Homework 04 : CS 558

Taylor Berger

November 13, 2015

1 Fixed Pont Computations

The addition of a new fixed point computation requires an additional term in the Lambda Calculus language definitions. We add the data constructor *Fix* to represent a fixed point combinator for the simply typed lambda calculus language extended with booleans and naturals (TLBN).

```
type VarName = String
data Term = Fix Term |
  Identifier VarName |
  Abstraction VarName Term |
  Application Term Term |
  If Term Term Term |
  Succ Term |
  Pred Term |
  IsZero Term |
  Tru |
  Fls |
  Zero deriving Eq
data Type = Function Type Type |
  Boole |
  Nat |
  NullType deriving Eq
instance Show Type where
  show Boole = "Boolean"
  show Nat = "Nat"
  show (Function t1 t2) = "(" ⧺ show t1 ⧺ " -> " ⧺ show t2 ⧺ ")"
  show NullType = "<NULL>"
type TypeContext = M.Map VarName Type
```

Additionally, the show function is improved for showing numerical terms of the TLBN. Other than that, there is no difference from the last report on the functionality of the TLBN. The full code can be found in the appendix at the end.

```
instance Show Term where
  show (Identifier name) = "Identifier " ⧺ name
    show (Abstraction name t) = "Abstraction " ⧺ name ⧺ " ("
       ⧺ (show t) ⧺ ")"
    show (Application t1 t2) = "Application (" ⧺ (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 (Succ t) = if isNumeric t
      then show $ convert (Succ t)
      else "Succ " ++ show t
    show (Pred t) = "Pred (" ++ show t ++ ")"
    show (IsZero t) = "IsZero (" ++ show t ++ ")"
    show Tru = "True"
    show Fls = "False"
    show Zero = "0"
```

The Fix term has to be evaluated according to its one-step evaluation rule. So, we pattern match out the Fix term and evaluate using the *evalFix* function below.

```
eval1 :: Term → Maybe Term
eval1 t
    | isValue t = Nothing   -- values do not require evaluation
    | otherwise = case t of
      Fix t → evalFix t
      Application t1 t2 → evalApplication t1 t2
      If t1 t2 t3 → evalIf t1 t2 t3
      IsZero t → evalIsZero t
      Succ t → evalSucc t
      Pred t → evalPred t
      otherwise → Nothing
```

EvalFix has two cases to consider. The first is where Fix is applied directly to an abstraction. If this is the case, we use the beta reduction function to replace any identifier for the abstraction with the fixed point calculation of the abstraction. The second case is where we need to evaluate a non-abstraction before we can evaluate fix. If there is no rule to evaluate the non-abstraction, we return Nothing as there is no rule to help us evaluate that construct.

```
evalFix :: Term → Maybe Term
evalFix a@(Abstraction varname t) = Just $ betaReduc varname (Fix a) t
evalFix t = eval1 t >>= return ∘ Fix
```

The repeated application of eval1 doesn't change.

```
eval :: Term → Term
eval t = case eval1 t of
  Just t1 → eval t1
  Nothing → t
```

2 TLBN Parser

Parsing the new Fix term with the Parsec Monad is pretty simple. First look for the keyword "fix" and an open parenthesis. Then, parse a term and check to make sure the type of the term fix is applied to is also a function type. The parser sets the return type of Fix to the the function parsed and then parses the trailing parenthesis.

```
fix :: Monad m ⇒ ParsecT String TypeContext m Term
fix = try $ do
  keyword "fix"
  keyword "("
  t ← term
  t_type ← getReturnState
  case t_type of
    (Function a b) → do
      keyword ")"
      modifyState $ merge returnType a
      return $ Fix t
    otherwise → fail $ "Fail, expected a function type for ’Fix’ but found " ++
      (show t_type)
```

To parse basic terms, we need to update this parser to include the possiblity of parsing a Fix term.

```
  term :: Monad m ⇒ ParsecT String TypeContext m Term
  term =
    fix <|>
    identifier_term <|>
    abstraction <|>
    application <|>
    tru <|>
    fls <|>
    if_statement <|>
    zero <|>
    succ <|>
    pred <|>
    iszero <|>
    (try (keyword "(" ≫ term ≫= λk → keyword ")" ≫ return k))
     <?> "Basic term parsing"
```

3 Example Programs

Since compilation and evaluation at a high level doesn’t change, the code remains unchanged. We can write programs to use the new features granted to us by Fix.

3.1 Example: `IsEven 7`

Given by the homework writup:

```
app (
  fix (abs (ie:arr(Nat, Bool).
    abs(x:Nat.
      if iszero(x)
      then true
      else if iszero (pred (x))
           then false
           else app (ie, pred (pred (x)))
           fi
      fi
    )
  )),
  succ (succ (succ (succ (succ (succ (succ (0)))))))
)
```

3.1.1 Evaluation

```
[taylor@localhost homework5]$ ./TLBN iseven.TLBN
Syntax Correct.
        Result type: Boolean
Evaluating...
        Result: False
```

3.2 Example: `leq 2 3`

```
app(
  app(
    fix (
      abs (leq:arr(Nat, arr(Nat, Bool)) .
        abs (x:Nat .
          abs (y:Nat .
            if iszero(x)
            then true
            else if iszero(y)
                 then false
                 else app (app (leq, pred(x)), pred(y))
                 fi
            fi
          )
        )
      )
    ),
    succ (succ (0))
  ),
  succ(succ(succ(0)))
)
```

3.2.1 Evaluation

```
Syntax Correct.
        Result type: Boolean
Evaluating...
        Result: True
```

3.3 Example: `equal 2 3`

```
app (
  app (
    fix (abs (equal: arr(Nat, arr(Nat, Bool)).
      abs(x:Nat .
        abs(y:Nat .
          if iszero(x)
          then
            if iszero(y)
            then true
            else false
            fi
          else
            if iszero(y)
            then false
            else app (app (equal, pred(x)), pred (y))
            fi
          fi
        )
      )
    )),
    succ (succ (0))
  ),
  succ (succ (succ (0)))
)
```

3.3.1 Evaluation

```
Syntax Correct.
        Result type: Boolean
Evaluating...
        Result: False
```

3.4 Example: plus 2 3

```
app (
  app (
    fix(abs( plus : arr (Nat, arr(Nat, Nat)) .
          abs (x : Nat .
            abs (y : Nat .
              if iszero(x)
              then
                y
              else
                succ (app (app (plus, pred (x)), y))
              fi
            )
          )
    )),
    succ (succ (0))
  ),
  succ (succ (succ (0)))
)
```

3.4.1 Evaluation

```
Syntax Correct.
        Result type: Nat
Evaluating...
        Result: 5
```

3.5 Example: times 2 3

```
app (
  app (
    fix( abs( times : arr (Nat, arr(Nat, Nat)) .
      abs (z : Nat .
        abs (w : Nat .
          if iszero(pred(z))
          then w
```

```
     else
      app(
       app(
        fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
         abs (x : Nat .
          abs (y : Nat .
           if iszero(x)
           then y
           else succ (app (app (plus, pred (x)), y))
           fi
          )
         )
        )),
        w
       ),
       app( app(times, pred(z)), w)
      )
     fi
    )
   )
  )),
  succ (succ (0))
 ),
 succ (succ (succ (0)))
)
```

3.5.1 Evaluation

```
Syntax Correct.
    Result type: Nat
Evaluating...
    Result: 5
```

3.6 Example: `exp 2 3`

```
app (
 app (
  fix (abs (exp : arr (Nat, arr(Nat, Nat)) .
   abs(a : Nat .
    abs(b : Nat .
     if iszero(b)
     then
      succ(0)
     else
      app(
       app(
        fix( abs( times : arr (Nat, arr(Nat, Nat)) .
         abs (z : Nat .
          abs (w : Nat .
           if iszero(pred(z))
           then w
           else
            app(
             app(
              fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
               abs (x : Nat .
                abs (y : Nat .
                 if iszero(x)
                 then y
                 else succ (app (app (plus, pred (x)), y))
                 fi
                )
               )
              )),
              w
             ),
             app( app(times, pred(z)), w)
            )
           fi
          )
         )
        )),
        a
       ),
       app (app (exp, a), pred(b))
      )
     fi
    )
   )
  )),
  succ (succ (0))
 ),
 succ (succ (succ (0)))
)
```

3.6.1 Evaluation

```
Syntax Correct.
    Result type: Nat
Evaluating...
    Result: 8
```

3.7 Example: `exp 3 2`

```
app (
 app (
  fix (abs (exp : arr (Nat, arr(Nat, Nat)) .
   abs(a : Nat .
    abs(b : Nat .
     if iszero(b)
     then
      succ(0)
     else
      app(
       app(
        fix( abs( times : arr (Nat, arr(Nat, Nat)) .
         abs (z : Nat .
          abs (w : Nat .
                      if iszero(pred(z))
                      then w
                      else
                        app(
                          app(
                            fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
                              abs (x : Nat .
                                abs (y : Nat .
                                  if iszero(x)
                                  then y
                                  else succ (app (app (plus, pred (x)), y))
                                  fi
                                )
                              )
                            )),
                            w
                          ),
                          app( app(times, pred(z)), w)
                        )
                      fi
                    )
                  )
                )),
                a
              ),
              app (app (exp, a), pred(b))
            )
          fi
        )
      )
    )),
    succ (succ (0))
  ),
  succ (succ (succ (0)))
)
```

3.7.1 Evaluation

```
Syntax Correct.
        Result type: Nat
Evaluating...
        Result: 9
```

3.8 Example: `fact 3`

```
app (
  fix (abs (fact : arr (Nat, Nat) .
    abs(a : Nat .
      if iszero(a)
      then
        succ(0)
      else
        app(
          app(
            fix( abs( times : arr (Nat, arr(Nat, Nat)) .
              abs (z : Nat .
                abs (w : Nat .
                  if iszero(pred(z))
                  then w
                  else
                    app(
                      app(
                        fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
                          abs (x : Nat .
                            abs (y : Nat .
                              if iszero(x)
                              then y
                              else succ (app (app (plus, pred (x)), y))
                              fi
                            )
                          )
                        )),
                        w
                      ),
                      app( app(times, pred(z)), w)
                    )
                  fi
                )
              )
            )),
            a
          ),
          app (fact, pred (a))
        )
      fi
    )
  )),
  succ (succ (succ (0)))
)
```

3.8.1 Evaluation

```
Syntax Correct.
        Result type: Nat
Evaluating...
        Result: 6
```

3.9 Example: fact 5

```
app (
  fix (abs (fact : arr (Nat, Nat) .
    abs(a : Nat .
      if iszero(a)
      then
        succ(0)
      else
        app(
          app(
            fix( abs( times : arr (Nat, arr(Nat, Nat)) .
              abs (z : Nat .
                abs (w : Nat .
                  if iszero(pred(z))
                  then w
                  else
                    app(
                      app(
                        fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
                          abs (x : Nat .
                            abs (y : Nat .
                              if iszero(x)
                              then y
                              else succ (app (app (plus, pred (x)), y))
                              fi
                            )
                          )
                        )),
                        w
                      ),
                      app( app(times, pred(z)), w)
                    )
                  fi
                )
              )
            )),
            a
          ),
          app (fact, pred (a))
        )
      fi
    )
  )),
  succ ( succ( succ (succ (succ (0)))))
)
```

3.9.1 Evaluation

```
Syntax Correct.
        Result type: Nat
Evaluating...
        Result: 120
```

3.10 Example: fact (fact 3)

```
app (
  fix (abs (fact : arr (Nat, Nat) .
    abs(a : Nat .
      if iszero(a)
      then
        succ(0)
      else
        app(
          app(
            fix( abs( times : arr (Nat, arr(Nat, Nat)) .
              abs (z : Nat .
                abs (w : Nat .
                  if iszero(pred(z))
                  then w
                  else
                    app(
                      app(
                        fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
                          abs (x : Nat .
                            abs (y : Nat .
                              if iszero(x)
                              then y
                              else succ (app (app (plus, pred (x)), y))
                              fi
                            )
                          )
                        )),
                        w
                      ),
                      app( app(times, pred(z)), w)
                    )
                  fi
                )
              )
            )),
            a
          ),
          app (fact, pred (a))
        )
      fi
    )
  )),
  app (
    fix (abs (fact : arr (Nat, Nat) .
      abs(a : Nat .
        if iszero(a)
        then
          succ(0)
        else
          app(
            app(
              fix( abs( times : arr (Nat, arr(Nat, Nat)) .
                abs (z : Nat .
                  abs (w : Nat .
                    if iszero(pred(z))
                    then w
                    else
                      app(
                        app(
                          fix( abs( plus : arr (Nat, arr(Nat, Nat)) .
                            abs (x : Nat .
                              abs (y : Nat .
                                if iszero(x)
```

```
                                then y
                                else succ (app (app (plus, pred (x)), y))
                                fi
                              )
                            )
                          )),
                          w
                        ),
                        app( app(times, pred(z)), w)
                      )
                    fi
                  )
                )
              )),
              a
            ),
            app (fact, pred (a))
          )
        fi
      )
    )),
    succ( succ( succ(0)))
  )
)
```

3.10.1 Evaluation

```
Syntax Correct.
        Result type: Nat
Evaluating...
        Result: 720
```

Appendix

3.11 LcData.hs

```
module LcData where
import Data.Map as M
type VarName = String
data Term = Identifier VarName |
  Abstraction VarName Term |
  Application Term Term |
  If Term Term Term |
  Fix Term |
  Succ Term |
  Pred Term |
  IsZero Term |
  Tru |
  Fls |
  Zero deriving Eq
data Type = Function Type Type |
  Boole |
  Nat |
  NullType deriving Eq
instance Show Term where
  show (Identifier name) = "Identifier " ++ name
  show (Abstraction name t) = "Abstraction " ++ name ++ " (" ++ (show t) ++ ")"
  show (Application t1 t2) = "Application (" ++ (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 (Succ t) = if isNumeric t
    then show $ convert (Succ t)
    else "Succ " ++ show t
  show (Pred t) = "Pred (" ++ show t ++ ")"
  show (IsZero t) = "IsZero (" ++ show t ++ ")"
  show Tru = "True"
  show Fls = "False"
  show Zero = "0"
instance Show Type where
  show Boole = "Boolean"
  show Nat = "Nat"
  show (Function t1 t2) = "(" ++ show t1 ++ " -> " ++ show t2 ++ ")"
  show NullType = "<NULL>"
type TypeContext = M.Map VarName Type
isNumeric :: Term → Bool
isNumeric Zero = True
isNumeric (Succ t) = isNumeric t
isNumeric _ = False
isValue :: Term → Bool
isValue Tru = True
isValue Fls = True
isValue (Identifier _) = True
isValue (Abstraction _ _) = True
isValue t = isNumeric t
convert :: Term → Int
convert (Succ t) = 1 + convert t
convert Zero = 0
```

3.12 LcEvaluator.hs

```
module LcEvaluator where
import LcData
eval :: Term → Term
eval t = case eval1 t of
  Just t1 → eval t1
  Nothing → t
eval1 :: Term → Maybe Term
eval1 t
   | isValue t = Nothing
   | otherwise = case t of
    Fix t → evalFix t
    Application t1 t2 → evalApplication t1 t2
    If t1 t2 t3 → evalIf t1 t2 t3
    IsZero t → evalIsZero t
    Succ t → evalSucc t
    Pred t → evalPred t
    otherwise → Nothing
evalFix :: Term → Maybe Term
evalFix a@(Abstraction varname t) = Just $ betaReduc varname (Fix a) t
evalFix t = eval1 t ≫= return ∘ Fix

betaReduc :: VarName → Term → Term → Term
betaReduc l r (Identifier name) = if name ≡ l
  then r
  else (Identifier name)
betaReduc l r (Abstraction name term) = Abstraction name $ betaReduc l r term
betaReduc l r (Application t1 t2) = Application (betaReduc l r t1) (betaReduc l r t2)
betaReduc l r (If t1 t2 t3) = If (betaReduc l r t1) (betaReduc l r t2) (betaReduc l r t3)
betaReduc l r (Succ t) = Succ (betaReduc l r t)
betaReduc l r (Pred t) = Pred (betaReduc l r t)
betaReduc l r (IsZero t) = IsZero (betaReduc l r t)
betaReduc l r t = t

evalApplication :: Term → Term → Maybe Term
evalApplication t1@(Abstraction name t) t2
   | isValue t2 = Just (betaReduc name t2 t)
   | otherwise = eval1 t2 ≫= return ∘ (Application t1)
evalApplication t1 t2
   | isValue t1 = eval1 t2 ≫= return ∘ (Application t1)
   | otherwise = eval1 t1 ≫= return ∘ λt → (Application t t2)

evalIf :: Term → Term → Term → Maybe Term
evalIf Tru t2 t3 = Just t2
evalIf Fls t2 t3 = Just t3
evalIf t1 t2 t3 = eval1 t1 ≫= return ∘ λt → (If t t2 t3)

evalSucc :: Term → Maybe Term
evalSucc t = eval1 t ≫= Just ∘ Succ

evalPred :: Term → Maybe Term
evalPred (Succ t) = Just t
evalPred Zero = Just Zero
evalPred t = eval1 t ≫= Just ∘ Pred

evalIsZero :: Term → Maybe Term
evalIsZero Zero = Just Tru
evalIsZero (Succ t)
   | isNumeric t = Just Fls
   | otherwise = Nothing
evalIsZero t = eval1 t ≫= Just ∘ IsZero
```

3.13 LcParser.hs

```
module LcParser where

import Prelude hiding (succ, pred)

import Control.Monad.Trans (liftIO)

import Text.Parsec
import Text.Parsec.Char
import Text.ParserCombinators.Parsec.Char
import qualified Data.Map.Strict as M
import LcData
returnType = "_"
whitespace :: Monad m ⇒ ParsecT String TypeContext m ()
whitespace = spaces ≫ return ()
keyword :: Monad m ⇒ String → ParsecT String TypeContext m ()
keyword p = try $ do
  whitespace
  string p <? > ("Expecting keyword: " ⧺ p)
  whitespace
merge :: VarName → Type → TypeContext → TypeContext
merge name t context = M.insert name t context
getReturnState :: Monad m ⇒ ParsecT String TypeContext m Type
getReturnState = do
  gamma ← getState
  return $ gammaM.! returnType
tru :: Monad m ⇒ ParsecT String TypeContext m Term
tru = try $ do
  keyword "true"
  modifyState $ merge returnType Boole
  return Tru
fls :: Monad m ⇒ ParsecT String TypeContext m Term
fls = try $ do
  keyword "false"
  modifyState $ merge returnType Boole
  return Fls
zero :: Monad m ⇒ ParsecT String TypeContext m Term
zero = try $ do
  keyword "0"
  modifyState $ merge returnType Nat
  return Zero
iszero :: Monad m ⇒ ParsecT String TypeContext m Term
iszero = try $ do
  keyword "iszero"
  keyword "("
  t ← term <? > "Error parsing: Succ ( Term ), expected Term but failed"
  t_type ← getReturnState
  if t_type ≡ Nat
  then do
    keyword ")"
      -- change the state from Nat to Boole
    modifyState $ merge returnType Boole
    return (IsZero t)
  else
    fail $ "Expected type 'Nat' in iszero but was " ⧺ show t_type
succ :: Monad m ⇒ ParsecT String TypeContext m Term
succ = try $ do
  keyword "succ"
  keyword "("
  t ← term <? > "Error parsing: Succ ( Term ), expected Term but failed"
  t_type ← getReturnState
  if t_type ≡ Nat
  then do
      -- no need to change the state, it is the same
    keyword ")"
    return (Succ t)
  else
    fail $ "Expected type 'Nat' in 'Succ' but was " ++ show t_type
pred :: Monad m ⇒ ParsecT String TypeContext m Term
pred = try $ do
  keyword "pred"
  keyword "("
  t ← term <?> "Error parsing: Pred ( Term ), expected Term but failed"
  t_type ← getReturnState
  if t_type ≡ Nat
  then do
      -- no need to change the state, it is the same
    keyword ")"
    return (Pred t)
  else
    fail $ "Expected type 'Nat' in 'Pred' but was " ++ show t_type
if_statement :: Monad m ⇒ ParsecT String TypeContext m Term
if_statement = try $ do
  keyword "if"
  cond ← term <?> "Expecting 'term' following _if_"
  cond_type ← getReturnState
  if cond_type ≢ Boole
  then
    fail $ "Expecting Boolean type for if-statement conditional, received: " ++
      show cond_type
  else do
    keyword "then"
    t_then ← term <?> "Expecting 'term' following _then_"
    then_type ← getReturnState
    keyword "else"
    t_else ← term <?> "Expecting 'term' following _else_"
    else_type ← getReturnState
    keyword "fi"
    if then_type ≡ else_type
    then do
      modifyState $ merge returnType then_type
      return (If cond t_then t_else)
    else
      fail $ "Type inconsistency for then/else parts of if statement\n" ++
        "then type: " ++ (show then_type) ++ "\n" ++
        "else type: " ++ (show else_type) ++ "\n"
application :: Monad m ⇒ ParsecT String TypeContext m Term
application = try $ do
  keyword "app"
  keyword "("
  t1 ← term <?> "Error parsing first 'term' following _app_"
  t1_type ← getReturnState
  keyword ","
  t2 ← term <?> "Error parsing second 'term' following \"_app_ Term\""
  t2_type ← getReturnState
  keyword ")"
  case t1_type of
    (Function t11 t12) → if t11 ≡ t2_type
      then do
        modifyState $ merge returnType t12
        return (Application t1 t2)
      else fail $ "Mismatch types for function application\n"
        ++ "function argument required type: "
        ++ show t11 ++ "\n"
        ++ "actual argument type : "
        ++ show t2_type
    otherwise → fail $ "Expecting Function type for the first term" ++
      "of an application, receieved: " ++ show t1_type
abstraction :: Monad m ⇒ ParsecT String TypeContext m Term
abstraction = try $ do
  keyword "abs"
  keyword "("
  iden ← identifier
  keyword ":"
  iden_type ← identifierType
  keyword "."
  modifyState $ merge iden iden_type
  t ← term
  t_type ← getReturnState
  keyword ")"
  modifyState $ merge returnType (Function iden_type t_type)
  modifyState $ M.delete iden
  return $ Abstraction iden t
fix :: Monad m ⇒ ParsecT String TypeContext m Term
fix = try $ do
  keyword "fix"
  keyword "("
  t ← term
  t_type ← getReturnState
  case t_type of
    (Function a b) → do
      keyword ")"
      modifyState $ merge returnType a
      return $ Fix t
    otherwise → fail $ "Fail, expected a function type for 'Fix' but found " ++
      (show t_type)
identifier :: Monad m ⇒ ParsecT String TypeContext m String
identifier = try $ do
  whitespace
  x ← many letter
  case all (x ≢) ["succ", "pred", "if", "fi", "arr", "Bool", "Nat",
    "abs", "app", "true", "false", "then", "else",
    "iszero", "fix", ""]
    of
    True → do
      return x
    otherwise → fail $ "Could not parse an identifier, must not be a reserved" ⧺
      " word or contain anything but characters: " ⧺ x
identifier_term :: Monad m ⇒ ParsecT String TypeContext m Term
identifier_term = try $ do
  x ← identifier
  context ← getState
  case M.lookup x context of
    Just t → modifyState $ merge returnType t
    Nothing → fail $ "Identifier: " ⧺ x ⧺ " has no type in current typing context"
  return $ Identifier x
term :: Monad m ⇒ ParsecT String TypeContext m Term
term =
  identifier_term < | >
  fix < | >
  abstraction < | >
  application < | >
  tru < | >
  fls < | >
  if_statement < | >
  zero < | >
  succ < | >
  pred < | >
  iszero < | >
    -- a term in parens
  (try (keyword "(" ≫ term ≫= λk → keyword ")" ≫ return k))
   <? > "Basic term parsing"
start :: Monad m ⇒ ParsecT String TypeContext m (Term, Type)
start = do
  t ← term
  term_type ← getReturnState
  return (t, term_type)
  -- typing information and ————————————————————
identifierType :: Monad m ⇒ ParsecT String TypeContext m Type
identifierType = boolType < | > natType < | > functionType
   <? > "identifier type parser"
boolType :: Monad m ⇒ ParsecT String TypeContext m Type
boolType = try $ keyword "Bool" ≫ return Boole
natType :: Monad m ⇒ ParsecT String TypeContext m Type
natType = try $ keyword "Nat" ≫ return Nat
functionType :: Monad m ⇒ ParsecT String TypeContext m Type
functionType = try $ do
  keyword "arr"
  keyword "("
  t1 ← identifierType
  keyword ","
  t2 ← identifierType
  keyword ")"
  return $ Function t1 t2
```

3.14 Main.hs

```
module Main where
import System.Environment (getArgs)
import System.IO (openFile, hGetContents, IOMode (ReadMode))
import Data.Map (singleton)
import Data.Either (Either (Left, Right))
import Text.Parsec (runParser)
import LcParser
import LcEvaluator
import LcData
main :: IO ()
main = do
  args ← getArgs
  case length args ≢ 1 of
    True → putStrLn help
    otherwise → parseLC args
parseLC :: [String] → IO ()
parseLC (filename: _) = do
  contents ← hGetContents =≪ openFile filename ReadMode
  case runParser start (singleton returnType NullType) filename contents of
    Left err → print err
    Right (term, term_type) → do
      putStrLn $ "Syntax Correct. \n\tResult type: " ⧺ show (term_type)
      putStrLn $ "Evaluating...\n\tResult: " ⧺ show (eval term)
help :: String
help = "Program requires only 1 argument. Usage: \n" ⧺
  "  TLBN <filename>"
```