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

```
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

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 t1 t2 t3, e) = S.If (f (t1, e)) (f (t2, e)) (f (t3, 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:

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

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

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

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

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

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

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

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

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

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

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

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

$$C[\text{(Access i : c, e, s)}] \rightarrow C[\text{(c, e, (e !! i) : s)}]$$

$$C[\text{(If:c, e, s2:s1:(Value True):s)}] \rightarrow C[\text{(c, e, s1:s)}]$$

$$C[\text{(If:c, e, s2:s1:(Value False):s)}] \rightarrow C[\text{(c, e, s2:s)}]$$

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

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

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

$$C[\text{(Return:c, e, s':(Code c'):(Env e'):s)}] \rightarrow C[\text{(c', e', s':s)}]$$

$$C[\text{(Int n:c, e, s)}] \rightarrow C[\text{(c, e, (Value n):s)}]$$

$$C[\text{(Bool b:c, e, s)}] \rightarrow C[\text{(c, e, (Value b):s)}]$$

$$C[\text{(bop:c, e, (Value v2):(Value v1):s)}] \rightarrow C[\text{(c, e, (Value (bop v1 v2)):s)}]$$

$$C[\text{(bpr:c, e, (Value v2):(Value v1):s)}] \rightarrow C[\text{(c, e, (Value (bpr v1 v2)):s)}]$$

$$C[\text{(Let:code, env, (Value v):s)}] \rightarrow C[\text{(code, v:env, s)}]$$

$$C[\text{(Let:code, env, (Env env'):s)}] \rightarrow C[\text{(code, env'++env, s)}]$$

$$C[\text{(EndLet:code, v:env, s)}] \rightarrow C[\text{(code, env, s)}]$$

$C[(\text{Fix:code, env, (Env [Clo code' env']):s})] \rightarrow C[(\text{code', (Clo [Close code', Fix] []):env, (Code code):(Env env):s})]$

3.1.1 Haskell Implementation

```
module CESMachine where
import Debug.Trace
import qualified EvaluationContext as E
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
          deriving Show
type Env = [Value]
data Slot = Value Value | Code Code | Env Env
          deriving Show
type Stack = [Slot]
type State = (Code, Env, Stack)
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 -> case compile t0 of
                        t0' -> case compile t1 of
                          t1' -> case compile t2 of
                            t2' -> t0' ++ t1' ++ t2' ++ [If]
      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 : c, e, s2 : s1 : (Value (BoolVal v0)) : s) -> case v0 of
                                                           True -> Just (c, e, s1 : s)
                                                           False -> Just (c, e, s2 : s)

      (Close code' : code, env, s) -> Just (code, env, Env [Clo code' env] : s)
      (Apply : code, env, (Value v) : (Env [Clo code' env']) : s) -> Just (code', v : env', (Code code) : (Env env) : s)
      (Apply : code, env, (Value v) : (Value (Clo code' env')) : s) -> Just (code', v : 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
                              Eq -> Just (c, e, (Value (BoolVal (I.intEq v1 v2))) : s)
```

$$Lt \to Just\ (c, e, (Value\ (BoolVal\ (I.intLt\ v1\ v2))) : s)$$

$$
\begin{aligned}
&(Let : code, env, (Value\ v) : s) \to Just\ (code, v : env, s)\\
&(Let : code, env, (Env\ env') : s) \to Just\ (code, env' +\!\!+\ env, s)\\
&(EndLet : code, v : env, s) \to Just\ (code, env, s)\\
&(Fix : code, env, (Env\ [Clo\ code'\ env']) : s) \to Just\ (code', (Clo\ [Close\ code', Fix]\ [\,]) : env, (Code\ code) : (Env\ env) : s)\\
&\_ \to Nothing
\end{aligned}
$$

$$
\begin{aligned}
&loop :: State \to State\\
&loop\ state =\\
&\quad \mathbf{case}\ step\ state\ \mathbf{of}\\
&\qquad Just\ state' \to loop\ state'\\
&\qquad Nothing \to state
\end{aligned}
$$

$$
\begin{aligned}
&eval :: DB.Term \to Value\\
&eval\ t = \mathbf{case}\ loop\ (compile\ t, [\,], [\,])\ \mathbf{of}\\
&\qquad (\_, \_, Value\ v : \_) \to v\\
&\qquad \_ \to error\ \texttt{"not a value"}
\end{aligned}
$$

4 CPS

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

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:

$$
\begin{aligned}
\text{conditional terms:}\quad & [\![\ \text{If}\ t_1\ t_2\ t_3]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v.\ \text{If}\ v\ ([\![t_2]\!]\ k)\ ([\![t_3]\!]\ k))\\
\text{let bindings:}\quad & [\![\ \text{Let}\ x\ t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v.\ \text{Let}\ x\ v\ ([\![t_2]\!]\ \kappa))\\
\text{binary operators:}\quad & [\![bop\ t_1\ t_2]\!] = \lambda\kappa.\ [\![t_1]\!]\ (\lambda v_1.\ [\![t_2]\!]\ (\lambda v_2.\ (\kappa\ (bop\ v_1\ v_2))))
\end{aligned}
$$

Besides, for the **fix** terms, we designed two rules below:

$$
\begin{aligned}
[\![\ \text{fix}\ t_1]\!] &= \lambda\kappa.\ [\![t_1]\!]\ (\lambda v.\ (\text{fix}\ v)\ \kappa)\\
[\![\ \text{fix}\ \lambda x.\ t_{11}]\!] &= \lambda\kappa.\ [\![\lambda x.\ t_{11}]\!]\ (\lambda v_1.\ [\![t_1 1]\!]\ (\lambda v_2.\ (\ \text{Let}\ x\ (\text{Fix}\ v_1)\ v_2)\ \kappa))
\end{aligned}
$$

However, these two rules only works for the **fix** terms with no recursion incide. So we need modify the rule to make our CPS module behavior correctly.

In addition, we also implement the type-preserved CPS transformation. First, we implement a function to transform the types into CPS form, i.e. *toCPSType* function in CPS module. Formally, 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. 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 terms.

In order to calculate the type of continuations, we construct the *toCPSWithContext* function, which takes the current typing context Γ as an argument and use *typing* function in **Typing** module to do the work.

The reference are

- *Continuation Semantics in Typed Lambda-Calculi* By Albert Meyer & Mitchell Wand, 1985.
- *The Essence of Compiling with Continuations* By C. Flanagan et al., 1993.

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 "kappa" (S.TypeArrow S.TypeInt answerType) (S.App (S.Var "kappa") t)
S.Tru → S.Abs "kappa" (S.TypeArrow S.TypeBool answerType) (S.App (S.Var "kappa") t)
S.Fls → S.Abs "kappa" (S.TypeArrow S.TypeBool answerType) (S.App (S.Var "kappa") t)
S.Var x → S.Abs "kappa" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                            (S.App (S.Var "kappa") t)
S.App t1 t2 → S.Abs "kappa" (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 "kappa"))))))
S.Abs x τ1 t1 → S.Abs "kappa" (S.TypeArrow (toCPSType (fromJust $ T.typing Γ t) answerType) answerType)
                        (S.App (S.Var "kappa") (S.Abs x (toCPSType τ1 answerType)
                                 (toCPSWithContext answerType t1 (T.Bind Γ x τ1))))
S.If t1 t2 t3 → S.Abs "kappa" (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 "kappa"))
                                      (S.App (toCPSWithContext answerType t3 Γ) (S.Var "kappa")))))
S.Let x t1 t2 → S.Abs "kappa" (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 "kappa")))))
S.Fix t1 →
  case t1 of
    S.Abs x τ11 t11 →
      S.Abs "kappa" (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 t11 (T.Bind Γ x τ11))
              (S.Abs "v2" (toCPSType (fromJust $ T.typing (T.Bind Γ x τ11) t11) answerType)
                (S.App (S.Let x (S.Fix (S.Var "v1")) (S.Var "v2")) (S.Var "kappa"))))))
    _ →
      S.Abs "kappa" (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 "kappa"))))
S.IntAdd t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntAdd (S.Var "v1") (S.Var "v2")))))))
S.IntSub t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntSub (S.Var "v1") (S.Var "v2")))))))
S.IntMul t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntMul (S.Var "v1") (S.Var "v2")))))))
S.IntDiv t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntDiv (S.Var "v1") (S.Var "v2")))))))
S.IntNand t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntNand (S.Var "v1") (S.Var "v2")))))))
    S.IntEq t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntEq (S.Var "v1") (S.Var "v2")))))))
    S.IntLt t1 t2 → S.Abs "kappa" (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 "kappa") (S.IntLt (S.Var "v1") (S.Var "v2")))))))

toCPS :: S.Type → S.Term → S.Term
toCPS answerType t =
  toCPSWithContext answerType t T.Empty
```

5 C-E-3R Compiler and Virtual Machine

C-E-3R machine is consisted of a compiler and a set of transition rules. Since the compiler takes the restricted forms of λ terms as its input, compiling the nameless body into the instructions, we first need to transform the DeBruijn terms into the restricted forms defined by atom and body. After the transformation, all DeBruijn terms can be recognized by C-E-3R's compiler, then we can follow the similar steps as we did in implementing the CES Machine to implement the compiler and transition rules.

The compiler is consisted of two subcompilers, where bcompiler transforms the body into instructions, and acompiler compiles the atom values into Register r_1, r_2, r_3.

Instead of explicitly defining data structures for **Body** and **Atom**, we directly deal with nameless terms and compile from them, since the information from these de Bruijn terms is sufficient for us to make decisions of compiling. We use *bcompile* and *acompile* to generate **Code**, which is a list of Instructions (**Inst**), and then evaluate the term from state to state based on the C-E-3R machine code and related rules discussed in class.

Hitherto, we just implemented the instructions discussed in class. In order to deal with binary operations, let bindings, fixed point functions, and conditional if terms, we need to add more instructions, associated with compilation and evaluation rules. We've made it as our next step for this project.

The implemented codes are as the followings:

```
module CE3RMachine where

import qualified AbstractSyntax as S
import qualified DeBruijn as DB
import qualified CPS as CPS


    -- instructions
data Inst = ACCESS1 Int
          | ACCESS2 Int
          | ACCESS3 Int
          | CONST1 Integer
          | CONST2 Integer
```

```
            | CONST3 Integer
            | CLOSE1 Code
            | CLOSE2 Code
            | CLOSE3 Code
            | TAILAPPLY1
            | TAILAPPLY2
           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]
    _ → error "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 "Code Generating Error"
    DB.IntConst n → case j of
                        1 → CONST1 n
                        2 → CONST2 n
```

```
                        3 → CONST3 n
                        _ → error "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 "Code Generating Error"
    DB.Abs τ1 t1 → case j of
                        1 → CLOSE1 (bcompile t1)
                        2 → CLOSE2 (bcompile t1)
                        3 → CLOSE3 (bcompile t1)
                        _ → error "Code Generating Error"
    _ → error "Unsupported term"
```

```
    -- transition rules
ce3rMachineStep :: State → Maybe State
ce3rMachineStep ((CONST1 n) : c, e, (UNCARE, v2, v3)) = Just (c, e, (IntVal n, v2, v3))
ce3rMachineStep ((CONST2 n) : c, e, (v1, UNCARE, v3)) = Just (c, e, (v1, IntVal n, v3))
ce3rMachineStep ((CONST3 n) : c, e, (v1, v2, UNCARE)) = Just (c, e, (v1, v2, IntVal n))
ce3rMachineStep ((ACCESS1 i) : c, e, (UNCARE, v2, v3)) = Just (c, e, (e !! i, v2, v3))
ce3rMachineStep ((ACCESS2 i) : c, e, (v1, UNCARE, v3)) = Just (c, e, (v1, e !! i, v3))
ce3rMachineStep ((ACCESS3 i) : c, e, (v1, v2, UNCARE)) = Just (c, e, (v1, v2, e !! i))
ce3rMachineStep ((CLOSE1 c') : c, e, (UNCARE, v2, v3)) = Just (c, e, (Clo c' e, v2, v3))
ce3rMachineStep ((CLOSE2 c') : c, e, (v1, UNCARE, v3)) = Just (c, e, (v1, Clo c' e, v3))
ce3rMachineStep ((CLOSE3 c') : c, e, (v1, v2, UNCARE)) = Just (c, e, (v1, v2, Clo c' e))
ce3rMachineStep ((TAILAPPLY1) : c, e, (Clo c' e', v, UNCARE)) = 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 _ = 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
                _ → error "Evaluation Error Occurred!"
```

6 Main

module *Main* **where**

```haskell
import qualified System.Environment
import Data.List
import 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 ("----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 Final state----")
      putStrLn (show (CES.loop (CES.compile dbterm, [], [])) ++ "\n")
      putStrLn ("----CES Eval----")
      putStrLn (show (CES.eval dbterm))


      let answerType = S.TypeInt
      let cpsterm = CPS.toCPS answerType term
      putStrLn ("----CPS Form----")
```

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

let *bodyterm* = (*S.App cpsterm* (*S.Abs* "x" *answerType* (*S.Var* "x")))
putStrLn ("----CE3R DBterm----")
putStrLn (*show* (*DB.toDeBruijn bodyterm*))
putStrLn ("----CE3R Machine----")
putStrLn (*show* (*CE3R.eval* (*DB.toDeBruijn bodyterm*)))

7 Test Cases

7.1 Test 1

```
app (abs (k:Int. =(0, app(app(abs (m:Int. abs(n:Int. -(m, *(n, /(m, n))))), k), 2))), 7)

----Term----
app(abs(k:Int.=(0,app(app(abs(m:Int.abs(n:Int.-(m,*(n,/(m,n))))),k),2))),7)
----Type----
Bool
----DBTerm----
app(abs(:Int.=(0,app(app(abs(:Int.abs(:Int.-((Index 1), *((Index 0), /((Index 1),
(Index 0)))))),(Index 0)),2))),7)
----Natural Semantics with Clo,Env and DB Term----
BoolVal False
----CES Machine Code----
[Close [Int 0,Close [Close [Access 1,Access 0,Access 1,Access 0,Bop /,Bop *,Bop -,
Return],Return],Access 0,Apply,Int 2,Apply,Bpr =,Return],Int 7,Apply]

----CES Final state----
([],[],[Value (BoolVal False)])

----CES Eval----
BoolVal False
----CPS Form----
abs(kappa:->(Bool,Int).app(abs(kappa:->(->(Int,->(->(Bool,Int),Int)),Int).app(kappa,
abs(k:Int.abs(kappa:->(Int,Int).app(abs(kappa:->(Int,Int).app(kappa,0)),abs(v1:Int.app
(abs(kappa:->(Int,Int).app(abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app(abs
(kappa:->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app(kappa,abs
(m:Int.abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app(kappa,abs(n:Int.abs(kappa:->
(Int,Int).app(abs(kappa:->(Int,Int).app(kappa,m)),abs(v1:Int.app(abs(kappa:->(Int,Int).app
(abs(kappa:->(Int,Int).app(kappa,n)),abs(v1:Int.app(abs(kappa:->(Int,Int).app(abs(kappa:->
(Int,Int).app(kappa,m)),abs(v1:Int.app(abs(kappa:->(Int,Int).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(v1:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)).app(abs(kappa:->
(Int,Int).app(kappa,k)),abs(v2:Int.app(app(v1,v2),kappa)))))),abs(v1:->(Int,->(->(Int,Int),
Int)).app(abs(kappa:->(Int,Int).app(kappa,2)),abs(v2:Int.app(app(v1,v2),kappa)))))),
abs(v2:Int.app(kappa,=(v1,v2)))))))))),abs(v1:->(Int,->(->(Bool,Int),Int)).app(abs
(kappa:->(Int,Int).app(kappa,7)),abs(v2:Int.app(app(v1,v2),kappa))))))
```

```
---CPS Normal Form----
false
----CE3R DBterm----
app(abs(:->(Bool,Int).app(abs(:->(->(Int,->(->(Bool,Int),Int)),Int).app((Index 0),
abs(:Int.abs(:->(Int,Int).app(abs(:->(Int,Int).app((Index 0),0)),abs(:Int.app(abs
(:->(Int,Int).app(abs(:->(->(Int,->(->(Int,Int),Int)),Int).app(abs(:->(->(Int,->
(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app((Index 0),abs(:Int.abs(:->(->
(Int,->(->(Int,Int),Int)),Int).app((Index 0),abs(:Int.abs(:->(Int,Int).app(abs(:->
(Int,Int).app((Index 0),(Index 4))),abs(:Int.app(abs(:->(Int,Int).app(abs(:->
(Int,Int).app((Index 0),(Index 4))),abs(:Int.app(abs(:->(Int,Int).app(abs(:->
(Int,Int).app((Index 0),(Index 8))),abs(:Int.app(abs(:->(Int,Int).app((Index 0),
(Index 7))),abs(:Int.app((Index 2),/((Index 1), (Index 0)))))))),abs(:Int.app
((Index 2),*((Index 1), (Index 0)))))))),abs(:Int.app((Index 2),-((Index 1),
(Index 0)))))))))))))),abs(:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)).app
(abs(:->(Int,Int).app((Index 0),(Index 6))),abs(:Int.app(app((Index 1),(Index 0)),
(Index 2))))))),abs(:->(Int,->(->(Int,Int),Int)).app(abs(:->(Int,Int).app((Index 0),2)),
abs(:Int.app(app((Index 1),(Index 0)),(Index 2))))))),abs(:Int.app((Index 2),=((Index 1),
(Index 0)))))))))))),abs(:->(Int,->(->(Bool,Int),Int)).app(abs(:->(Int,Int).app((Index 0),7)),
abs(:Int.app(app((Index 1),(Index 0)),(Index 2))))))),abs(:Int.(Index 0)))
----CE3R Machine----
Main: Unsupported term
```

7.2 Test 2

```
if <(app(abs(x:Int.-(x,1)),2), 0) then true else false fi

----Term----
if <(app(abs(x:Int.-(x,1)),2),0) then true else false fi
----Type----
Bool
----DBTerm----
if <(app(abs(:Int.-((Index 0), 1)),2),0) then true else false fi
----Natural Semantics with Clo,Env and DB Term----
BoolVal False
----CES Machine Code----
[Close [Access 0,Int 1,Bop -,Return],Int 2,Apply,Int 0,Bpr <,Bool True,Bool False,If]

----CES Final state----
([],[],[Value (BoolVal False)])

----CES Eval----
BoolVal False
----CPS Form----
abs(kappa:->(Bool,Int).app(abs(kappa:->(Int,Int).app(abs(kappa:->(Int,Int).app(abs
(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app(kappa,abs(x:Int.abs(kappa:->
(Int,Int).app(abs(kappa:->(Int,Int).app(kappa,x)),abs(v1:Int.app(abs(kappa:->
(Int,Int).app(kappa,1)),abs(v2:Int.app(kappa,-(v1,v2)))))))))),abs(v1:->(Int,->
(->(Int,Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,2)),abs(v2:Int.app(app
(v1,v2),kappa)))))),abs(v1:Int.app(abs(kappa:->(Int,Int).app(kappa,0)),abs
(v2:Int.app(kappa,<(v1,v2))))))),abs(v:Bool.if v then app(abs(kappa:->(Bool,
```

```
Int).app(kappa,true)),kappa) else app(abs(kappa:->(Bool,Int).app(kappa,false)),kappa) fi)))
---CPS Normal Form----
false
----CE3R DBterm----
app(abs(:->(Bool,Int).app(abs(:->(Int,Int).app(abs(:->(Int,Int).app(abs(:->
(->(Int,->(->(Int,Int),Int)),Int).app((Index 0),abs(:Int.abs(:->(Int,Int).app
(abs(:->(Int,Int).app((Index 0),(Index 2))),abs(:Int.app(abs(:->(Int,Int).app(
(Index 0),1)),abs(:Int.app((Index 2),-((Index 1), (Index 0))))))))))),abs(:->
(Int,->(->(Int,Int),Int)).app(abs(:->(Int,Int).app((Index 0),2)),abs(:Int.app
(app((Index 1),(Index 0)),(Index 2))))))),abs(:Int.app(abs(:->(Int,Int).app(
(Index 0),0)),abs(:Int.app((Index 2),<((Index 1),(Index 0)))))))),abs(:Bool.if
(Index 0) then app(abs(:->(Bool,Int).app((Index 0),true)),(Index 1)) else app
(abs(:->(Bool,Int).app((Index 0),false)),(Index 1)) fi))),abs(:Int.(Index 0)))
----CE3R Machine----
Main: Unsupported term
```

7.3 Test 3

```
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
----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 Final state----
([],[],[Value (BoolVal False)])

----CES Eval----
BoolVal False
----CPS Form----
```

```
abs(kappa:->(Bool,Int).app(abs(kappa:->(->(Int,->(->(Bool,Int),Int)),Int).app(abs(kappa:
->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app(kappa,abs(m:Int.abs(kappa:
->(->(Int,->(->(Int,Int),Int)),Int).app(kappa,abs(n:Int.abs(kappa:->(Int,Int).app(abs
(kappa:->(Int,Int).app(kappa,m)),abs(v1:Int.app(abs(kappa:->(Int,Int).app(abs(kappa:->
(Int,Int).app(kappa,n)),abs(v1:Int.app(abs(kappa:->(Int,Int).app(abs(kappa:->(Int,
Int).app(kappa,m)),abs(v1:Int.app(abs(kappa:->(Int,Int).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,Int),Int)),Int),Int)).let mod v app(abs(kappa:->(->(Int,
->(->(Bool,Int),Int)),Int).app(kappa,abs(k:Int.abs(kappa:->(Int,Int).app(abs(kappa:->(Int,
nt).app(kappa,0)),abs(v1:Int.app(abs(kappa:->(Int,Int).app(abs(kappa:->(->(Int,->(->(Int,
Int),Int)),Int).app(abs(kappa:->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),
Int).app(kappa,mod)),abs(v1:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)).app(abs
(kappa:->(Int,Int).app(kappa,k)),abs(v2:Int.app(app(v1,v2),kappa)))))),abs(v1:->(Int,
->(->(Int,Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,2)),abs(v2:Int.app(app(v1,v2),
kappa)))))),abs(v2:Int.app(kappa,=(v1,v2)))))))))),kappa)))),abs(v:->(Int,->(->(Bool,Int),
Int)).let iseven v app(abs(kappa:->(Bool,Int).app(abs(kappa:->(->(Int,->(->(Bool,Int),Int)),
Int).app(kappa,iseven)),abs(v1:->(Int,->(->(Bool,Int),Int)).app(abs(kappa:->(Int,
Int).app(kappa,7)),abs(v2:Int.app(app(v1,v2),kappa)))))),kappa))))
---CPS Normal Form----
false
----CE3R DBterm----
app(abs(:->(Bool,Int).app(abs(:->(->(Int,->(->(Bool,Int),Int)),Int).app(abs(:->(->(Int,
->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app((Index 0),abs(:Int.abs(:->(->(Int,
->(->(Int,Int),Int)),Int).app((Index 0),abs(:Int.abs(:->(Int,Int).app(abs(:->(Int,Int
).app((Index 0),(Index 4))),abs(:Int.app(abs(:->(Int,Int).app(abs(:->(Int,Int).app((Index 0),
(Index 4))),abs(:Int.app(abs(:->(Int,Int).app(abs(:->(Int,Int).app((Index 0),(Index 8))),
abs(:Int.app(abs(:->(Int,Int).app((Index 0),(Index 7))),abs(:Int.app((Index 2),/((Index 1),
(Index 0)))))))),abs(:Int.app((Index 2),*((Index 1), (Index 0)))))))),abs(:Int.app((Index 2),
-((Index 1), (Index 0))))))))))))),abs(:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),
Int)).let (Index 0) app(abs(:->(->(Int,->(->(Bool,Int),Int)),Int).app((Index 0),abs(:Int.abs
(:->(Int,Int).app(abs(:->(Int,Int).app((Index 0),0)),abs(:Int.app(abs(:->(Int,Int).app(abs
(:->(->(Int,->(->(Int,Int),Int)),Int).app(abs(:->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),
Int),Int)),Int).app((Index 0),(Index 7))),abs(:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),
Int)).app(abs(:->(Int,Int).app((Index 0),(Index 6))),abs(:Int.app(app((Index 1),(Index 0)),
(Index 2))))))),abs(:->(Int,->(->(Int,Int),Int)).app(abs(:->(Int,Int).app((Index 0),2)),
abs(:Int.app(app((Index 1),(Index 0)),(Index 2))))))),abs(:Int.app((Index 2),=((Index 1),
(Index 0))))))))))),(Index 2))))),abs(:->(Int,->(->(Bool,Int),Int)).let (Index 0) app(abs
(:->(Bool,Int).app(abs(:->(->(Int,->(->(Bool,Int),Int)),Int).app((Index 0),(Index 2))),
abs(:->(Int,->(->(Bool,Int),Int)).app(abs(:->(Int,Int).app((Index 0),7)),abs(:Int.app(
app((Index 1),(Index 0)),(Index 2))))))),(Index 2))))),abs(:Int.(Index 0)))
----CE3R Machine----
Main: Unsupported term
```

7.4 Test 4

```
app(app(app(abs(x:Int.abs(y:Int.abs(z:Int. +(x, *(y, z))))), 2), 3), 6)
```

```
----Term----
app(app(app(abs(x:Int.abs(y:Int.abs(z:Int.+(x,*(y,z))))),2),3),6)
----Type----
```

```
Int
----DBTerm----
app(app(app(abs(:Int.abs(:Int.abs(:Int.+((Index 2), *((Index 1), (Index 0)))))),2),3),6)
----Natural Semantics with Clo,Env and DB Term----
IntVal 20
----CES Machine Code----
[Close [Close [Close [Access 2,Access 1,Access 0,Bop *,Bop +,Return],Return],Return],
Int 2,Apply,Int 3,Apply,Int 6,Apply]

----CES Final state----
([],[],[Value (IntVal 20)])

----CES Eval----
IntVal 20
----CPS Form----
abs(kappa:->(Int,Int).app(abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app(abs(kappa:->
(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app(abs(kappa:->(->(Int,->(->
(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int),Int)),Int).app(kappa,abs
(x:Int.abs(kappa:->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app
(kappa,abs(y:Int.abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app(kappa,abs(z:Int.abs
(kappa:->(Int,Int).app(abs(kappa:->(Int,Int).app(kappa,x)),abs(v1:Int.app(abs(kappa:->
(Int,Int).app(abs(kappa:->(Int,Int).app(kappa,y)),abs(v1:Int.app(abs(kappa:->
(Int,Int).app(kappa,z)),abs(v2:Int.app(kappa,*(v1,v2))))))),abs(v2:Int.app(kappa,+
(v1,v2))))))))))))))),abs(v1:->(Int,->(->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),
Int),Int)),Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,2)),abs(v2:Int.app(app(v1,
v2),kappa)))))),abs(v1:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)).app(abs
(kappa:->(Int,Int).app(kappa,3)),abs(v2:Int.app(app(v1,v2),kappa)))))),abs(v1:->(Int,
->(->(Int,Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,6)),abs(v2:Int.app(app(v1,v2),
kappa))))))
---CPS Normal Form----
20
----CE3R DBterm----
app(abs(:->(Int,Int).app(abs(:->(->(Int,->(->(Int,Int),Int)),Int).app(abs(:->(->(Int,
->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app(abs(:->(->(Int,->(->(->(Int,->
(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int),Int)),Int).app((Index 0),abs(:Int.abs
(:->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int).app((Index 0),abs
(:Int.abs(:->(->(Int,->(->(Int,Int),Int)),Int).app((Index 0),abs(:Int.abs(:->
(Int,Int).app(abs(:->(Int,Int).app((Index 0),(Index 6))),abs(:Int.app(abs(:->
(Int,Int).app(abs(:->(Int,Int).app((Index 0),(Index 6))),abs(:Int.app(abs(:->
(Int,Int).app((Index 0),(Index 5))),abs(:Int.app((Index 2),*((Index 1),
(Index 0)))))))),abs(:Int.app((Index 2),+((Index 1), (Index 0))))))))))))))))),
abs(:->(Int,->(->(->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)),Int),Int)
).app(abs(:->(Int,Int).app((Index 0),2)),abs(:Int.app(app((Index 1),(Index 0)),
(Index 2))))))),abs(:->(Int,->(->(->(Int,->(->(Int,Int),Int)),Int),Int)).app(abs
(:->(Int,Int).app((Index 0),3)),abs(:Int.app(app((Index 1),(Index 0)),(Index 2))
))))),abs(:->(Int,->(->(Int,Int),Int)).app(abs(:->(Int,Int).app((Index 0),6)),
abs(:Int.app(app((Index 1),(Index 0)),(Index 2))))))),abs(:Int.(Index 0)))
----CE3R Machine----
Main: Unsupported term
```

7.5 Test 5

```
app (abs (x: Int . 1234), 10)

----Term----
app(abs(x:Int.1234),10)
----Type----
Int
----DBTerm----
app(abs(:Int.1234),10)
----Natural Semantics with Clo,Env and DB Term----
IntVal 1234
----CES Machine Code----
[Close [Int 1234,Return],Int 10,Apply]

----CES Final state----
([],[],[Value (IntVal 1234)])

----CES Eval----
IntVal 1234
----CPS Form----
abs(kappa:->(Int,Int).app(abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app
(kappa,abs(x:Int.abs(kappa:->(Int,Int).app(kappa,1234))))),abs(v1:->(Int,->
(->(Int,Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,10)),abs(v2:Int.app
(app(v1,v2),kappa))))))
---CPS Normal Form----
1234
----CE3R DBterm----
app(abs(:->(Int,Int).app(abs(:->(->(Int,->(->(Int,Int),Int)),Int).app((Index 0),
abs(:Int.abs(:->(Int,Int).app((Index 0),1234))))),abs(:->(Int,->(->(Int,Int),
Int)).app(abs(:->(Int,Int).app((Index 0),10)),abs(:Int.app(app((Index 1),
(Index 0)),(Index 2))))))),abs(:Int.(Index 0)))
----CE3R Machine----
IntVal 1234
```

7.6 Test 6

```
app(abs(x:Int. x), 5)

----Term----
app(abs(x:Int.x),5)
----Type----
Int
----DBTerm----
app(abs(:Int.(Index 0)),5)
----Natural Semantics with Clo,Env and DB Term----
IntVal 5
----CES Machine Code----
[Close [Access 0,Return],Int 5,Apply]

----CES Final state----
```

```
([],[],[Value (IntVal 5)])

----CES Eval----
IntVal 5
----CPS Form----
abs(kappa:->(Int,Int).app(abs(kappa:->(->(Int,->(->(Int,Int),Int)),Int).app
(kappa,abs(x:Int.abs(kappa:->(Int,Int).app(kappa,x))))),abs(v1:->(Int,->
(->(Int,Int),Int)).app(abs(kappa:->(Int,Int).app(kappa,5)),abs(v2:Int.app
(app(v1,v2),kappa))))))
---CPS Normal Form----
5
----CE3R DBterm----
app(abs(:->(Int,Int).app(abs(:->(->(Int,->(->(Int,Int),Int)),Int).app((Index 0),
abs(:Int.abs(:->(Int,Int).app((Index 0),(Index 1)))))),abs(:->(Int,->(->(Int,Int),
Int)).app(abs(:->(Int,Int).app((Index 0),5)),abs(:Int.app(app((Index 1),(Index 0)),
(Index 2))))))),abs(:Int.(Index 0)))
----CE3R Machine----
IntVal 5
```