Exercise 1, CS 555

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

Due: February 27th, 2012

1 Concrete Lambda Language

We start the project with a small core lambda language, consisting of the lambda calculus with booleans and integers. Here is the concrete syntax in BNF:

```
Type --> arr lpar Type comma Type rpar
       | Bool_keyword
       | Int_keyword

Term --> identifier
       | abs_keyword lpar identifier colon Type fullstop Term rpar
       | app_keyword lpar Term comma Term rpar
       | true_keyword
       | false_keyword
       | if_keyword Term then_keyword Term else_keyword Term fi_keyword
       | inliteral
       | plus lpar Term comma Term rpar
       | minus lpar Term comma Term rpar
       | div lpar Term comma Term rpar
       | nand lpar Term comma Term rpar
       | equal lpar Term comma Term rpar
       | lt lpar Term comma Term rpar
       | lpar Term rpar
```

Here are the terminal symbols used in the grammar above:

arrow	->
lpar	(
comma	,
rpar	)
Bool_keyword	Bool
Int_keyword	Int
identifier	an identifier, as in Haskell
abs_keyword	abs
colon	:
fullstop	.
app_keyword	app
true_keyword	true
false_keyword	false
if_keyword	if
then_keyword	then
else_keyword	else
fi_keyword	fi
inliteral	a non-negative decimal numeral
plus	+
minus	-
mul	*
div	/
nand	^
equal	=
lt	<

White space, such as space, tab, and newline characters, is permitted between tokens. White space is required between adjacent keyword tokens.

Here are some example programs:

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

---

if true then true else false fi

---

if =(0,0) then 8 else 9 fi

/(4294967295,76)

---

2 Lexer and Parser

1):Lexer takes input string, returns a list of tokens. We define a data structure called Token as the followings,together with the show functions:

```
data Token = ARROW
           | LPAR
           | COMMA
           | RPAR
           | BOOL
           | INT
           | ABS
           | COLON
           | FULLSTOP
           | APP
           | TRUE
           | FALSE
           | IF
           | THEN
           | ELSE
           | FI
           | PLUS
           | SUB
           | MUL
           | DIV
           | NAND
           | EQUAL
           | LT_keyword
           | ID String
           | NUM String
           deriving Eq

instance Show Token where
           show ARROW = "->"
           show LPAR = "("
           show COMMA = ","
           show RPAR = ")"
           show BOOL = "Bool"
           show INT = "Int"
           show ABS = "abs"
           show COLON = ":"
           show FULLSTOP = "."
           show APP = "app"
           show TRUE = "true"
           show FALSE = "false"
```

```
show IF = "if"
show THEN = "then"
show ELSE = "else"
show FI = "fi"
show PLUS = "+"
show SUB = "-"
show MUL = "*"
show DIV = "/"
show NAND = "^"
show EQUAL = "="
show LT_keyword = "<"
show (ID id) = id
show (NUM num) = num
```

For the Token identifier and decimal number, we use regular expression to recognize them, so we have two corresponding subscan function to deal with them.

```
    -- reguar expresiion
ex_num = mkRegex "(0|[1-9][0-9]*)"
ex_id = mkRegex "([a-zA-Z][a-zA-Z0-9_]*)"

    -- subscan for id
subscan1 :: String → Maybe ([Token], String)
subscan1 str = case (matchRegexAll ex_id str) of
                   Just (a1, a2, a3, a4) → case a1 of
                                              "" → Just ([ID a2], a3)
                                              _ → Nothing
                   Nothing → Nothing

    -- subscan for num
subscan2 :: String → Maybe ([Token], String)
subscan2 str = case (matchRegexAll ex_num str) of
                   Just (a1, a2, a3, a4) → case a1 of
                                              "" → Just ([NUM a2], a3)
                                              _ → Nothing
                   Nothing → Nothing
```

Function scan takes an input string and returns a list tokens. If unexpected symbols exists,or the input string cannot mactch any defined token, the function reports errors and the program stops at the lexer level.

```
    -- lexer
scan :: String → [Token]
scan "" = []
    -- white spase
```

```haskell
scan (' ':xs) = scan xs
scan ('\t':xs) = scan xs
scan ('\n':xs) = scan xs
      -- keyword
scan (':':xs) = [COLON] ++ scan xs
scan ('-':'>':xs) = [ARROW] ++ scan xs
scan ('(':xs) = [LPAR] ++ scan xs
scan (',':xs) = [COMMA] ++ scan xs
scan (')':xs) = [RPAR] ++ scan xs
scan ('B':'o':'o':'l':xs) = [BOOL] ++ scan xs
scan ('I':'n':'t':xs) = [INT] ++ scan xs
scan ('a':'b':'s':xs) = [ABS] ++ scan xs
scan ('a':'p':'p':xs) = [APP] ++ scan xs
scan ('.':xs) = [FULLSTOP] ++ scan xs
scan ('t':'r':'u':'e':xs) = [TRUE] ++ scan xs
scan ('f':'a':'l':'s':'e':xs) = [FALSE] ++ scan xs
scan ('i':'f':xs) = [IF] ++ scan xs
scan ('t':'h':'e':'n':xs) = [THEN] ++ scan xs
scan ('e':'l':'s':'e':xs) = [ELSE] ++ scan xs
scan ('f':'i':xs) = [FI] ++ scan xs
scan ('+':xs) = [PLUS] ++ scan xs
scan ('-':xs) = [SUB] ++ scan xs
scan ('*':xs) = [MUL] ++ scan xs
scan ('/':xs) = [DIV] ++ scan xs
scan ('^':xs) = [NAND] ++ scan xs
scan ('=':xs) = [EQUAL] ++ scan xs
scan ('<':xs) = [LT_keyword] ++ scan xs
      -- id and num
scan str = case subscan1 str of
                Nothing -> case subscan2 str of
                                Nothing -> error "[Scan]err: unexpected symbols!"
                                Just (tok, xs) -> tok ++ scan xs
                Just (tok, xs) -> tok ++ scan xs
str str = error "[Scan]err: unexpected symbols!"
```

2):Parser takes a list of tokens, returns a term. We define the two data structures Type and Term, and two functions parseType and parseTerm to deal with them.
parseType function returns a matched Type and the remaining tokens, parseTerm function returns a matched Term and the remaining tokens.

Data structure:

```haskell
data Type = TypeArrow Type Type
          | TypeBool
          | TypeInt
```

```
    deriving Eq

instance Show Type where
    show (TypeArrow τ1 τ2) = "->(" ++ show τ1 ++ "," ++ show τ2 ++ ")"
    show TypeBool = "Bool"
    show TypeInt = "Int"

type Var = String
data Term = Var Var
    | Abs Var Type Term
    | App Term Term
    | Tru
    | Fls
    | If Term Term Term
    | IntConst Integer
    | IntAdd Term Term
    | IntSub Term Term
    | IntMul Term Term
    | IntDiv Term Term
    | IntNand Term Term
    | IntEq Term Term
    | IntLt Term Term
    deriving Eq

instance Show Term where
    show (Var x) = x
    show (Abs x τ t) = "abs(" ++ x ++ ":" ++ show τ ++ "." ++ show t ++ ")"
    show (App t1 t2) = "app(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show Tru = "true"
    show Fls = "false"
    show (If t1 t2 t3) = "if " ++ show t1 ++ " then " ++ show t2 ++ " else " ++ show t3 ++ " fi"
    show (IntConst n) = show n
    show (IntAdd t1 t2) = "+(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntSub t1 t2) = "-(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntMul t1 t2) = "*(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntDiv t1 t2) = "/(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntNand t1 t2) = "^(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntEq t1 t2) = "=(" ++ show t1 ++ "," ++ show t2 ++ ")"
    show (IntLt t1 t2) = "<(" ++ show t1 ++ "," ++ show t2 ++ ")"
```

Function parseType, parseTerm and parse:

```
-- parser

-- type parser
```

```
parseType :: [Token] → Maybe (Type, [Token])
parseType (BOOL : ty) = Just (TypeBool, ty)
parseType (INT : ty) = Just (TypeInt, ty)
parseType (RPAR : ty) = parseType ty
parseType (COMMA : ty) = parseType ty
parseType (ARROW : LPAR : ty) =
        case parseType ty of
                Just (t1, (COMMA : tl)) → case parseType tl of
                                                  Just (t2, (RPAR : tll)) → Just ((TypeArrow t1 t2), tll)
                                                  Nothing → Nothing
                                  Nothing → Nothing
parseType tok = error "[P]err: type parsing error!"

    -- term parser
parseTerm :: [Token] → Maybe (Term, [Token])
    -- id
parseTerm ((ID id) : ts) = Just ((Var id), ts)
    -- num
parseTerm ((NUM num) : ts) = Just ((IntConst (read num :: Integer)), ts)
    -- symbol
    -- parseTerm (COMMA:ts) = parseTerm ts
    -- parseTerm (COLON:ts) = parseTerm ts
    -- parseTerm (RPAR:ts) = parseTerm ts
    -- parseTerm (FULLSTOP:ts) = parseTerm ts
    -- keyword
parseTerm (THEN : ts) = parseTerm ts
parseTerm (ELSE : ts) = parseTerm ts
parseTerm (FI : ts) = parseTerm ts
parseTerm (TRUE : ts) = Just (Tru, ts)
parseTerm (FALSE : ts) = Just (Fls, ts)
    -- (term)
parseTerm (LPAR : ts) = case parseTerm ts of
                            Just (t, (RPAR : tl)) → Just (t, tl)
                            Nothing → Nothing
                            _ → error "[P]err: t is not a term in the (t)"
    -- op
parseTerm (PLUS : LPAR : ts) =
        case parseTerm ts of
                Just (t1, (COMMA : tl)) → case parseTerm tl of
                                                  Just (t2, (RPAR : tll)) → Just ((IntAdd t1 t2), tll)
                                                  Nothing → Nothing
                                                  _ → error "[P]err: plus term"
                Nothing → Nothing
                _ → error "[P]err: plus term"
parseTerm (SUB : LPAR : ts) =
```

```
    case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntSub t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: sub term"
        Nothing -> Nothing
        _ -> error "[P]err: sub term"
parseTerm (MUL : LPAR : ts) =
    case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntMul t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: mul term"
        Nothing -> Nothing
        _ -> error "[P]err: mul term"
parseTerm (DIV : LPAR : ts) =
     case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntDiv t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: div term"
        Nothing -> Nothing
        _ -> error "[P]err: div term"
parseTerm (NAND : LPAR : ts) =
    case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntNand t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: nand term"
        Nothing -> Nothing
        _ -> error "[P]err: nand term"
parseTerm (EQUAL : LPAR : ts) =
     case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntEq t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: eq term"
        Nothing -> Nothing
        _ -> error "[P]err: eq term"
parseTerm (LT_keyword : LPAR : ts) =
    case parseTerm ts of
        Just (t1, (COMMA : tl)) -> case parseTerm tl of
                                        Just (t2, (RPAR : tll)) -> Just ((IntLt t1 t2), tll)
                                        Nothing -> Nothing
                                        _ -> error "[P]err: lt term"
```

```
                Nothing → Nothing
                _ → error "[P]err: lt term"
    -- if-then-else
parseTerm (IF : ts) =
        case parseTerm ts of
                Just (t1, (THEN : tl)) → case parseTerm tl of
                                            Just (t2, (ELSE : tll)) → case parseTerm tll of
                                                                          Just (t3, (FI : tn)) → Just ((If t1 t2 t3), tn)
                                                                          Nothing → Nothing
                                                                          _ → error "[P]err: if term"
                                            Nothing → Nothing
                                            _ → error "[P]err: if term"
                Nothing → Nothing
                _ → error "[P]err: if term"

    -- abs
parseTerm (ABS : LPAR : (ID id) : COLON : ts) =
        case parseType ts of
                Just (ty, (FULLSTOP : tl)) → case parseTerm tl of
                                                 Just (t, (RPAR : tll)) → Just ((Abs id ty t), tll)
                                                 Nothing → Nothing
                                                 _ → error "[P]err: abs term"
                 Nothing → Nothing
                 _ → error "[P]err: abs term"
    -- app
parseTerm (APP : LPAR : ts) = case parseTerm ts of
                                Just (t1, (COMMA : tl)) → case parseTerm tl of
                                                              Just (t2, (RPAR : tll)) → Just ((App t1 t2), tll)
                                                              Nothing → Nothing
                                                              _ → error "[P]err: app term"
                                Nothing → Nothing
                                _ → error "[P]err: app term"
    -- otherwise
parseTerm tok = Nothing

    -- parser
parse :: [Token] → Term
parse t =
          case parseTerm t of
                Just (x, t) → case t of
                                [] → x
                                _ → error "parsing error!"
                Nothing → error "parsing error!"
```

If the input string can't match any defined Term, function parser reports an error and the program stops at the parser level.

3 Binding and Free Variables

Define functions to manipulate the abstract syntax. Place them together with the above type definitions in a module *AbstractSyntax*.

Enumerate the free variables of a term:

$$
\begin{array}{l}
fv :: Term \rightarrow [Var] \\
fv\ (Var\ x) = [x] \\
fv\ (Abs\ x\ \_\ t) = filter\ (\not\equiv x)\ (fv\ t) \\
fv\ (App\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (If\ t_1\ t_2\ t_3) = (fv\ t_1) + (fv\ t_2) + (fv\ t_3) \\
fv\ (IntAdd\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntSub\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntMul\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntDiv\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntNand\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntEq\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ (IntLt\ t_1\ t_2) = (fv\ t_1) + (fv\ t_2) \\
fv\ \_ = [\,]
\end{array}
$$

Substitution: subst $x\ s\ t$, or in writing $[x7 \rightarrow s]t$, is the result of substituting s for x in t.

$$
\begin{array}{l}
subst :: Var \rightarrow Term \rightarrow Term \rightarrow Term \\
subst\ x\ s\ (Var\ v) = \mathbf{if}\ x \equiv v\ \mathbf{then}\ s\ \mathbf{else}\ (Var\ v) \\
subst\ x\ s\ (Abs\ y\ \tau\ t_1) = \\
\quad \mathbf{if}\ x \equiv y\ \mathbf{then} \\
\qquad\qquad\qquad Abs\ y\ \tau\ t_1 \\
\qquad\qquad \mathbf{else} \\
\qquad\qquad\quad Abs\ y\ \tau\ (subst\ x\ s\ t_1) \\
subst\ x\ s\ (App\ t_1\ t_2) = App\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (If\ t_1\ t_2\ t_3) = If\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2)\ (subst\ x\ s\ t_3) \\
subst\ x\ s\ (IntAdd\ t_1\ t_2) = IntAdd\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntSub\ t_1\ t_2) = IntSub\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntMul\ t_1\ t_2) = IntMul\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntDiv\ t_1\ t_2) = IntDiv\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntNand\ t_1\ t_2) = IntNand\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntEq\ t_1\ t_2) = IntEq\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ (IntLt\ t_1\ t_2) = IntLt\ (subst\ x\ s\ t_1)\ (subst\ x\ s\ t_2) \\
subst\ x\ s\ t = t
\end{array}
$$

Syntactic values: primitive constants and abstractions are values.

$$
\begin{array}{l}
isValue :: Term \rightarrow Bool \\
isValue\ (Abs\ \_\ \_\ \_) = True
\end{array}
$$

```
isValue Tru = True
isValue Fls = True
isValue (IntConst _) = True
isValue _ = False
```

4 Structural Operational Semantics

Express the small-step semantics, as defined in class, in Haskell code. The completed source code is as follows:

```
module StructuralOperationalSemantics where
import List
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I

eval1 :: S.Term -> Maybe S.Term
    -- E-IFTRUE
eval1 (S.If S.Tru t2 t3) = Just t2

    -- E-IFFALSE
eval1 (S.If S.Fls t2 t3) = Just t3

    -- E-IF
eval1 (S.If t1 t2 t3) =
  case eval1 t1 of
    Just t1' -> Just (S.If t1' t2 t3)
    Nothing -> Nothing

    -- E-APPABS, E-APP1 and E-APP2
eval1 (S.App t1 t2) =
  if S.isValue t1
    then if S.isValue t2
           then case t1 of
                  S.Abs x tau11 t12 -> Just (S.subst x t2 t12)     -- E-APPABS
                  _ -> Nothing
           else case eval1 t2 of
                  Just t2' -> Just (S.App t1 t2')     -- E-APP2
                  Nothing -> Nothing
    else case eval1 t1 of
           Just t1' -> Just (S.App t1' t2)     -- E-APP1
           Nothing -> Nothing

eval1 (S.IntAdd t1 t2) =
  if S.isValue t1
    then case t1 of
```

```
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → Just (S.IntConst (I.intAdd n1 n2))
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2′ → Just (S.IntAdd t_1 t2′)
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1′ → Just (S.IntAdd t1′ t_2)
            Nothing → Nothing

eval1 (S.IntSub t_1 t_2) =
   if S.isValue t_1
      then case t_1 of
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → Just (S.IntConst (I.intSub n1 n2))
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2′ → Just (S.IntSub t_1 t2′)
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1′ → Just (S.IntSub t1′ t_2)
            Nothing → Nothing

eval1 (S.IntMul t_1 t_2) =
   if S.isValue t_1
      then case t_1 of
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → Just (S.IntConst (I.intMul n1 n2))
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2′ → Just (S.IntMul t_1 t2′)
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1′ → Just (S.IntMul t1′ t_2)
            Nothing → Nothing

eval1 (S.IntDiv t_1 t_2) =
   if S.isValue t_1
      then case t_1 of
```

```
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → Just (S.IntConst (I.intDiv n1 n2))
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2' → Just (S.IntDiv t_1 t2')
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1' → Just (S.IntDiv t1' t_2)
            Nothing → Nothing

eval1 (S.IntNand t_1 t_2) =
  if S.isValue t_1
    then case t_1 of
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → Just (S.IntConst (I.intNand n1 n2))
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2' → Just (S.IntNand t_1 t2')
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1' → Just (S.IntNand t1' t_2)
            Nothing → Nothing

eval1 (S.IntEq t_1 t_2) =
  if S.isValue t_1
    then case t_1 of
            S.IntConst n1 → if S.isValue t_2
                              then case t_2 of
                                     S.IntConst n2 → case I.intEq n1 n2 of
                                                       True → Just S.Tru
                                                       _ → Just S.Fls
                                     _ → Nothing
                              else case eval1 t_2 of
                                     Just t2' → Just (S.IntEq t_1 t2')
                                     Nothing → Nothing
            _ → Nothing
      else case eval1 t_1 of
            Just t1' → Just (S.IntEq t1' t_2)
            Nothing → Nothing

eval1 (S.IntLt t_1 t_2) =
```

```
  if S.isValue t1
    then case t1 of
           S.IntConst n1 → if S.isValue t2
                             then case t2 of
                                    S.IntConst n2 → case I.intLt n1 n2 of
                                                      True → Just S.Tru
                                                      _ → Just S.Fls
                                    _ → Nothing
                             else case eval1 t2 of
                                    Just t2′ → Just (S.IntLt t1 t2′)
                                    Nothing → Nothing
           _ → Nothing
    else case eval1 t1 of
           Just t1′ → Just (S.IntLt t1′ t2)
           Nothing → Nothing

    -- All other cases
eval1 _ = Nothing

eval :: S.Term → S.Term
eval t =
  case eval1 t of
    Just t′ → eval t′
    Nothing → t
```

5 Arithmetic

The module *IntegerArithmetic* formalizes the *primitive* operators for integer arithmetic. In a nutshell, even though we use the Haskell infinite-precision type Integer to store integers, the numbers are really only using the 32-bit 2's complement range, and arithmetic operations must work accordingly. Roughly speaking, arithmetic is as in C on a 32-bit machine. Complete the code.

```
module IntegerArithmetic where
import Data.Bits

intRestrictRangeAddMul :: Integer → Integer
intRestrictRangeAddMul m = m ‘mod‘ 4294967296

intAdd :: Integer → Integer → Integer
intAdd m n = intRestrictRangeAddMul (m + n)

intSub :: Integer → Integer → Integer
intSub m n = m − n

intMul :: Integer → Integer → Integer
```

```
intMul m n = intRestrictRangeAddMul (m ∗ n)

intDiv :: Integer → Integer → Integer
intDiv m n = if n ≡ 0 then error "integer division by zero" else m ‘div‘ n

intNand :: Integer → Integer → Integer
intNand m n = complement (m .&. n)

intEq :: Integer → Integer → Bool
intEq m n = m ≡ n

intLt :: Integer → Integer → Bool
intLt m n = m < n
```

6 Type Checker

It is always good to be sure a program is well-typed before we try to evaluate it. You can use the following type checker or write your own.

```
module Typing where
import qualified AbstractSyntax as S
import List
data Context = Empty
             | Bind Context S.Var S.Type
             deriving Eq
instance Show Context where
  show Empty = "<>"
  show (Bind Γ x τ) = show Γ ++ "," ++ x ++ ":" ++ show τ

contextLookup :: S.Var → Context → Maybe S.Type
contextLookup x Empty = Nothing
contextLookup x (Bind Γ y τ)
  | x ≡ y = Just τ
  | otherwise = contextLookup x Γ

typing :: Context → S.Term → Maybe S.Type
    -- T-Var
typing Γ (S.Var x) = contextLookup x Γ
    -- T-Abs
typing Γ (S.Abs x tau_1 t2) = case typing (Bind Γ x tau_1) t2 of
                  Just (tp0) → Just (S.TypeArrow tau_1 tp0)
                  Nothing → Nothing
typing Γ (S.App t0 t2) =
        case typing Γ t0 of
                  Just (S.TypeArrow tp tp0) → case typing Γ t2 of
```

```
                                                  Just tp' → if tp ≡ tp'
                                                              then Just tp0
                                                              else Nothing
                                                  Nothing → Nothing
              _ → Nothing

   -- T-True
typing Γ S.Tru = Just S.TypeBool

   -- T-False
typing Γ S.Fls = Just S.TypeBool

   -- T-If
typing Γ (S.If t0 t2 t3)
      | (typing Γ t2 ≡ typing Γ t3 ∧ typing Γ t0 ≡ Just S.TypeBool) = typing Γ t2
      | otherwise = Nothing

typing Γ (S.IntConst _) = Just S.TypeInt

   -- T-IntAdd
typing Γ (S.IntAdd t1 t2) =
      case typing Γ t1 of
            Just S.TypeInt → case typing Γ t1 of
                                    Just S.TypeInt → Just S.TypeInt
                                    Nothing → Nothing
   -- T-IntSub
typing Γ (S.IntSub t1 t2) =
      case typing Γ t1 of
            Just S.TypeInt → case typing Γ t1 of
                                    Just S.TypeInt → Just S.TypeInt
                                    Nothing → Nothing
   -- T-IntMul
typing Γ (S.IntMul t1 t2) =
      case typing Γ t1 of
            Just S.TypeInt → case typing Γ t1 of
                                    Just S.TypeInt → Just S.TypeInt
                                    Nothing → Nothing
   -- T-IntDiv
typing Γ (S.IntDiv t1 t2) =
      case typing Γ t1 of
            Just S.TypeInt → case typing Γ t1 of
                                    Just S.TypeInt → Just S.TypeInt
                                    Nothing → Nothing
   -- T-IntNand
typing Γ (S.IntNand t1 t2) =
```

```
      case typing Γ t1 of
            Just S.TypeInt → case typing Γ t1 of
                                  Just S.TypeInt → Just S.TypeInt
                                  Nothing → Nothing
   -- T-IntEq
typing Γ (S.IntEq t1 t2) =
      case typing Γ t1 of
            Just S.TypeBool → case typing Γ t1 of
                                  Just S.TypeBool → Just S.TypeBool
                                  Nothing → Nothing
   -- T-IntLt
typing Γ (S.IntLt t1 t2) =
      case typing Γ t1 of
            Just S.TypeBool → case typing Γ t1 of
                                  Just S.TypeBool → Just S.TypeInt
                                  Nothing → Nothing
typeCheck :: S.Term → S.Type
typeCheck t =
  case typing Empty t of
    Just τ → τ
    _ → error "type error"
```

7 Main Program

Write a main program which will (1) read the program text from a file into a string, (2) invoke the parser to produce an abstract syntax tree for the program, (3) type-check the program, and (4) evaluate the program using the small-step evaluation relation.

```
module Main where

import qualified System.Environment
import Data.List
import IO
import qualified AbstractSyntax as S
import qualified StructuralOperationalSemantics as E
import qualified NaturalSemantics as N
import qualified IntegerArithmetic as I
import qualified Typing as T

main :: IO ()
main =
    do
      args ← System.Environment.getArgs
      let [sourceFile] = args
      source ← readFile sourceFile
```

```
let tokens = S.scan source
let term = S.parse tokens
putStrLn ("----Term----")
putStrLn (show term)
putStrLn ("----Type----")
putStrLn (show (T.typeCheck term))
putStrLn ("----Normal Form in Structureal Operational Semantics----")
putStrLn (show (E.eval term))
putStrLn ("----Normal Form of Natural Semantics----")
putStrLn (show (N.eval term))
```

8 Structural Operational Semantics

Formally stating the rules that give the structural operational semantics of the core lambda language, the rules are listed below:

$$\text{if true then } t_2 \text{ else } t_3 \to t_2 \quad \text{(E-IfTrue)}$$

$$\text{if false then } t_2 \text{ else } t_3 \to t_3(\quad \text{E-IfFalse)}$$

$$\frac{t_1 \to t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \to \text{if } t_1' \text{ then } t_2 \text{ else} t_3} \quad \text{(E-If)}$$

$$\frac{t_1 \to t_1'}{t_1 \ t_2 \to t_1' \ t_2} \quad \text{E-App1}$$

$$\frac{t_2 \to t_2'}{t_1 \ t_2 \to t_1 \ t_2'} \quad \text{E-App2}$$

$$(\lambda x : T_{11}.t_{12}) v_2 \to [x \mapsto v_2]_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \to t_1'}{+(t_1, t_2) \to +(t_1', t_2)} \quad \text{(E-IntAdd1)}$$

$$\frac{t_2 \to t_2'}{+(t_1, t_2) \to +(t_1, t_2')} \quad \text{(E-IntAdd2)}$$

$$+(v_1, v_2) \to v_1 \widetilde{+} v_2 \quad \text{(E-IntAppAdd)}$$

$$\frac{t_1 \to t_1'}{-(t_1, t_2) \to -(t_1', t_2)} \quad \text{(E-IntSub1)}$$

$$\frac{t_2 \to t_2'}{-(t_1, t_2) \to -(t_1, t_2')} \quad \text{(E-IntSub2)}$$

$$-(v_1, v_2) \to v_1 \widetilde{-} v_2 \quad \text{(E-AppIntSub)}$$

$$\frac{t_1 \to t_1'}{*(t_1, t_2) \to *(t_1', t_2)} \quad \text{(E-IntMul1)}$$

$$\frac{t_2 \to t_2'}{*(t_1, t_2) \to *(t_1, t_2')} \quad \text{(E-IntMul2)}$$

$$*(v_1, v_2) \to v_1 \widetilde{*} v_2 \quad \text{(E-AppIntMul)}$$

$$\frac{t_1 \to t_1'}{/(t_1, t_2) \to /(t_1', t_2)} \quad \text{(E-IntDiv1)}$$

$$\frac{t_2 \to t_2'}{/(t_1, t_2) \to /(t_1, t_2')} \quad \text{(E-IntDiv2)}$$

$$/(v_1, v_2) \to v_1 \widetilde{/} v_2 \quad \text{(E-AppIntDiv)}$$

$$\frac{t_1 \to t_1'}{\wedge(t_1, t_2) \to \wedge(t_1', t_2)} \quad \text{(E-IntNand1)}$$

$$\frac{t_2 \to t_2'}{\wedge(t_1, t_2) \to \wedge(t_1, t_2')} \quad \text{(E-IntNand2)}$$

$$\wedge(v_1, v_2) \rightarrow v_1 \widetilde{\wedge} v_2 \quad \text{(E-AppIntNand)}$$

$$\frac{t_1 \rightarrow t_1'}{= (t_1, t_2) \rightarrow = (t_1', t_2)} \quad \text{(E-IntEq1)}$$

$$\frac{t_2 \rightarrow t_2'}{= (t_1, t_2) \rightarrow = (t_1, t_2')} \quad \text{(E-IntEq2)}$$

$$= (v_1, v_2) \rightarrow v_1 \widetilde{\equiv} v_2 \quad \text{(E-AppIntEq)}$$

$$\frac{t_1 \rightarrow t_1'}{< (t_1, t_2) \rightarrow < (t_1', t_2)} \quad \text{(E-IntLt1)}$$

$$\frac{t_2 \rightarrow t_2'}{< (t_1, t_2) \rightarrow < (t_1, t_2')} \quad \text{(E-IntLt2)}$$

$$< (v_1, v_2) \rightarrow v_1 \widetilde{<} v_2 \quad \text{(E-AppIntLt)}$$

where

$\widetilde{+}$	is the funtion that adds the two arguments and returns an Integer result
$\widetilde{-}$	is the function that subtracts the two arguments and returns an Integer result
$\widetilde{*}$	is the function that times the two arguments and returns an Integer result
$\widetilde{/}$	is the function that divides the two arguments and returns an Integer result
$\widetilde{\wedge}$	is the function that gets the nand result of the two arguments and returns it
$\widetilde{\equiv}$	is the function that judges whether the two values are equal. If so, returns **true**, otherwise **false**
$\widetilde{<}$	is the function that judges whether the first value is less than the second one. If so, returns **true**, otherwise **false**

9 Natural Semantics

Formally state the rules that give the natural semantics (big-step operational semantics) of the core lambda language. (Note: here we mean the version of natural semantics that operates on terms and performs substitutions, rather than the version with environments.)

The formal rules of the natural semantics for this programming language is as follows:

$$a \Downarrow v \quad (\text{B-ClosedForm})$$

for closed form a, and a should have no free variable inside.

$$v \Downarrow v \quad (\text{B-Value})$$

$$\frac{a \Downarrow \lambda x.a' \quad b \Downarrow v' \quad [x \mapsto v']a' \Downarrow v}{a\ b \Downarrow v} \quad (\text{B-App})$$

$$\frac{t_1 \Downarrow \text{ true} \quad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_2} \quad (\text{B-IfTrue})$$

$$\frac{t_1 \Downarrow \text{ false} \quad t_3 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_3} \quad (\text{B-IfFalse})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{+}(v_1, v_2)}{+(t_1, t_2) \Downarrow v} \quad (\text{B-IntAdd})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{-}(v_1, v_2)}{-(t_1, t_2) \Downarrow v} \quad (\text{B-IntSub})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{*}(v_1, v_2)}{*(t_1, t_2) \Downarrow v} \quad (\text{B-IntMul})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{/}(v_1, v_2)}{/(t_1, t_2) \Downarrow v} \quad (\text{B-IntDiv})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{\wedge}(v_1, v_2)}{\wedge(t_1, t_2) \Downarrow v} \quad (\text{B-IntNand})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{=}(v_1, v_2)}{= (t_1, t_2) \Downarrow v} \quad (\text{B-IntEq})$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad v = \widetilde{<}(v_1, v_2)}{< (t_1, t_2) \Downarrow v} \quad (\text{B-IntLt})$$

10 Natural Semantics

Express the natural semantics in Haskell code, as an interpreter for lambda terms given by the Haskell function *eval::Term* → *Term* in a module *NaturalSemantics*. The completed module source code is as follows:

```
module NaturalSemantics where

import List
import qualified AbstractSyntax as S
import qualified IntegerArithmetic as I

eval :: S.Term → S.Term

eval (S.If t1 t2 t3) =
  case eval t1 of
    S.Tru → eval t2      -- B-IfTrue
    S.Fls → eval t3      -- B-IfFalse
    _ → S.If t1 t2 t3

    -- B-App
eval (S.App t1 t2) =
  if (S.isValue $ eval t1)
    then case eval t1 of
           S.Abs x τ t11 → if ((S.isValue $ eval t2) ∧ ((S.fv (S.Abs x τ t11)) ≡ []))
                             then eval (S.subst x (eval t2) t11)
                             else S.App t1 t2
           _ → S.App t1 t2
    else S.App t1 t2

    -- B-IntAdd
eval (S.IntAdd t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → S.IntConst (I.intAdd v1 v2)
                      _ → S.IntAdd t1 t2
    _ → S.IntAdd t1 t2


    -- B-IntSub
eval (S.IntSub t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → S.IntConst (I.intSub v1 v2)
                      _ → S.IntSub t1 t2
    _ → S.IntSub t1 t2
```

```
    -- B-IntMul
eval (S.IntMul t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → S.IntConst (I.intMul v1 v2)
                      _ → S.IntSub t1 t2
    _ → S.IntSub t1 t2

    -- B-IntDiv
eval (S.IntDiv t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → S.IntConst (I.intDiv v1 v2)
                      _ → S.IntDiv t1 t2
    _ → S.IntDiv t1 t2

    -- B-IntNand
eval (S.IntNand t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → S.IntConst (I.intNand v1 v2)
                      _ → S.IntNand t1 t2
    _ → S.IntNand t1 t2

    -- B-IntEq
eval (S.IntEq t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → case I.intEq v1 v2 of
                                        True → S.Tru
                                        False → S.Fls
                      _ → S.IntEq t1 t2
    _ → S.IntEq t1 t2

    -- B-IntLt
eval (S.IntLt t1 t2) =
  case eval t1 of
    S.IntConst v1 → case eval t2 of
                      S.IntConst v2 → case I.intLt v1 v2 of
                                        True → S.Tru
                                        False → S.Fls
                      _ → S.IntLt t1 t2
    _ → S.IntLt t1 t2
```

-- B-Value and Exceptions
$eval\ t = t$

11 Test Cases

11.1 Test 1

```
app(abs(x:Int.+(x, 3)),4)
```

```
----Term----
app(abs(x:Int.+(x,3)),4)
----Type----
Int
----Normal Form of Small-Step Style----
7
----Normal Form of Big-Step Style----
7
```

11.2 Test 2

```
if +(0, 0) then 8 else 9 fi
```

```
----Term----
if +(0,0) then 8 else 9 fi
----Type----
Main: type error
```

11.3 Test 3

```
if abs(x:Int.x) then 8 else 9 fi
```

```
----Term----
if abs(x:Int.x) then 8 else 9 fi
----Type----
Main: type error
```

11.4 Test 4

```
app(abs(x:Int.app(abs(z:Int.+(z,x)) ,5)), app(abs(x:Int.-(x, 2)), 4))
```

```
----Term----
app(abs(x:Int.app(abs(z:Int.+(z,x)),5)),app(abs(x:Int.-(x,2)),4))
----Type----
Int
----Normal Form of Small-Step Style----
```

```
7
----Normal Form of Big-Step Style----
7
```

11.5 Test 5

```
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
----Normal Form of Small-Step Style----
false
----Normal Form of Big-Step Style----
false
```

11.6 Test 6

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

```
----Term----
app(abs(x:Int.1234),10)
----Type----
Int
----Normal Form of Small-Step Style----
1234
----Normal Form of Big-Step Style----
1234
```

11.7 Test 7

```
 if true then true else false fi
```

```
----Term----
if true then true else false fi
----Type----
Bool
----Normal Form of Small-Step Style----
true
----Normal Form of Big-Step Style----
true
```

11.8 Test 8

```
if =(0,0) then 8 else 9 fi
```

```
----Term----
if =(0,0) then 8 else 9 fi
----Type----
Int
----Normal Form of Small-Step Style----
8
----Normal Form of Big-Step Style----
8
```

11.9 Test 9

```
 /(4294967295,76)
```

```
----Term----
/(4294967295,76)
----Type----
Int
----Normal Form of Small-Step Style----
56512727
----Normal Form of Big-Step Style----
56512727
```

11.10 Test 10

```
app(abs(x:Int.app(abs(z:Int.*(z,x)) ,5)), app(abs(x:Int.^(x, 2)), 4))
```

```
----Term----
app(abs(x:Int.app(abs(z:Int.*(z,x)),5)),app(abs(x:Int.^(x,2)),4))
----Type----
Int
----Normal Form of Small-Step Style----
4294967291
----Normal Form of Big-Step Style----
4294967291
```