Homework 4 — assigned 8 November — due Sunday 21 November

4.1 An arithmetic expression evaluator: writing recursive functions over algebraic datatypes (30pts)

We use the following data type declaration to introduce a language of simple arithmetic expressions, with variables and binding:

```
type Identifier = String
data Expr = Num Integer
          | Var Identifier
          | Let {var :: Identifier, value :: Expr, body :: Expr}
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
type Env = Identifier -> Integer
emptyEnv :: Env
emptyEnv = \s -> error ("unbound: " ++ s)
extendEnv :: Env -> Identifier -> Integer -> Env
extendEnv oldEnv s n s' = if s' == s then n else oldEnv s'
```

4.1.1 Evaluation (20pts)

Write a function `evalInEnv`, with type `Env -> Expr -> Integer`, which computes the arithmetic value of an expression (which may have free variables) in a given environment (a mapping from variables to `Integer` values).

Then define:

```
eval :: Expr -> Integer
eval e = evalInEnv emptyEnv e
```

so that `eval` evaluates closed expressions.

Example usage:

```
evalInEnv emptyEnv (Let "x" (Num 3) (Add (Var "x") (Num 5)))
```

evaluates to `8`.

4.1.2 Formatting (10pts)

Declare the data type `Expr` as an instance of the type class `Show` such that `Expr` can be printed in human-friendly form. For instance,

```
show (Let "x" (Num 3) (Add (Var "x") (Num 5)))
```

evaluates to `"let x = 3 in (x + 5) end"`.

To keep things simple, in the printed concrete syntax, I am using fully bracketted arithmetic expressions, as well as fully bracketted let-expressions (with the keyword `end` as a bracket on the right). An optional task might be to minimize the number of brackets used for arithmetic expressions by an appeal to standard operator precedence.

4.2 Booleans and numbers (50pts)

This exercise concerns the language of booleans and numbers from Chapter 3 of our textbook, *Types and Programming Languages* by Benjamin Pierce. Feel free to consult the OCaml implementations provided by the author (mentioned in the footnotes at the beginning of each chapter), `http://www.cis.upenn.edu/~bcpierce/tapl/`. You may also consult the Haskell TAPL project, `http://code.google.com/p/tapl-haskell/`.

In Haskell, implement an evaluator based on the small-step evaluation relation for the language of booleans Bool and natural numbers Nat, as in the textbook.

The terms to be evaluated will be supplied in textual form in files. This textual form is fully bracketted and intended to be easily parsable.

Here are two examples:

File `example1.BN`:

```
if iszero (0) then 0 else pred (0) fi
```

File `example2.BN`:

```
if iszero (pred (false)) then
   if true then
      pred (0)
   else
      succ (false)
   fi
else
   succ (0)
fi
```

Here is the formal syntax:

```
Term --> true
       | false
       | if Term then Term else Term fi
       | 0
       | succ lpar Term rpar
       | pred lpar Term rpar
       | iszero lpar Term rpar
       | lpar Term rpar
```

Your program will take one command-line argument, the name of the source language file, such as `example1.BN`. It will read that file and place the entire contents of the file into a string. It will then lexically analyze, or *scan*, the string to produce a list of *tokens*. It will then *parse* the list of tokens to produce a term. Any of these steps may trigger an error; if this happens, your program should abort with a suitable error message. If a term was successfully created, it should then be evaluated, a trace of the evaluation, highlighting the redex on each step, should be printed on the console, and finally the resulting normal form should be printed on the console.

For the two examples given, program output should look like this:

```
$ BN example1.BN
---Trace:---
if *iszero(0)* then 0 else pred(0) fi
*if true then 0 else pred(0) fi*
---Normal form:---
0

$ BN example2.BN
---Trace:---
---Normal form:---
if iszero(pred(false)) then if true then pred(0) else succ(false) fi else succ(0) fi
```

To ease the design process, here are some Haskell declarations that you will use:

```
module Main where
import qualified System.Environment

data Term  =  Tru
           |  Fls
           |  If Term Term Term
           |  Zero
           |  Succ Term
           |  Pred Term
           |  IsZero Term
           deriving Eq

instance Show Term where
...
isValue :: Term -> Bool
...
isNumericValue :: Term -> Bool
...
eval1 :: Term -> Maybe Term  --one step
...
eval :: Term -> Term  --multi-step (simple, non-tracing evaluator: write this first)
...
data Token  =  Identifier String -- only needed because keywords will be exceptions to it
            |  TrueKeyword
            |  FalseKeyword
            |  IfKeyword
            |  ThenKeyword
            |  ElseKeyword
            |  FiKeyword
            |  ZeroKeyword
            |  SuccKeyword
            |  PredKeyword
            |  IsZeroKeyword
            |  LPar
            |  RPar
            deriving (Eq, Show)
...
scan :: String -> [Token]
scan s = map makeKeyword (f s)
  where
    f :: [Char] -> [Token]
    ...
    makeKeyword :: Token -> Token
    ...
parseTerm :: [Token] -> Maybe (Term, [Token])
parseTerm (TrueKeyword:tl) = Just (Tru, tl)
parseTerm (FalseKeyword:tl) = Just (Fls, tl)
parseTerm (IfKeyword:tl) =
...
parse :: [Token] -> Term
...
printEval1 :: Term -> Maybe (String, Term)
...
printEval :: Term -> [String]
printEval t =
  case printEval1 t of
    Just (s, t') -> s:printEval t'
    Nothing -> []
traceEval :: Term -> IO ()
traceEval t = sequence_ (map putStrLn (printEval t))
main :: IO ()
main =
    do
      args <- System.Environment.getArgs
      let [sourceFile] = args
      source <- readFile sourceFile
      let tokens = scan source
      let term = parse tokens
      putStrLn ("---Trace:---")
      traceEval term
      putStrLn ("---Normal form:---")
      putStrLn (show (eval term))
```

4.3 Lambda-calculus (100pts)

This exercise concerns the simply-typed call-by-value lambda-calculus from Chapter 9 of our textbook, *Types and Programming Languages* by Benjamin Pierce, augmented with booleans and numbers (Chapter 8). Feel free to consult the OCaml implementations provided by the author (mentioned in the footnotes at the beginning of each chapter), `http://www.cis.upenn.edu/~bcpierce/tapl/`. You may also consult the Haskell TAPL project, `http://code.google.com/p/tapl-haskell/`.

In Haskell, implement a type checker and an evaluator based on the small-step evaluation relation for the simply-typed call-by-value lambda-calculus with booleans and numbers, formally described as follows:

Syntax

t	::=		*terms:*
		x	*variable*
		$\lambda x : T.t$	*abstraction*
		$t\ t$	*application*
		true	*constant true*
		false	*constant false*
		if t then t else t	*conditional*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=		*values:*
		$\lambda x : T.t$	*abstraction value*
		true	*true value*
		false	*false value*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*
T	::=		*types:*
		$T \to T$	*type of functions*
		Bool	*type of booleans*
		Nat	*type of natural numbers*
Γ	::=		*typing contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x : T$	*term variable binding*

Evaluation

$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2} \ (\text{E-App1})$$

$$\frac{t_2 \to t_2'}{v_1\, t_2 \to v_1\, t_2'} \ (\text{E-App2})$$

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

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

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

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

$$\frac{t_1 \to t_1'}{\mathsf{succ}\ t_1 \to \mathsf{succ}\ t_1'} \ (\text{E-Succ})$$

$$\mathsf{pred}\ 0 \to 0 \ (\text{E-PredZero})$$

$$\mathsf{pred}(\mathsf{succ}\ nv_1) \to nv_1 \ (\text{E-PredSucc})$$

$$\frac{t_1 \to t_1'}{\mathsf{pred}\ t_1 \to \mathsf{pred}\ t_1'} \ (\text{E-Pred})$$

$$\mathsf{iszero}\,0 \rightarrow \mathsf{true}\ (\text{E-IszeroZero})$$

$$\mathsf{iszero}(\mathsf{succ}\,nv_1) \rightarrow \mathsf{false}\ (\text{E-IszeroSucc})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{iszero}\,t_1 \rightarrow \mathsf{iszero}\,t_1'}\ (\text{E-Iszero})$$

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\text{T-Var})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\,t_2 : T_{12}}\ (\text{T-App})$$

$$\mathsf{true} : \mathsf{Bool}\ (\text{T-True})$$

$$\mathsf{false} : \mathsf{Bool}\ (\text{T-False})$$

$$\frac{t_1 : Bool \qquad t_2 : T \qquad t_3 : T}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}\ (\text{T-If})$$

$$0 : \mathsf{Nat}\ (\text{T-Zero})$$

$$\frac{t_1 \;:\; \mathsf{Nat}}{\mathsf{succ}\, t_1 \;:\; \mathsf{Nat}} \;(\text{T-SUCC})$$

$$\frac{t_1 \;:\; \mathsf{Nat}}{\mathsf{pred}\, t_1 \;:\; \mathsf{Nat}} \;(\text{T-PRED})$$

$$\frac{t_1 \;:\; \mathsf{Nat}}{\mathsf{iszero}\, t_1 \;:\; \mathsf{Bool}} \;(\text{T-ISZERO})$$

The terms to be evaluated will be supplied in textual form in files. This textual form is fully bracketted and intended to be easily parsable. Here is the formal syntax:

```
Type --> arr lpar Type comma Type rpar
       | bool
       | nat

Term --> identifier
       | abs lpar identifier colon Type fullstop Term rpar
       | app lpar Term comma Term rpar
       | true
       | false
       | if Term then Term else Term fi
       | 0
       | succ lpar Term rpar
       | pred lpar Term rpar
       | iszero lpar Term rpar
       | lpar Term rpar
```

As in the preceding homework, the lexical structure is not given formally, but it should be clear from the examples.

Here are three simple examples:

File `example1.TLBN`:

```
app (abs (x: Nat . succ(succ(x))),
     succ(0))
```

File `example2.TLBN`:

```
if iszero (0) then succ (0) else false fi
```

File `example3.TLBN`:

```
app (abs (x:Bool.x), false)
```

Your program will take one command-line argument, the name of the source language file, such as `example1.TLBN`. It will read that file and place the entire contents of the file into a string. It will then lexically analyze, or *scan*, the string to produce a list of *tokens*. It will then *parse* the list of tokens to produce a term. Any of these steps may trigger an error; if this happens, your program should abort with a suitable error message.

If a term was successfully created, it should then be type-checked. If it has a type, the term should be evaluated and finally the resulting normal form should be printed on the console.

For the two examples given, program output should look like this:

```
$ TLBN example1.TLBN
---Term:---
app(abs(x:Nat.succ(succ(x))),succ(0))
---Type:---
Nat
---Normal form:---
succ(succ(succ(0)))

$ TLBN example2.TLBN
---Term:---
if iszero(0) then succ(0) else false fi
---Type:---
TLBN: type error

$ TLBN example3.TLBN
---Term:---
app(abs(x:Bool.x),false)
---Type:---
Bool
---Normal form:---
false
```

To ease the design process, some Haskell code that you must complete is provided in a file `TLBN-skeleton.lhs`; you may also use the LaTeX and lhs2TeX declarations as a guide. Also provided is a file `TLBN-skeleton.pdf` containing the same code formatted.

Challenge tasks

It is not necessary to complete these tasks to receive full credit on the homework assignment.

- **Evaluation contexts.** An *evaluation context* describes the position of a subterm within a term; in particular it can be used to describe the position of a redex within a term.

  The file `TLBN-skeleton-extra.hs` contains this code.

  ```
  data EvalContext  =  Hole
                    |  EvalContextAppV Term EvalContext -- where Term is a value
                    |  EvalContextAppT EvalContext Term
                    |  EvalContextIf EvalContext Term Term
                    |  EvalContextSucc EvalContext
                    |  EvalContextPred EvalContext
                    |  EvalContextIsZero EvalContext
  ```

  Write a function `decompose :: Term -> Maybe (EvalContext, Term)` to decompose a term into a redex and an evaluation context it fits into (if there is a redex).

  Use this notion to produce a trace of the evaluation, highlighting the redex at each step (the highlighting can be done in plain text using asterisks as in Homework 4, or using underlining in LaTeX , or in some other suitable manner).

  Can you think of a way to exploit this notion for the evaluation itself?

- **Typing, more explicitly.** A *typing derivation* is a tree of inference rules instances that exhibits a proof that a certain term has a certain type. An example is shown on p. 94 of TAPL. Come up with a Haskell data type `TypingDerivation` to describe a typing derivation; write a function `deriveType :: Term -> TypingDerivation` to construct a typing derivation; and write a function `drawDerivation :: TypingDerivation -> String` to produce a graphic of the tree (here it is deliberately left unspecified what the output string should be: you might produce LaTeX code, or PostScript code, or some other textual representation that can be turned into a graphic).

What will be handed out

The examples shown above as well as the skeleton Haskell implementation will be posted on the course wiki.

How to turn in

Submission instructions: use the `turnin` facility provided by the department; see course wiki for details.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.