cpsterm))

let bodyterm = (S.App cpsterm (S.Abs "x" answerType (S.Var "x")))
putStrLn ("---CPS Normal Form----")
putStrLn (show (E.eval bodyterm))


putStrLn ("----CE3R Machine Code----")
putStrLn (show (CE3R.bcompile (DB.toDeBruijn bodyterm)))
putStrLn ("----CE3R Machine----")
putStrLn (show (CE3R.eval (DB.toDeBruijn bodyterm)))
```

7 Test Cases

7.1 Test 1: iseven 7

```
let
  iseven =
    let
      mod = abs (m:Int. abs(n:Int. -(m, *(n, /(m, n)))))
    in
      abs (k:Int. =(0, app(app(mod, k), 2)))
    end
in
  app (iseven, 7)
end
```

```
----Term----
let [iseven]
    [let [mod]
         [abs(m:Int.abs(n:Int.-(m,*(n,/(m,n)))))]
         [abs(k:Int.=(0,app(app(mod,k),2)))]]
    [app(iseven,7)]

----Type----
Bool

----Normal Form----
false

----DBTerm----
let
    let
      abs(:Int.abs(:Int.-((Index 1), *((Index 0), /((Index 1), (Index 0))))))
      abs(:Int.=(0,app(app((Index 1),(Index 0)),2)))
    app((Index 0),7)

----Natural Semantics with Clo,Env and DB Term----
BoolVal False

----CES Machine Code----
[Close [Close [Access 1,Access 0,Access 1,Access 0,Bop /,Bop *,Bop -,Return],Return],
  Let,Close [Int 0,Access 1,Access 0,Apply,Int 2,Apply,Bpr =,Return],EndLet,
  Let,Access 0,Int 7,Apply,EndLet]

----CES Eval----
BoolVal False

----CPS Form----
abs(kappa:->(Bool,0).app(abs(kappa:->(->(Int,->(->(Bool,0),0)),0).
app(abs(kappa:->(->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)),0).app(kappa,
abs(m:Int.abs(kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,abs(n:Int.
abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,m)),abs(v1:Int.
app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,n)),abs(v1:Int.
app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,m)),abs(v1:Int.
app(abs(kappa:->(Int,0).app(kappa,n)),abs(v2:Int.app(kappa,/(v1,v2))))))),
abs(v2:Int.app(kappa,*(v1,v2))))))),abs(v2:Int.app(kappa,-(v1,v2))))))))))))),
abs(v:->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)).let [mod] [v]
[app(abs(kappa:->(->(Int,->(->(Bool,0),0)),0).app(kappa,abs(k:Int.
abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).app(kappa,0)),abs(v1:Int.
app(abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).
app(abs(kappa:->(->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)),0).app(kappa,mod)),
abs(v1:->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)).app(abs(kappa:->(Int,0).
app(kappa,k)),abs(v2:Int.app(app(v1,v2),kappa)))))),abs(v1:->(Int,->(->(Int,0),0)).
app(abs(kappa:->(Int,0).app(kappa,2)),abs(v2:Int.app(app(v1,v2),kappa)))))),
abs(v2:Int.app(kappa,=(v1,v2)))))))))),kappa)]))),abs(v:->(Int,->(->(Bool,0),0)).
let [iseven] [v] [app(abs(kappa:->(Bool,0).app(
abs(kappa:->(->(Int,->(->(Bool,0),0)),0).app(kappa,iseven)),
```

```
abs(v1:->(Int,->(->(Bool,0),0)).app(abs(kappa:->(Int,0).
app(kappa,7)),abs(v2:Int.app(app(v1,v2),kappa)))))),kappa)])))

----CPS Type----
->(->(Bool,0),0)

---CPS Normal Form----
false

----CE3R Machine Code----
[CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [ACCESS1 0,CLOSE2 [CLOSE1
[ACCESS1 0,ACCESS2 4,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,
ACCESS2 4,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 8,TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2 [ACCESS1 2,DIV2 1 0,TAILAPPLY1],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 2,MUL2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [LET 0 [CLOSE1 [ACCESS1 0,CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],CLOSE2
[CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2
[CLOSE1 [ACCESS1 0,ACCESS2 6,TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 2,TAILAPPLY1],CLOSE2
[ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 2,
EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],ACCESS2 2,TAILAPPLY1],ENDLET],
TAILAPPLY1],CLOSE2 [LET 0 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 2,TAILAPPLY1],CLOSE2
[CLOSE1 [ACCESS1 0,CONST2 7,TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],
TAILAPPLY1],TAILAPPLY1],ACCESS2 2,TAILAPPLY1],ENDLET],TAILAPPLY1],CLOSE2 [ACCESS1 0],TAILAPPLY1]

----CE3R Machine----
BoolVal False
```

7.2 Test 2: isven 7 in fix term form

```
app (fix (abs (ie:->(Int, Bool). abs(x:Int. if =(0, x) then true
  else if =(0, -(x, 1)) then false else app(ie, -(x, 2)) fi fi))), 7)

----Term----
app(fix abs(ie:->(Int,Bool).abs(x:Int.
    if =(0,x) then true else if =(0,-(x,1)) then false else app(ie,-(x,2)) fi fi)),
   7)

----Type----
Bool

----Normal Form----
false

----DBTerm----
app(fix abs(:->(Int,Bool).abs(:Int.
    if =(0,(Index 0))
       then true
       else if =(0,-((Index 0), 1))
               then false
```

```
                else app((Index 1),-((Index 0), 2))
            fi
    fi)),
    7)

----Natural Semantics with Clo,Env and DB Term----
BoolVal False

----CES Machine Code----
[Close [Close [Int 0,Access 0,Bpr =,If,Close [Bool True],
Close [Int 0,Access 0,Int 1,Bop -,Bpr =,If,Close [Bool False],
Close [Access 1,Access 0,Int 2,Bop -,Apply]],Return],Return],Fix,Int 7,Apply]

----CES Eval----
BoolVal False

----CPS Form----
abs(kappa:->(Bool,0).app(abs(kappa:->(->(Int,->(->(Bool,0),0)),0).
app(kappa,fix abs(ie:->(Int,->(->(Bool,0),0)).abs(x:Int.abs(kappa:->(Bool,0).
app(abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).app(kappa,0)),
abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,x)),abs(v2:Int.app(kappa,=(v1,v2))))))),
abs(v:Bool.if v then app(abs(kappa:->(Bool,0).app(kappa,true)),kappa) else app(
abs(kappa:->(Bool,0).app(abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).app(kappa,0)),
abs(v1:Int.app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,x)),abs(v1:Int.
app(abs(kappa:->(Int,0).app(kappa,1)),abs(v2:Int.app(kappa,-(v1,v2))))))),abs(v2:Int.
app(kappa,=(v1,v2))))))),abs(v:Bool.if v then app(abs(kappa:->(Bool,0).
app(kappa,false)),kappa) else app(abs(kappa:->(Bool,0).app(abs(
kappa:->(->(Int,->(->(Bool,0),0)),0).app(kappa,ie)),abs(v1:->(Int,->(->(Bool,0),0)).
app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(
kappa:->(Int,0).app(kappa,2)),abs(v2:Int.app(kappa,-(v1,v2))))))),abs(v2:Int.
app(app(v1,v2),kappa)))))),kappa) fi))),kappa) fi))))))),abs(v1:->(Int,->(->(Bool,0),0)).
app(abs(kappa:->(Int,0).app(kappa,7)),abs(v2:Int.app(app(v1,v2),kappa))))))

----CPS Type----
->(->(Bool,0),0)

---CPS Normal Form----
false

----CE3R Machine Code----
[CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [FIX [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 4,TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,BOOL2 True,TAILAPPLY1],ACCESS2 1,
TAILAPPLY1] [CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],CLOSE2 [CLOSE1
[CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],
CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,
TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,BOOL2 False,TAILAPPLY1],
ACCESS2 1,TAILAPPLY1] [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2 [CLOSE1
[CLOSE1 [ACCESS1 0,ACCESS2 8,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 2,TAILAPPLY1],
CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,
ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1],ACCESS2 1,
```

```
TAILAPPLY1]],TAILAPPLY1]],TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 7,TAILAPPLY1],
CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 0],
TAILAPPLY1]

----CE3R Machine----
BoolVal False
```

7.3 Test 3: calculate 2^3

```
app (app (fix (abs (e:->(Int, ->(Int, Int)). abs (x:Int. abs(y:Int.
  if =(0, y) then 1 else *(x, app(app(e, x), -(y, 1))) fi)))), 2), 3)

----Term----
app(app(
       fix abs(e:->(Int,->(Int,Int)).abs(x:Int.abs(y:Int.
             if =(0,y) then 1 else *(x,app(app(e,x),-(y,1))) fi))),
       2),
    3)

----Type----
Int

----Normal Form----
8

----DBTerm----
app(app(
       fix abs(:->(Int,->(Int,Int)).abs(:Int.abs(:Int.
              if =(0,(Index 0))
                 then 1
                 else *((Index 1),
                        app(app((Index 2),(Index 1)),-((Index 0), 1)))
              fi))),
       2),
    3)

----Natural Semantics with Clo,Env and DB Term----
IntVal 8

----CES Machine Code----
[Close [Close [Close [Int 0,Access 0,Bpr =,If,
Close [Int 1],Close [Access 1,Access 2,Access 1,Apply,Access 0,
Int 1,Bop -,Apply,Bop *],Return],Return],Return],Fix,Int 2,Apply,Int 3,Apply]

----CES Eval----
IntVal 8

----CPS Form----
abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).
app(abs(kappa:->(->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)),0).
app(kappa,fix abs(e:->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)).
```

```
abs(x:Int.abs(kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,abs(y:Int.
abs(kappa:->(Int,0).app(abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).
app(kappa,0)),abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,y)),abs(v2:Int.
app(kappa,=(v1,v2))))))),abs(v:Bool.if v then app(abs(kappa:->(Int,0).
app(kappa,1)),kappa) else app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).app(abs(
kappa:->(->(Int,->(->(Int,0),0)),0).app(abs(
kappa:->(->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)),0).app(kappa,e)),
abs(v1:->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)).app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v2:Int.app(app(v1,v2),kappa)))))),
abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(
abs(kappa:->(Int,0).app(kappa,y)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,1)),abs(v2:Int.app(kappa,-(v1,v2))))))),abs(v2:Int.
app(app(v1,v2),kappa)))))),abs(v2:Int.app(kappa,*(v1,v2))))))),kappa)
fi)))))))))),abs(v1:->(Int,->(->(->(Int,->(->(Int,0),0)),0),0)).app(
abs(kappa:->(Int,0).app(kappa,2)),abs(v2:Int.app(app(v1,v2),kappa)))))),
abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(kappa,3)),
abs(v2:Int.app(app(v1,v2),kappa))))))

----CPS Type----
->(->(Int,0),0)

---CPS Normal Form----
8

----CE3R Machine Code----
[CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [FIX [ACCESS1 0,CLOSE2 [CLOSE1
[CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 4,
TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],ACCESS2 1,TAILAPPLY1]
[CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 6,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [CLOSE1
[ACCESS1 0,ACCESS2 10,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 10,TAILAPPLY1],
CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2
[CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 8,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,
TAILAPPLY1],CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2
[ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 2,
MUL2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1],
TAILAPPLY1]],TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 2,TAILAPPLY1],CLOSE2
[ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [CLOSE1
[ACCESS1 0,CONST2 3,TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 0],TAILAPPLY1]

----CE3R Machine----
IntVal 8
```

7.4 Test 4: calculate $(3!)!$

```
app (fix (abs (f:->(Int, Int). abs (x:Int. if =(0, x) then 1 else
*(x, app(f, -(x, 1))) fi))), app (fix (abs (f:->(Int, Int).
  abs (x:Int. if =(0, x) then 1 else *(x, app(f, -(x, 1))) fi))), 3))
```

```
----Term----
app(fix abs(f:->(Int,Int).abs(x:Int.if =(0,x) then 1 else *(x,app(f,-(x,1))) fi)),
app(fix abs(f:->(Int,Int).abs(x:Int.if =(0,x) then 1 else *(x,app(f,-(x,1))) fi)),3))

----Type----
Int

----Normal Form----
720

----DBTerm----
app(fix abs(:->(Int,Int).abs(:Int.if =(0,(Index 0)) then 1 else *((Index 0),
app((Index 1),-((Index 0), 1))) fi)),app(fix abs(:->(Int,Int).
abs(:Int.if =(0,(Index 0)) then 1 else *((Index 0),
app((Index 1),-((Index 0), 1))) fi)),3))

----Natural Semantics with Clo,Env and DB Term----
IntVal 720

----CES Machine Code----
[Close [Close [Int 0,Access 0,Bpr =,If,Close [Int 1],
Close [Access 0,Access 1,Access 0,Int 1,Bop -,Apply,Bop *],Return],Return],
Fix,Close [Close [Int 0,Access 0,Bpr =,If,Close [Int 1],
Close [Access 0,Access 1,Access 0,Int 1,Bop -,Apply,Bop *],Return],Return],
Fix,Int 3,Apply,Apply]

----CES Eval----
IntVal 720

----CPS Form----
abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,
fix abs(f:->(Int,->(->(Int,0),0)).abs(x:Int.abs(kappa:->(Int,0).app(abs(kappa:->(Bool,0).
app(abs(kappa:->(Int,0).app(kappa,0)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v2:Int.app(kappa,=(v1,v2))))))),abs(v:Bool.if v then
app(abs(kappa:->(Int,0).app(kappa,1)),kappa) else app(abs(kappa:->(Int,0).
app(abs(kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,f)),
abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(abs(
kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,1)),abs(v2:Int.app(kappa,-(v1,v2))))))),abs(v2:Int.app(
app(v1,v2),kappa)))))),abs(v2:Int.app(kappa,*(v1,v2))))))),kappa) fi))))))),
abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(abs(
kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,fix abs(f:->(Int,->(->(Int,0),0)).
abs(x:Int.abs(kappa:->(Int,0).app(abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).
app(kappa,0)),abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,x)),abs(v2:Int.app(
kappa,=(v1,v2))))))),abs(v:Bool.if v then app(abs(kappa:->(Int,0).app(kappa,1)),
kappa) else app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).app(kappa,x)),
abs(v1:Int.app(abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).
app(kappa,f)),abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(abs(
kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,1)),
abs(v2:Int.app(kappa,-(v1,v2))))))),abs(v2:Int.app(app(v1,v2),kappa)))))),
```

```
abs(v2:Int.app(kappa,*(v1,v2))))))),kappa) fi))))))),abs(v1:->(Int,->(->(Int,0),0)).
app(abs(kappa:->(Int,0).app(kappa,3)),abs(v2:Int.app(app(v1,v2),kappa)))))),
abs(v2:Int.app(app(v1,v2),kappa))))))

----CPS Type----
->(->(Int,0),0)

---CPS Normal Form----
720

----CE3R Machine Code----
[CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [FIX [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 4,TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],ACCESS2 1,
TAILAPPLY1] [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 4,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1
[ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 8,TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],
TAILAPPLY1],CLOSE2 [ACCESS1 2,MUL2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,
TAILAPPLY1]],TAILAPPLY1]],TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [FIX
[CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 4,
TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [IF 0
[CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],ACCESS2 1,TAILAPPLY1] [CLOSE1 [CLOSE1 [ACCESS1 0,
ACCESS2 4,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2
[CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 8,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,
TAILAPPLY1],CLOSE2 [ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2
[ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 2,
MUL2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1]],TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 3,TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,
TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 0],TAILAPPLY1]

----CE3R Machine----
IntVal 720
```

7.5 Test 5

```
app (fix (abs (collatz:->(Int, Int). abs (x:Int.
                 if app (fix (abs (ie:->(Int, Bool). abs (x:Int.
                     if =(0, x) then true else
                        if =(1, x) then false else
                                   app (ie, -(x, 2)) fi fi))), x) then app (collatz, /(x, 2)) else
                                      if =(x, 1) then 1 else
                                          app (collatz, +(*(3, x), 1)) fi fi))), 100)


----Term----
app(fix abs(collatz:->(Int,Int).abs(x:Int.
if app(fix abs(ie:->(Int,Bool).abs(x:Int.
if =(0,x) then true else if =(1,x) then false else
app(ie,-(x,2)) fi fi)),x) then app(collatz,/(x,2))
else if =(x,1) then 1 else app(collatz,+(*(3,x),1)) fi fi)),100)
```

```
----Type----
Int

----Normal Form----
1

----DBTerm----
app(fix abs(:->(Int,Int).abs(:Int.if app(fix abs(:->(Int,Bool).
abs(:Int.if =(0,(Index 0)) then true else if =(1,(Index 0))
then false else app((Index 1),-((Index 0), 2)) fi fi)),(Index 0))
then app((Index 1),/((Index 0), 2)) else if =((Index 0),1) then 1
else app((Index 1),+(*(3, (Index 0)), 1)) fi fi)),100)

----Natural Semantics with Clo,Env and DB Term----
IntVal 1

----CES Machine Code----
[Close [Close [Close [Close [Int 0,Access 0,Bpr =,If,Close [Bool True],
Close [Int 1,Access 0,Bpr =,If,Close [Bool False],Close [Access 1,
Access 0,Int 2,Bop -,Apply]],Return],Return],Fix,Access 0,Apply,
If,Close [Access 1,Access 0,Int 2,Bop /,Apply],Close [Access 0,Int 1,
Bpr =,If,Close [Int 1],Close [Access 1,Int 3,Access 0,Bop *,Int 1,
Bop +,Apply]],Return],Return],Fix,Int 100,Apply]

----CES Eval----
IntVal 1

----CPS Form----
abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).
app(kappa,fix abs(collatz:->(Int,->(->(Int,0),0)).abs(x:Int.abs(
kappa:->(Int,0).app(abs(kappa:->(Bool,0).app(abs(k
appa:->(->(Int,->(->(Bool,0),0)),0).app(kappa,fix abs(
ie:->(Int,->(->(Bool,0),0)).abs(x:Int.abs(kappa:->(Bool,0).
app(abs(kappa:->(Bool,0).app(abs(kappa:->(Int,0).app(kappa,0)),
abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,x)),abs(v2:Int.
app(kappa,=(v1,v2))))))),abs(v:Bool.if v then app(abs(kappa:->(Bool,0).
app(kappa,true)),kappa) else app(abs(kappa:->(Bool,0).app(abs(kappa:->(Bool,0).
app(abs(kappa:->(Int,0).app(kappa,1)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v2:Int.app(kappa,=(v1,v2))))))),abs(v:Bool.if v then app(
abs(kappa:->(Bool,0).app(kappa,false)),kappa) else app(abs(kappa:->(Bool,0).
app(abs(kappa:->(->(Int,->(->(Bool,0),0)),0).app(kappa,ie)),abs(
v1:->(Int,->(->(Bool,0),0)).app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).app(kappa,2)),abs(v2:Int.
app(kappa,-(v1,v2))))))),abs(v2:Int.app(app(v1,v2),kappa)))))),kappa) fi))),
kappa) fi))))))),abs(v1:->(Int,->(->(Bool,0),0)).app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v2:Int.app(app(v1,v2),kappa)))))),abs(v:Bool.if v then
app(abs(kappa:->(Int,0).app(abs(kappa:->(->(Int,->(->(Int,0),0)),0).
app(kappa,collatz)),abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).
app(abs(kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,2)),abs(v2:Int.app(kappa,/(v1,v2))))))),abs(v2:Int.app(app(v1,v2),
```

```
kappa)))))),kappa) else app(abs(kappa:->(Int,0).app(abs(kappa:->(Bool,0).
app(abs(kappa:->(Int,0).app(kappa,x)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,1)),abs(v2:Int.app(kappa,=(v1,v2))))))),abs(v:Bool.if v then app(
abs(kappa:->(Int,0).app(kappa,1)),kappa) else app(abs(kappa:->(Int,0).app(
abs(kappa:->(->(Int,->(->(Int,0),0)),0).app(kappa,collatz)),abs(
v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).app(abs(kappa:->(Int,0).
app(abs(kappa:->(Int,0).app(kappa,3)),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,x)),abs(v2:Int.app(kappa,*(v1,v2))))))),abs(v1:Int.app(abs(kappa:->(Int,0).
app(kappa,1)),abs(v2:Int.app(kappa,+(v1,v2))))))),abs(v2:Int.app(app(v1,v2),kappa)))))),
kappa) fi))),kappa) fi))))))),abs(v1:->(Int,->(->(Int,0),0)).app(abs(kappa:->(Int,0).
app(kappa,100)),abs(v2:Int.app(app(v1,v2),kappa))))))

----CPS Type----
->(->(Int,0),0)

---CPS Normal Form----
1

----CE3R Machine Code----
[CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2 [FIX [CLOSE1 [CLOSE1 [ACCESS1 0,CLOSE2
[FIX [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 0,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,
ACCESS2 4,TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,BOOL2 True,TAILAPPLY1],ACCESS2 1,TAILAPPLY1]
[CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,
ACCESS2 6,TAILAPPLY1],CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [IF 0 [CLOSE1 [ACCESS1 0,BOOL2 False,TAILAPPLY1],ACCESS2 1,TAILAPPLY1]
[CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 7,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [ACCESS1 0,
ACCESS2 8,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 2,TAILAPPLY1],CLOSE2
[ACCESS1 2,SUB2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,
ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1],
ACCESS2 1,TAILAPPLY1]],TAILAPPLY1]],TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 4,
TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],
CLOSE2 [IF 0 [CLOSE1 [CLOSE1 [ACCESS1 0,ACCESS2 5,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1
[ACCESS1 0,ACCESS2 6,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 2,TAILAPPLY1],CLOSE2
[ACCESS1 2,DIV2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,
ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1] [CLOSE1 [CLOSE1
[CLOSE1 [ACCESS1 0,ACCESS2 5,TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],
CLOSE2 [ACCESS1 2,EQ2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [IF 0 [CLOSE1
[ACCESS1 0,CONST2 1,TAILAPPLY1],ACCESS2 1,TAILAPPLY1] [CLOSE1 [CLOSE1 [ACCESS1 0,
ACCESS2 7,TAILAPPLY1],CLOSE2 [CLOSE1 [CLOSE1 [CLOSE1 [ACCESS1 0,CONST2 3,TAILAPPLY1],
CLOSE2 [CLOSE1 [ACCESS1 0,ACCESS2 10,TAILAPPLY1],CLOSE2 [ACCESS1 2,MUL2 1 0,TAILAPPLY1],
TAILAPPLY1],TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 1,TAILAPPLY1],CLOSE2 [ACCESS1 2,
ADD2 1 0,TAILAPPLY1],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,ACCESS3 2,TAILAPPLY2],
TAILAPPLY1],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1],ACCESS2 1,TAILAPPLY1]],TAILAPPLY1]],
TAILAPPLY1],CLOSE2 [CLOSE1 [ACCESS1 0,CONST2 100,TAILAPPLY1],CLOSE2 [ACCESS1 1,ACCESS2 0,
ACCESS3 2,TAILAPPLY2],TAILAPPLY1],TAILAPPLY1],CLOSE2 [ACCESS1 0],TAILAPPLY1]

----CE3R Machine----
IntVal 1
```

References

[1] Cormac Flanagan, Amr Sabry, Bruce F. Duda, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI's* 93 [36], pages 237-247, 1993.

[2] Albert R. Meyer and Mitchell Wand. Continuation Semantics in Typed Lambda-Calculi (summary). In *Rohit Parikh, editor, logics of Programs*, vol. 224 of *Lecture Notes in Computer Science*, pages 219-224, Springer-Verlag, 1985.