Due: October 27th, 2004

Please fill this first:

• Name: ________________________________
• Id number: __________________________
• Email: ________________________________

Introduction

1. This exam is closed book and closed notes. You do NOT need to remember the syntax of any language or built-in functions. If you do not remember syntax or built-in functions, use your own, and give a brief description. You may not use a computer to solve or check any of the problems.

2. Fill in the information above and the pledge below. You will receive 5 points for providing correct and complete information.

3. There are 10 problems on the exam, totaling 95 points plus 20 extra credit. The maximum possible score is 120 (5+95+20).

4. You have 4 hours to complete the exam. You must take the exam during a continuous four hour block plus an optional 15 minute break. Do not discuss the contents of the exam with anyone other than the instructor (taha@rice.edu) between now and the due date of the exam.

Pledge:

Points:

1. 10 points
2. 15 points
3. 10 points
4. 20 points
5. 10 points
6. 10 points
7. 10 points
8. 10 points
9. 10 points
10. 10 points
**Problem 1:** (10 points) Using a *minimal* amount of renaming of bound variables, write out lambda terms that result from the following:

**Grading:** 1pt each (7pts) and 3pts for correct renaming  
-1pt for each unnecessary renaming (max 3pts)

1. $(\lambda x.x)[x := \lambda y.y] = (\lambda x.x)$
2. $(\lambda x.x)[y := \lambda y.y] = (\lambda x.x)$
3. $(\lambda y.y)[y := \lambda x.x] = (\lambda y.y)$
4. $(\lambda x.xy)[y := \lambda y.y] = (\lambda x.\lambda y.y)$
5. $(\lambda x.xy)[x := \lambda y.y] = (\lambda x.xy)$
6. $(\lambda x.xy)[y := \lambda y.x] = (\lambda z.\lambda y.x)$
7. $(\lambda x.xy)[y := \lambda y.xy] = (\lambda z.\lambda y.xy)$
Problem 2: (15 points)

1. Give an inductive definition of the capture-avoiding substitution operation $e_1[x := e_2]$ on lambda terms. Do not assume that $e_1$ or $e_2$ are closed. **10pts**

   $i[x := e_2] = i$ **0pt**

   $x[x := e_2] = e_2$ **1.5pt**

   $y[x := e_2] = y$ if $y \neq x$ **1.5pt**

   $(\lambda x.e_3)[x := e_2] = (\lambda x'.e_3[x := x'][x := e_2])$ with $x' \notin FV(e_2)$ **3 pts** for correct substitution, **2pts** for $FV$

   $e_3 e_4[x := e_2] = (e_3[x := e_2])(e_4[x := e_2])$ **2pt**

2. Under what conditions can substitution be viewed as a function? **5pts**

   Key: Student’s answer must work toward alpha-equivalence. Functions must satisfy the condition \((f(x) = x_1 \text{ and } f(x) = x_2) \Rightarrow x_1 = x_2\)

   1. Results viewed under alpha-equivalence
   2. Ignore names of bound variables
   3. de Bruijn indicies
   4. Variables generated have deterministic nature.
**Problem 3:** (10 points) Assuming that \#0 refers to the closest binder, convert the following terms into de Bruijn notation:

1. \(\lambda x. x = \lambda. \#0 1\text{pt}\)
2. \(\lambda x. x \lambda y. y = \lambda. \#0 \lambda. \#0 1\text{pt}\)
3. \(\lambda x. x \lambda y. x = \lambda. \#0 \lambda. \#1 2\text{pt}\)
4. \((\lambda x. x) \lambda y. y = (\lambda. \#0) \lambda. \#0 2\text{pt}\)
5. \(\lambda x. x \lambda y. x \ y = \lambda. \#0 \lambda. \#1 \#0 2\text{pt}\)
6. \(\lambda x. x \lambda y. x \lambda x. y = \lambda. \#0 \lambda. \#1 \lambda. \#0 \#1 2\text{pt}\)
Problem 4: (20 points) In class we have used the factorial (fact) function to illustrate various concepts. Another typical example is the Fibonacci (fib) function:

```ocaml
let rec fib n =  
  match n with  
  | 0 -> 1  
  | 1 -> 1  
  | x -> (fib (x-1)) + (fib (x-2))
```

We can define a fixed-point (or Y) combinator as follows:

```ocaml
let rec y f = f (fun x -> y f x)
```

1. Write a function `myFib` and use it to give an alternate definition to `fib` without using `let rec`, but using the above fixed-point combinator instead. **5pt**

```ocaml
let myFib f =  
  fun n ->  
  match n with  
  | 0 -> 1  
  | 1 -> 1  
  | x -> (f (x-1) + (f (x-2))
```

2. We can implement a function that has an empty graph (that is, it is everywhere undefined) as follows:

```ocaml
exception Undefined

let fib0 = raise Undefined
```

Write out the graph for the following functions:

(a) `let fib1 = myFib fib0`  
   `{(0,1), (1,1)}` **1pt**

(b) `let fib2 = myFib fib1`  
   `{(0,1), (1,1),(2,2)}` **2pt**

(c) `let fib3 = myFib fib2`  
   `{(0,1), (1,1),(2,2),(3,3)}` **2pt**
3. Under what conditions can we define the fixed-point combinator simply as: 5pt

\[
\text{let rec } y f = f (y f)
\]

Call-by-name or call-by-need semantics

4. Under what conditions can we define the fixed-point combinator itself without using \texttt{let rec}? 5pt

1. Language is untyped.
2. Type language with recursive universal datatypes
   -1pt \textbf{if do not explicitly mention universal datatype in answer}
   #2
Problem 5: (10 points) Consider the following two functions:

```ocaml
let rec listM f l =
  match l with
  [] -> []
| x::xs -> (f x)::(listM f xs)

let rec listF f b l =
  match l with
  [] -> b
| x::xs -> (f x (listF f b xs))
```

Can we define

1. `listM` in terms of `listF`? If so, how? If not, why? 5pt
   ```ocaml
   let ListM f l = ListF (fun x y -> (f x)::y) [] l
   ```

2. `listF` in terms of `listM`? If so, how? If not, why? 5pt
   1. Types unconvertible: `listF` can return an arbitrary return type that cannot be defined in terms of `listM` which returns a `'b` list. 5/5pt
   ```ocaml
   listM : ('a -> 'b) -> 'a list -> 'b list
   listF : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
   ```
   Must be specific: list cannot be converted to arbitrary data type
   2. Counter-example: i.e. list summation 5/5pt
   3. Fluffy arguments: Map is point-wise while fold has an accumulator 2-3/5pts
Problem 6: (10 points) In class, we discussed the Church encodings of booleans, where True is encoded as $\lambda x.\lambda y.x$ and False as $\lambda x.\lambda y.y$.

1. Give an appropriate encoding for an if-statement that would work under call-by-name. 3pt

$$[[if \ e_1 \ then \ e_2 \ else \ e_3]] = [[e_1]] [[e_2]] [[e_3]]$$

2. Give an appropriate encoding for an if-statement that would work under call-by-value. 4pt

$$[[if \ e_1 \ then \ e_2 \ else \ e_3]] = ([[e_1]] \ \lambda x.[[e_2]] \ \lambda x.[[e_3]])(\lambda x.x)$$
Answer can use any value instead of $(\lambda x.x)$

3. Show how the logical operators not, and, and or can be encoded. 3pt

$$[[not \ b]] = [[b]] \ False \ True$$
$$[[and \ x \ y]] = [[x]] [[y]] \ False$$
$$[[or \ x \ y]] = [[x]] \ True \ [[y]]$$
Problem 7: (10 points) For this question, you are free to use de Bruijn or name-based representations. If you use a name-based representation, avoid unnecessary renaming.

Using the rules for substitution-based big-step semantics, write out the derivations for the evaluation of the term:

\[(\lambda a. \lambda b. a) \ (\ (\lambda c. c) \ (\lambda d. d) ))\]

1. assuming call-by-value semantics 5pt

\[\lambda c. c \hookrightarrow \lambda c. c\]
\[\lambda d. d \hookrightarrow \lambda d. d\]
\[c[c := \ (\lambda d. d d)] \hookrightarrow \lambda d. d d\]

\[\begin{align*}
\frac{(\lambda a. \lambda b. \ a) \ (\ (\lambda c. c) \ (\lambda d. d) )) \hookrightarrow \lambda b. \ (\ (\lambda c. c) \ (\lambda d. d) )) \quad \lambda b. \ a[a := \ (\lambda d. d d)] \hookrightarrow \lambda b. \lambda d. d d
\end{align*}\]

2. assuming call-by-name semantics 5pt

\[\lambda a. \lambda b. \ a \hookrightarrow \lambda a. \lambda b. \ a\]
\[\lambda b. \ a[a := \ (\lambda c. c) \ (\lambda d. d d)] \hookrightarrow \lambda b. \ (\ (\lambda c. c) \ (\lambda d. d d))\]

Two key aspects: 1. Must have derivation, 2. Must use big step semantics

0pts: no derivations
- 1-3pt big step semantics partially incorrect
Problem 8:  (10 points) Consider the following function:

```ml
let rec sum l a =  
  match l with  
    [] -> !a  
  | x::xs -> (a:=(!a + x); sum xs a)
```

What does \texttt{sum [1;2;3] (ref 0)} return when we assume:

1. call-by-value semantics \textbf{3pt}

6

2. call-by-name semantics \textbf{4pt}

0

3. call-by-need semantics \textbf{3pt}

6

If student has correct explanation, but wrong answer on any part, student can get up to half credit (1.5/3 or 2/4)
**Problem 9:** (Extra credit: 10 points)

1. Convert the following function into continuation-passing style: *4pt*

   ```ml
   let rec listM f l =
   match l with
   | [] -> []
   | x::xs -> (f x)::(listM f xs)
   ```

   ```ml
   let rec listM f l k =
   match l with
   | [] -> k []
   | x::xs -> (listM f xs (fun rest -> k (f x)::rest))
   ```

2. Replace the continuation (which is a function) in what you have written for the first part of this question by an accumulator that is a list. (Hint: You will need to use list reverse in the base case.) *3pt*

   ```ml
   let rec listM f l acc =
   match l with
   | [] -> List.rev acc
   | x::xs -> (listM f xs (f x)::acc)
   ```

3. Translate function from the second part of this question into imperative pseudo code that uses goto's and no recursion. *3pt, must be systematic, with goto*

   ```ml
   list listM (fun f, list l, list acc) {
   int x; //should be untyped

   listM_0: if (l == [])
   { return List.rev(acc); }
   else
   { x = List.hd (l);
     l = List.tl(l);
     acc = (f x):: acc;
     goto listM_0;}
   ```
Problem 10: (Extra credit: 10 points) We studied two ways of defining equivalence for programs written in the language we are implementing: First, for operational semantics we defined observational equivalence, and second, for denotational semantics we used equivalence in the implementation language. This question is about the first type of equivalence, which is defined by observing how two terms behave in all possible contexts.

Argue that, for a lambda calculus with booleans and conditionals, observing just termination is not more powerful than observing both termination AND requiring that boolean results match.