Functional Programming Skills
Assignment 6

Arthur Nunes-Harwitt

Explicitly write the type of each function.

1. (10 points) Define a type `Const` representing constants that consists of four choices. `IConst` is for integers; `BConst` is for Booleans; `FConst1` for functions of arity one, which takes and returns a `Const`; and `FConst2` for functions of arity two, which takes two `Const` and returns a `Const`. The function choices should also have name string components. `Const` should be an instance of both `Show` and `Eq`. Function constants should be compared based only on their string component.

   Examples:

   ```
   *Assign6> IConst 5
   5
   *Assign6> BConst True
   True
   *Assign6> FConst1 "abs" primAbs
   <prim1:abs>
   *Assign6> FConst2 "+" primPlus
   <prim2:+>
   ```

2. (10 points) Define a type `Exp` representing terms in the extended λ-calculus language that consists of five choices. `Econ` is for constants; `Var` is for variables, which are represented as strings; `Lambda` is for λ-expressions (a.k.a. abstractions); `IfExp` is for if-expressions; and `Appl` is for applications, which has exactly two components. `Exp` should be an instance of both `Show` and `Eq`.

   Examples:

   ```
   *Assign6> Econ (IConst 2)
   Econ 2
   *Assign6> Var "x"
   Var "x"
   *Assign6> Lambda "x" (Var "x")
   Lambda "x" (Var "x")
   *Assign6> IfExp (Econ (BConst True)) (Var "x") (Var "y")
   IfExp (Econ True) (Var "x") (Var "y")
   *Assign6> Appl (Lambda "x" (Var "x")) (Econ (IConst 2))
   Appl (Lambda "x" (Var "x")) (Econ 2)
   ```

3. (10 points) The λ-calculus is equivalent to a "combinator" calculus requiring only two operators: S and K. The abstract syntax for combinator terms is specified via the grammar below.

$$
\begin{array}{lll}
M_c, N_c & ::= & c \\
 & ::= & x \\
 & ::= & O \\
 & ::= & (M_c\ N_c) \\
O & ::= & \mathsf{S} \\
 & ::= & \mathsf{K} \\
 & ::= & \mathsf{I} \\
 & ::= & \mathsf{B} \\
 & ::= & \mathsf{C} \\
 & ::= & \mathsf{CIf}
\end{array}
$$

Note that although variables are part of the syntax, they cannot be given values in the combinator calculus. Thus terms in the combinator calculus typically contain *no* variables. Also note that there are more than two combinator operators. These additional operators are for efficiency and readability.

Define a type `CExp` representing terms in the combinator calculus that consists of four choices. `Ccon` is for constants; `CVar` is for variables; `Cop` is for combinator operators; and `CAppl` is for applications, which has exactly two components. `CExp` should be an instance of both `Show` and `Eq`.

Examples:

```
*Assign6> Ccon (IConst 5)
Ccon 5
*Assign6> CVar "x"
CVar "x"
*Assign6> Cop S
Cop S
*Assign6> Cop CIf
Cop CIf
*Assign6> CAppl (Cop I) (CVar "x")
CAppl (Cop I) (CVar "x")
*Assign6> CAppl (CAppl (Cop S) (Cop K)) (Cop K)
CAppl (CAppl (Cop S) (Cop K)) (Cop K)
```

4. (25 points) We will now write three functions to compile, or translate, λ-terms to combinator terms. Most of the translation process involves two functions: $\mathcal{C}$ and $\mathcal{A}$. The function $\mathcal{C}$ translates terms structurally using $\mathcal{A}$ to eliminate bound variables in abstractions. The third function will replace global variables with the appropriate constant from the initial environment. NOTE: The term domain of $\mathcal{A}$ and the replacement function is the set of combinator terms.

$$
\begin{array}{lll}
\mathcal{C}[\![c]\!] & = & c \\
\mathcal{C}[\![x]\!] & = & x \\
\mathcal{C}[\![(\mathit{if}\ M_0\ M_1\ M_2)]\!] & = & (((\mathsf{CIf}\ \mathcal{C}[\![M_0]\!])\ \mathcal{C}[\![M_1]\!])\ \mathcal{C}[\![M_2]\!] \\
\mathcal{C}[\![(\lambda x.M)]\!] & = & \mathcal{A}_x(\mathcal{C}[\![M]\!]) \\
\mathcal{C}[\![(M\ N)]\!] & = & (\mathcal{C}[\![M]\!]\ \mathcal{C}[\![N]\!])
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{A}_x\llbracket c \rrbracket & = & (\mathsf{K}\ c) & \\
\mathcal{A}_x\llbracket x \rrbracket & = & \mathsf{I} & \\
\mathcal{A}_x\llbracket y \rrbracket & = & (\mathsf{K}\ y) & \\
\mathcal{A}_x\llbracket O \rrbracket & = & (\mathsf{K}\ O) & \\
\mathcal{A}_x\llbracket (M_c\ N_c) \rrbracket & = & M_c' & \text{if } \mathcal{A}_x\llbracket M_c \rrbracket = (\mathsf{K}\ M_c') \text{ and } \mathcal{A}_x\llbracket N_c \rrbracket = \mathsf{I} \\
\mathcal{A}_x\llbracket (M_c\ N_c) \rrbracket & = & (\mathsf{K}\ (M_c'\ N_c')) & \text{if } \mathcal{A}_x\llbracket M_c \rrbracket = (\mathsf{K}\ M_c') \text{ and } \mathcal{A}_x\llbracket N_c \rrbracket = (\mathsf{K}\ N_c') \\
\mathcal{A}_x\llbracket (M_c\ N_c) \rrbracket & = & ((\mathsf{B}\ M_c')\ N_c') & \text{if } \mathcal{A}_x\llbracket M_c \rrbracket = (\mathsf{K}\ M_c') \text{ and } \mathcal{A}_x\llbracket N_c \rrbracket = N_c' \text{ where } N_c' \neq \mathsf{I} \text{ and } N_c' \neq (\mathsf{K}\ N_c'') \\
\mathcal{A}_x\llbracket (M_c\ N_c) \rrbracket & = & ((\mathsf{C}\ M_c')\ N_c') & \text{if } \mathcal{A}_x\llbracket M_c \rrbracket = M_c' \neq (\mathsf{K}\ M_c'') \text{ and } \mathcal{A}_x\llbracket N_c \rrbracket = (\mathsf{K}\ N_c') \\
\mathcal{A}_x\llbracket (M_c\ N_c) \rrbracket & = & ((\mathsf{S}\ M_c')\ N_c') & \text{if } \mathcal{A}_x\llbracket M_c \rrbracket = M_c' \neq (\mathsf{K}\ M_c'') \text{ and } \mathcal{A}_x\llbracket N_c \rrbracket = N_c' \neq (\mathsf{K}\ N_c'')
\end{array}
$$

(a) Write functions `compile` and `abstract` that implement $\mathcal{C}$ and $\mathcal{A}$, respectively.

(b) Write functions `replaceGlobal` and `compileReplacing`. The function `replaceGlobal` takes a combinator term and returns a combinator term where all the variables have been replaced with the constants from the initial environment. The function `compileReplacing` is the composition of `replaceGlobal` and `compile`.

The initial environment is represented as a list of pairs.

Examples:

```
*Assign6> initEnv
[("abs",<prim1:abs>),
 ("+",<prim2:+>),
 ("-",<prim2:->),
 ("*",<prim2:*>),
 ("div",<prim2:div>),
 ("==",<prim2:==>)]
*Assign6> compileReplacing (Econ (IConst 5))
Ccon 5
*Assign6> compileReplacing (Var "x")
CVar "x"
*Assign6> compileReplacing (IfExp (Var "x") (Var "y") (Var "z"))
CAppl (CAppl (CAppl (Cop CIf) (CVar "x")) (CVar "y")) (CVar "z")
*Assign6> compileReplacing (Lambda "x" (Var "x"))
Cop I
*Assign6> compileReplacing (Var "abs")
Ccon <prim1:abs>
*Assign6> compileReplacing (Appl (Var "abs") (Econ (IConst (-2))))
CAppl (Ccon <prim1:abs>) (Ccon -2)
*Assign6> compileReplacing (Lambda "x" (Appl (Var "abs") (Var "x")))
Ccon <prim1:abs>
*Assign6> compileReplacing (Lambda "x" (Appl (Appl (Var "+") (Var "x"))
                                             (Econ (IConst 1))))
CAppl (CAppl (Cop C) (Ccon <prim2:+>)) (Ccon 1)
```

5. (10 points) Write the function `reduceComb` that implements the following reductions.

$$
\begin{array}{lcl}
(\mathsf{I}\ X) & = & X \\
((\mathsf{K}\ X)\ Y) & = & X \\
(((\mathsf{S}\ F)\ G)\ X) & = & ((F\ X)\ (G\ X)) \\
(((\mathsf{B}\ F)\ G)\ X) & = & (F\ (G\ X)) \\
(((\mathsf{C}\ F)\ X)\ Y) & = & ((F\ Y)\ X) \\
(((\mathsf{CIf\ True})\ X)\ Y) & = & X \\
(((\mathsf{CIf\ False})\ X)\ Y) & = & Y
\end{array}
$$

Also if a `FConst1` is applied to one constant, it should reduce to a constant, and if a `FConst2` is applied to two constants it should reduce to a constant. All other terms should reduce to themselves.

Examples:

```
*Assign6> reduceComb (CAppl (Cop I) (CVar "X"))
CVar "X"
*Assign6> reduceComb (CAppl (CAppl (Cop K) (CVar "X")) (CVar "Y"))
CVar "X"
*Assign6> reduceComb (CAppl (CAppl (CAppl (Cop S) (CVar "F")) (CVar "G")) (CVar "X"))
CAppl (CAppl (CVar "F") (CVar "X")) (CAppl (CVar "G") (CVar "X"))
*Assign6> reduceComb (CAppl (CAppl (CAppl (Cop B) (CVar "F")) (CVar "G")) (CVar "X"))
CAppl (CVar "F") (CAppl (CVar "G") (CVar "X"))
*Assign6> reduceComb (CAppl (CAppl (CAppl (Cop C) (CVar "F")) (CVar "X")) (CVar "Y"))
CAppl (CAppl (CVar "F") (CVar "Y")) (CVar "X")
*Assign6> reduceComb (CAppl (CAppl (CAppl (Cop CIf) (Ccon (BConst True))) (CVar "X"))
(CVar "Y"))
CVar "X"
*Assign6> reduceComb (CAppl (CAppl (CAppl (Cop CIf) (Ccon (BConst False)))
                                      (CVar "X"))
                             (CVar "Y"))
CVar "Y"
*Assign6> reduceComb (CAppl (Ccon (FConst1 "abs" primAbs)) (Ccon (IConst (-2))))
Ccon 2
*Assign6> reduceComb (CAppl (CAppl (Ccon (FConst2 "+" primPlus))
                                      (Ccon (IConst (-2))))
                             (Ccon (IConst 5)))
Ccon 3
```

6. (10 points) Write the function `run` which takes a combinator term and returns a combinator term. It should repeatedly use `reduceComb` to simplify a term. The use of the pattern matcher is encouraged.

Example:

```
*Assign6> run (CAppl (Cop I) (CAppl (Cop I) (Ccon (IConst 5))))
Ccon 5
```

7. (10 points) Write the function `compileAndRun` which takes a closed λ-term and returns the combinator term it reduces to. (This function should make use of `run` and `compileReplacing`.)

   Examples:

```
*Assign6> compileAndRun (Var "abs")
Ccon <prim1:abs>
*Assign6> compileAndRun (Appl (Var "abs") (Econ (IConst (-2))))
Ccon 2
*Assign6> compileAndRun (Appl (Lambda "x" (Var "x")) (Econ (IConst 5)))
Ccon 5
*Assign6> compileAndRun (Appl (Lambda "x" (Econ (IConst 5)))
                        (Appl (Appl (Var "div") (Econ (IConst 1)))
                              (Econ (IConst 0))))
Ccon 5
*Assign6> compileAndRun (Appl (Lambda "x" (IfExp (Appl (Appl (Var"==") (Var "x"))
                                                       (Econ (IConst 0)))
                                                 (Var "x")
                                                 (Appl (Appl (Var "+")(Var "x"))
                                                       (Econ (IConst 1)))))
                              (Econ (IConst 3)))
Ccon 4
*Assign6> compileAndRun (Appl (Lambda "x" (IfExp (Appl (Appl (Var"==") (Var "x"))
                                                       (Econ (IConst 0)))
                                                 (Var "x")
                                                 (Appl (Appl (Var "+")(Var "x"))
                                                       (Econ (IConst 1)))))
                              (Econ (IConst 0)))
Ccon 0
```

Graduate Problems/Undergraduate Extra Credit

1. (10 points) Implement recursion in the λ-term language as follows. Add `LetRec` as one of the choices for a λ-term. Add the `Y` operator as one of the choices for a combinator operator. Add a line to `compile` so that `LetRec` is translated into a combinator term involving `Y`. Add a reduction rule to `reduceComb` to give `Y` meaning.