

**CS 502. Compilers**  
**Midterm Examination**  
**October 17, 2006**

Name: \_\_\_\_\_

Answer each of the questions in the space provided. If you require additional space, use the blank pages attached at the end of the exam. Unless otherwise noted, the language under consideration is defined by the following grammar:

$$\begin{aligned} e \in Exp & ::= c \mid x \mid C \mid (\text{fn } x \Rightarrow e) \mid e(e) \mid C(e) \mid p(e \dots e) \mid \text{if } e_{test} \text{ then } e_{true} \text{ else } e_{false} \\ c \in Const & ::= 0 \mid 1 \mid \dots \\ C \in Constructor & ::= \text{Cons} \mid \text{true} \mid \text{false} \mid \dots \\ p \in Primop & ::= + \mid - \mid * \mid \dots \end{aligned}$$

In the program fragments you write, you are free to use any syntactic abstractions you wish including `let` and `letrec`.

Score: \_\_\_\_\_  
Maximum Score: 145

**Question 1** (50 points):

- (10 points) There is a one-to-one correspondence between the language of CPS and *direct* (non-CPS) expressions. Assuming that  $k$  ranges over continuation variables, write an expression that when CPS'ed would produce the following CPS expression; you may assume  $(\text{fn } y \Rightarrow y)$  defines the initial continuation. (The expression  $\lambda x.e$  is shorthand for “ $\text{fn } x \Rightarrow e$ .”, and  $\lambda_c x.e$  denotes a continuation expression.)

$$\lambda x k.f(x, (\lambda_c v_1. g(x, (\lambda_c v_2. v_1(v_2, (\lambda_c v_3. k v_3))))))$$

- (20 points) The ML language provides *exceptions* as way to express non-local transfers of control. The expression:

`handle  $e_1$   $e_2$`

evaluates  $e_1$  in the context of an exception handler defined by  $e_2$ . If  $e_1$  produces a value  $v$ , then the value of the entire expression is  $v$ . If  $e_1$  *raises* an exception with value  $v$ , then the value of the expression is the result of applying  $e_2$  to  $v$ . A subexpression of  $e_1$  can raise an exception by evaluating:

`raise  $e$`

The `raise` operator supplies the value of  $e$  to the closest dynamically-enclosing handler.

One way to compile exceptions to a CPS intermediate representation is to have every CPS expression take two continuations, one which represents the normal continuation, and the other which represents the *exception continuation*, the continuation to jump to if an exception is raised.

Show the CPS translation for a core language consisting of variables, conditional expressions, (single-argument) functions, and (single-argument) function application augmented with `handle` and `raise` expressions, by defining a function  $\mathcal{C}_{\text{exn}}$  that given an expression, a normal continuation, and an exception continuation returns a CPS'ed expression.

3. (10 points) The CPS transform makes explicit both order of evaluation, and the naming of intermediate computation(s). *A-normal form* is an alternative intermediate representation that avoids the creation of explicit continuations, encapsulating their role using `let`.

For example, the following expression:

```
f(g(a), let x = 1 in f(x))
```

The CPS translation would convert this program to:

```
g( λc v1. let x = 1 in f(λc v2. f(k0, (v1,v2)),x),a)
```

The A-normal form for the original program would be:

```
let t1 = g(a)
in let x = 1
  in let t2 = f(x)
    in f(t1,t2)
```

Based on this example, how does A-normal appear to satisfy the CPS properties of making explicit order-of-evaluation and naming of intermediate computation? In particular, can you distinguish between tail, non-tail and jump calls with A-normal form? If so, how?

4. (10 points) Informally describe the translation steps necessary to convert programs written in the language under consideration to A-normal form.

**Question 2.** (30 points)

Consider a program  $P$  of the form **begin**  $D$ ;  $S$  **end** where  $D$  consists of a collection of procedure declarations, and  $S$  is a sequence of statements drawn from the simple while language augmented with procedures and procedure calls. Procedures take a single call-by-value parameter. The label on **is** marks the program point upon entry to the procedure, and the label on **end** marks the program point on exit. The two labels associated with a call mark the program point immediately prior and immediately after the corresponding call.

$$\begin{aligned}
 e \in AExp & ::= x \mid n \mid a_1 \text{ op } a_2 \\
 b \in BExp & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op } b_2 \\
 D \in Decl & ::= \text{proc } p(\text{val } x) \text{ is}^{\ell_n} S \text{ end}^{\ell_x} \mid D D \\
 S \in Stmt & ::= [x:=a]^\ell \mid [\text{skip}]^\ell \mid S_1; S_2 \\
 & \quad \mid \text{call } p(a, z)^{\{\ell_y, \ell_z\}} \\
 & \quad \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\
 & \quad \mid \text{while } [b]^\ell \text{ do } S
 \end{aligned}$$

For each procedure of the form:

$$\text{proc } p(\text{val } x) \text{ is}^{\ell_n} S \text{ end}^{\ell_x}$$

the set  $IAV(p)$  contains the global variables that might be assigned directly or indirectly when  $p$  is called.

1. (10 points) Precisely define the set  $AV(S)$  that for each statement  $S$  gives the set of variables that could be assigned to  $S$ , but *ignores the effects of procedure calls*; and the set  $CP(S)$  that gives for each statement the set of procedure names that could be directly called in  $S$  but *ignores the effects of procedure calls*.
  
2. (15 points) Formulate a system of dataflow equations that specifies how to obtain the set  $IAV(p)$  in terms of  $AV$  and  $CP$ .

3. (5 points) Show the *IAV* equations and solutions for procedures `fib` and `add` in the following program. Explain whether you must compute the least or greatest solution.

```
begin
  proc fib(val z) is if z<3 then call add(1)
                    else (call fib(z-1); call fib(z-2))
  end;
  proc add(val u) is (y := y + u; u:= 0)
  end;
  y := 0; call fib(x)
end
```

Question 3 (20 points)

Consider the following SML definitions:

```
fun map f [] = []  
  | map f (x::xs) = f x :: map f xs;  
fun rev [] = []  
  | rev(x::xs) = rev(xs)@[x]
```

1. (5 points) Why is the following expression ill-typed under a type-checker that implements Hindley-Milner type inference using the value restriction?

```
val revlists = map rev
```

2. (10 points) How would you rewrite the expression to make it well-typed?

3. (5 points) Is your solution semantics-preserving in ML? Justify your answer



Question 5 (25 points)

Consider the following program fragment:

```
let val x = 1
    val y = 2
    val z = 3
    val f = fn w => x = y + w
in f 100
end
```

One possible closure-converted representation for this program fragment would be:

```
let val x = 1
    val y = 2
    val z = 3
    val code = (fn env => (fn w => (#x env) + (#y env) + w))
    val env = { x = x, y = y }
    val f = (code,env)
in
  (#1 f) (#2 f) 100
end
```

1. (5 points) What is the type of `code`?
2. (5 points) Based on this answer, justify the conclusion that closure-conversion does not in general preserve types under a Hindley-Milner type system.
3. (15 points) Construct an example that shows a well-typed program which fails to type-check after the closure conversion translation shown above.