PLEASE READ

This is a SAMPLE final exam rather than a PRACTISE exam.

- It is a final exam from 2006.
- There are some questions about things we have not covered (e.g., the “TigerScheme” interpreter).
- My purpose in distributing it is to give you an idea of the kinds of questions I ask.
- If you see something we haven’t covered in class, don’t panic.

CS 4450 Final Examination

a) This is a closed book, closed note exam.
b) You may not use a calculator or similar device.
c) Write all work on the exam itself. Additional pages are attached at the back should you require them.

1. **(10 points total, 1 each).** Indicate which of the following Haskell expressions is **valid** or **invalid** by circling the corresponding word. Consider an expression as valid if it does not cause an error when typed into the Hugs interpreter. An expression that generates a warning is still considered valid.

VALID	INVALID	`[[1,2.0],[3,4.0]]`
VALID	INVALID	`(1.0, 5, ["a","dog"], Just 1)`
VALID	INVALID	`if 1<2 then length "I gotta be me!"`
VALID	INVALID	`f x = if x then g y = y+2` `else h l = l`
VALID	INVALID	`f x = if x then x else 2`
VALID	INVALID	`f (a:int) (b:real) = a + b`
VALID	INVALID	`f (x:t) = f t` `f []    = []`
VALID	INVALID	`[1]:(head (tail [[[1,2],[3,5]],[[0],[7,8]]]))`
VALID	INVALID	`(map (\ x -> x+1)) . tail`
VALID	INVALID	`(head . tail) [[1],[2]]`

2. (12 points total, 4 points each)

For each of the following Haskell function declarations, write the type of the function it defines.

a)
```
f x = if x then x else x
```

b)
```
g a [] = [a]
g [] b = [b]
g a b  = g (tail a) (tail b)
```

c)
```
h (a,b,c) = if a<1 then c else b:(h(a-1,b,c))
```

TigerScheme Interpreter

```
data Value = I Int | R Float
           | ConsCell Value Value
           | FunVal ([Value]->Value)
           | NilVal
           | T | F
           | Closure [String] Term Env
           | RecClosure String [String] Term Env
           | MystVal Term

type Env = [(String,Value)]
initEnv  = []
extendEnv xs vs rho = (zip xs vs) ++ rho
applyEnv rho x      = case (lookup x rho) of
                            Just (MystVal e) -> eval e rho
                            Just v           -> v
                            Nothing          -> error ("Var "++x++" unbound\n")

eval :: Term -> Env -> Value
eval (Litint i) rho   = I i
eval (Var x) rho      = applyEnv rho x
eval Cons rho         = FunVal consVal
    where
       consVal [v1,v2] = ConsCell v1 v2
       consVal _       = error "cons takes two and only two arguments\n"

eval Plus rho         = FunVal saddl
eval Times rho        = FunVal smull
eval Minus rho        = FunVal ssubl

eval (App (e:es)) rho = apply (eval e rho)
                              (map (\ t -> eval t rho) es)

eval (Lambda xs e) rho = Closure xs e rho

eval Nil rho               = NilVal
eval Car rho               = FunVal (\ [x] -> car x)
eval Cdr rho               = FunVal (\ [x] -> cdr x)

eval (Letexp bs body) rho  = eval body rho'
     where rho' = extendEnv vars vals rho
           vars = map fst bs
           es   = map snd bs
           vals = map (\ e -> eval e rho) es

eval (Let? bs body) rho  = eval body rho'
     where rho' = extendEnv vars vals rho
           vars = map fst bs
           es   = map snd bs
           vals = map MystVal es

apply (FunVal f) vs                 = f vs
apply (Closure xs body env) vs      = eval body env'
             where env' = extendEnv xs vs env
apply (RecClosure r xs body env) vs = eval body env'
             where env' = extendEnv xs' vs' env
                   xs'  = r : xs
                   vs'  = (RecClosure r xs body env) : vs
```

(12 points) Say we introduce a mysterious new kind of let-binding to the TigerScheme interpreter called **`let?`**. "Mystery let" has the same syntactic form as other forms we have seen this semester; assume that the abstract syntax for **`let?`** is defined by this new clause in the **`Term`** data type: **`data Term = … | Let? [(String,Term)] Term`**

Here are two terms:

```
;; term a is:
   (let? ((s 10))
      (let? ((s 5))
          s))

;; term b is:
   (let? ((s 10)) in
       (let? ((f (lambda (d) s)))
          (let? ((g (lambda (x) (f 1)) (s 5))
             (g 7))))
```

The **`eval`** clause for **`Let?`** is defined on the previous page.

Questions:

The value calculated by the TigerScheme interpreter for term **`a`** is:

- **i.** **`(I 5)`**
- **ii.** **`(I 10)`**
- **iii.** **`error "Var s unbound"`**
- **iv.** None of the above.

The value calculated by the TigerScheme interpreter for term **`b`** is:

- **i.** **`(I 5)`**
- **ii.** **`(I 7)`**
- **iii.** **`(I 10)`**
- **iv.** **`error "Var s unbound\n"`**
- **v.** **`error "Var g unbound\n"`**
- **vi.** **`(I 1)`**
- **vii.** None of the above.

7. **(12 points total, 4 points each)** Give brief definitions showing you understand the issues involved. You may refer to parts of our interpreter or to TigerScheme if it helps.

a) What is a **closure** and what is it used for?

b) What is **referential transparency**?

c) Explain the difference between the **formal** and **actual** parameters of a procedure.

5. (20 points total) Consider the following grammar:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → *ident*

where *ident* is a represents any string over {a,…,z}.

a) (10 points) Write a derivation of the string "x * y + (u * z)" starting from E. You **must** label each transition with its number to receive credit. Each transition must be the result of applying one and only one rule; you may not take shortcuts.

b) (10 points) Write a Haskell data type corresponding to the above grammar.

6. (10 points) Recall "lexical addressing" from class. Lexical addressing replaces a variable reference by its lexical depth. The lexical depth of a variable reference is the number of "lambda" binders between the occurrence of the variable and the lambda binding it. Consider the example below on the left in which each variable is labeled by its lexical depth (e.g., the last reference to "**x**" is label by "1" ; the "**lambda (y)**" lies between that occurrence of "**x**" and the outermost lambda where "**x**" is defined. We can, in fact, eliminate variables altogether, replacing them by their lexical depths; below right is the result of doing so to the example on the left.

```
(lambda (x)
   (lambda (y)
      ((lambda (x)
             (x:0 y:1))
         x:1)))
```

```
(lambda
   (lambda
      ((lambda
             (0 1))
         1)))
```

a) (5 points) Translate the following lexically addressed term back to an equivalent term using variables.

```
(lambda ( (lambda (lambda (1 0))) (lambda 1) ) )
```

Hint: translate the smaller inner terms first.

b) (5 points) Below is an abstract syntax for the λ-calculus as shown in the upper left example. Modify it to represent lexically addressed version.

```
data Expr = Ident String
          | Lambda String Expr
          | App Expr Expr
```

7. Problem: Al Gaulle is responsible for maintaining a Scheme program written by another programmer. In the middle of the program, Al notices the application

```
((lambda (f) (lambda (x) (map f x)))
        (lambda (z) (+ x z)))
```

Al decides to optimize the program by reducing the application (using substitution) to

```
(lambda (x) (map (lambda (z) (+ x z)) x)
```

Did he optimize the program correctly? Why or why not?

8. Consider the following context-free grammar for *Exp*:

Exp `::=` *Identifier* | `(lambda` *Identifier* *Exp*`)` | `(`*Exp* *Exp*`)`

An *Identifier* can be any ascii string. For the following three strings, circle those that are <u>not</u> in the language of *Exp*:

```
(lambda)

(lambda lambda)

(lambda lambda lambda)
```