Principles of Programming Languages (CS329) **Final exam**
Computer Science and Engineering **2004-11-25**
Indian Institute of Technology Bombay **09:30–13:00**

NAME _____ ROLL _____

This quiz has 5 printed page/s. Write your answer clearly within the spaces provided and on any last blank page. Start with rough work elsewhere, but do not attach rough work. Use the marks alongside each question for time management. **Illogical or incoherent answers are worse than wrong answers or even *no* answer, and may fetch negative credit.** You may not use any electronic computing device during the exam. You may use textbooks, class notes written by you, material downloaded **prior to the exam** from the course Web page, course news group, or the Internet, or notes made available by me for xeroxing. If you use class notes from other student/s, you must obtain them **prior to the exam** and **write down his/her/their name/s and roll number/s** here.

1. In class we saw that although `((λ a (a a)) (λ a (a a)))` is not in normal form, the expression remains unchanged upon applying a β-reduction, so there is no terminating reduction sequence. Write down a λ-expression where applying any 'reductions' makes the expression more complex, longer, or both.

		2

   `((λ a ((a a) f)) (λ a ((a a) f)))` should do it.

2. Here are lambda expressions defining `INC` (increments a given Church numeral), `ADD` (adds two Church numerals), `MUL` (multiplies two Church numerals) and `EXP` (raises 2 to the power given by the input Church numeral).

   $$\begin{aligned}
   \texttt{INC} &\equiv \texttt{(}\lambda\ \texttt{a (}\lambda\ \texttt{f (}\lambda\ \texttt{x ((a f) (f x)))))} \\
   \texttt{ADD} &\equiv \texttt{(}\lambda\ \texttt{a (}\lambda\ \texttt{b ((b INC) a)))} \\
   \texttt{MUL} &\equiv \texttt{(}\lambda\ \texttt{a (}\lambda\ \texttt{b ((b (ADD a)) } \bar{0}\texttt{)))} \\
   \texttt{EXP} &\equiv \texttt{(}\lambda\ \texttt{c ((c (MUL } \bar{2}\texttt{)) } \bar{1}\texttt{))}
   \end{aligned}$$

   If `EXP` is given input $\bar{n}$, what is the number of β-reduction steps needed (in normal-order reduction) until the result $\overline{2^n}$ is returned, as an asymptotic function of n?

		3

   The computation is dominated by the final multiplication of $\overline{2^{n-1}}$ by $\bar{2}$, which can be verified to take $O(2^{2n})$ reductions.

3. Neither static nor dynamic scoping models the dangerous variable capture possibilities opened up by macro substitution (as in the C preprocessor). In this question we will study a somewhat similar form of variable capture, in a variant of FL called FLL (the extra L is for "lazy" binding). In FLL, an identifier is looked up in an environment to return a *Macro*, which is itself a mapping from an environment to a value. At every point of use of an identifier, the use-point environment is passed to the *Macro* bound to the identifier to get a value. More formally,

   $$\begin{aligned}
   \mathit{Env} &= \texttt{Id} \rightarrow \mathit{Macro} \\
   \mathit{Macro} &= \mathit{Env} \rightarrow \mathit{Value} \\
   \mathit{Proc} &= \mathit{Macro} \rightarrow \mathit{Value}
   \end{aligned}$$

$$\begin{aligned} \mathcal{E} \;:\; \texttt{Expr} \to \mathit{Env} \to \mathit{Value} \;&=\; \texttt{Expr} \to \mathit{Macro} \\ \mathcal{E}\,[\![\texttt{(}E_p\ E_a\texttt{)}]\!] \;&=\; \lambda u\,\Big((\mathcal{E}\,[\![E_p]\!]\,u)\ \mathcal{E}\,[\![E_a]\!]\Big) \\ \mathcal{E}\,[\![\texttt{I}]\!] \;&=\; \lambda u\,\Big((u\ \texttt{I})\ u\Big). \end{aligned}$$

Give the value of the following expressions under FLL and static-scoped FL, with suitable explanations. (These expressions are all evaluated "at top level" without any other variable bindings available; i.e. they are not subexpressions of larger expressions.)

(a)
```
( (let ((x 3)) (lambda (y) y))
x )
```

		2

FL: `x` is undefined. FLL: 3.

(b)
```
( (let ((x 3)) (lambda (y) y))
a )
```

		2

FL: `a` is undefined. FLL: `a` is undefined.

(c)
```
( (let ((x 3)) (lambda (y) y))
y )
```

		2

FL: `y` is undefined. FLL: Infinite computation.

4. If Scheme were dynamically scoped, could you still implement $\mathcal{E}$ for a statically-scoped language like FL using Scheme? If yes, give an outline of your strategy; if no, give a formal proof that it is impossible. (The converse question has already been settled in class: we can obviously implement a dynamically-scoped language given Scheme.)

		2

(Outline:) First write out $\mathcal{E}$ for a statically-scoped target language using the dynamically-scoped source implementation language. Identify all function bodies that use variables not in the explicit parameter list. Eliminate these free variable occurrences by either replacing with the known value from the declaration environment, or by passing them as additional parameters (painful, but doable).

5. We have seen that currying multi-argument procedures is not valid with dynamic scoping. In this question, assume we are given FLD, a dynamically scoped dialect of Scheme with multi-argument procedures provided as a special form. FLD does not have `rec` or `letrec`, and you are not allowed to implement Y on your own.

   (a) Complete the following implementation of the factorial function and its invocation on the number 5.

```
( (lambda (fact) (fact 5))
  (lambda (n) .... )  )
```

		2

There is nothing special to do!

```
( (lambda (fact) (fact 5))
  (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)) )  ) ) )
```

(b) Explain how and why your code works. (Hint: the solution is extremely simple; the explanation takes more care.)

		2

Unlike in static scoping, in dynamic scoping there is no need to resolve the name `fact` in the function `(lambda (n)...` at the point of declaration because the declare-point environment is discarded anyway. The call-point environment *will* have a binding for `fact` which will do the trick.

(c) Can you find any similarity between what's going on here and the use of method dispatchers and `this` in FLOP? (You will get credit for this part only if you answered the earlier parts correctly.)

		1

(Some students just wrote "yes" or "no", I find that cheeky enough to deserve negative score; it's like saying "yes" when someone asks you "do you have the time?") The dynamic (call-point) environment is like the object ("`this`") in FLOP, and `fact` is like the method name. In FLOP, method names mentioned in bodies of other methods do not need to be resolved statically, but remain as "strings" until invocation. That's why mutually recursive methods in FLOP are no big deal.

6. Debuggers and profilers attach code dynamically to a program to monitor the *Store* and collect execution statistics. Suppose we want to keep track of the total number of returns from procedure calls up to any point of time in the execution of a program. The code required for this can be modeled as a function `CountReturns` from *Store* to *Store*, whose only action is to increment an integer in a fixed cell after every return is completed. Recall the standard denotational semantics for a procedure call:

$$\mathcal{E}\,[\![(E_p\ \ E_a)]\!] \;=\; \lambda u\ \lambda k\ \lambda s_0\ \Big(\mathcal{E}\,[\![E_p]\!]\ u\ \boxed{\lambda p\ \lambda s_1\ \Big(\mathcal{E}\,[\![E_a]\!]\ u\ \boxed{\lambda a\ \lambda s_2\ \underline{(p\ a\ k\ s_2)}}\ s_1\Big)}\ s_0\Big)$$

Modify the inner underlined portion suitably so that the profiling code is invoked properly. Write down only your replacement expression for the underlined part and explain your solution. (You do not need to implement `CountReturns`; use it as a black box.)

		4

Change the continuation k to insert the *Store* update ahead of it's effect:

$$(p\ a\ \underline{\lambda\, v\ \lambda\, s_3\ (k\ v\ \texttt{CountReturns}(s_3))}\ s_2)$$

Modifying s_2 itself will update the counter *before* the call, and several other solutions do not take care that an exception thrown from inside p should *prevent* bumping up the counter.

7. Consider exception handling under dynamic lookup of handler names as we have discussed in class. To make things simple for this question, assume the language has no *Store*.

$$\mathcal{E}\,[\![(\texttt{try}\ E_t\ \texttt{catch Ih Ia}\ E_b)]\!] \;=\; \lambda u\ \lambda w\ \lambda k\ \Bigg(\mathcal{E}\,[\![E_t]\!]\ u\ \boxed{\begin{array}{c}\texttt{Ih} \to \lambda e_a\ \Big(\mathcal{E}\,[\![E_b]\!]\ \boxed{\begin{array}{c}\texttt{Ia} \to e_a\\ \hline u\end{array}}\ w\ k\Big)\\ \hline w\end{array}}\ k\Bigg)$$

If an exception with the same name `Ih` is thrown from inside the handler body E_b, given the evaluation rule above, some "outer" `Ih` handler must catch it. Suppose we want to change the

rules so that, if an exception with the name `Ih` is thrown from inside E_b, then E_b is invoked *recursively* (unless E_b installs an inner `Ih` handler, of course). In case of a recursive E_b invocation, control must commence in the "outer" E_b.

Replace the dynamic environment in the rhs above (inside the big box) to implement this behavior. (Hint: you can use a construct in FL that can be desugared into a lambda expression.)

		5

$$\left(\texttt{rec}\ w' \boxed{\texttt{Ih} \to \lambda e_a \left(\mathcal{E}\ [\![E_b]\!] \boxed{\begin{array}{c}\texttt{Ia} \to e_a \\ \hline u\end{array}}\ w'\ k\right)} \right)$$
$$\boxed{\qquad\qquad w \qquad\qquad}$$

8. Here is some (very simple) Prolog code to generate subsets of elements from a given list.

```
comb1(0,_,[]).
comb1(N,[X|T],[X|Comb]) :- N>0, N1 is N-1, comb1(N1,T,Comb).
comb1(N,[_|T],Comb)     :- N>0, comb1(N,T,Comb).
```

Convince yourself that Prolog's search mechanism will generate all combinations as desired. (In particular, that termination is guaranteed even though the third rule does not reduce `N`.)

(a) Conventionally, which are the input parameters and which is the output parameter? What is the significance of each variable? (Continue reading this question to see how you should answer this part.)

		3

The first parameter is the size of the subset desired. The second parameter is the input list. The third parameter is the output subset list.

(b) What will happen if you enter the goal

```
comb1(2,X,[5,7]).
```

and keep looking for solutions until failure?

		4

Will never fail; will keep returning longer and longer solutions of the form:

```
1 ?- comb1(2,X,[5,7]).
X = [5, 7|_G360] ;
X = [5, _G359, 7|_G363] ;
X = [5, _G359, _G362, 7|_G366] ;
X = [5, _G359, _G362, _G365, 7|_G369] ;
X = [5, _G359, _G362, _G365, _G368, 7|_G372]
etc.
```

Next, write another implementation `comb2` that does not need arithmetic ("`N-1`") by completing the code below. Each `...` is a blank for you to fill. The first argument to `comb2` is the input list. The second argument must be a list of desired size, containing only free variables; these will be progressively bound to different subsets of the given size.

```
comb2(_,[]).
comb2([X|...],[...|Comb]):-comb2(T,...).
comb2([_|T],[X|...]):-comb2(...,[X|...]).
```

		6

```
comb2(_,[]).
comb2([X|T],[X|Comb]):-comb2(T,Comb).
comb2([_|T],[X|Comb]):-comb2(T,[X|Comb]).
```

9. Here is an informal example of a recursive type: a `TreeNode` is either an `Integer`, or it is a `RecordOf` two fields called `left` and `right`, both of type `TreeNode`. More formally, we can write the type "equation"

   `TreeNode = (UnionOf Integer (RecordOf (left TreeNode) (right TreeNode)))`,

   and the "solution" to this type equation gives the type `TreeNode`. At a high level this is like `rec`; to reflect this similarity, we can coin a type expression of the form

```
(rectype TreeNode (UnionOf Integer
                           (RecordOf (left TreeNode)
                                     (right TreeNode))) )
```

   Under this setting, it is reasonable to claim that

   `(rectype S (Func S Integer))` and `(rectype T (Func (Func T Integer) Integer))`

   are structurally equivalent. (Here `(Func T1 T2)` means a function that accepts an argument of type `T1` and returns a value of type `T2`.) Propose and justify an algorithm to test the structural equivalence of two recursive types `(rectype R Tr)` and `(rectype S Ts)`.

		4

   See page 2 of `http://www.cs.cornell.edu/courses/cs611/2001fa/scribe/lecture33.pdf` for an outline of the answer.

10. What is the value of the following expression in Scheme? Give at most 2 lines of justification.

```
(let ((r 1) (top (call-with-current-continuation (lambda (c) c))))
  ( (top (lambda (x) (1+ x) )) r ) )
```

		2

   Runtime error, with a message to the effect "cannot add 1 to a continuation or procedure".

Total: 46