Midterm Examination
CS 565
March 8, 2006

Name: ________________________________

Instructions: Answer all questions in the space provided. Extra blank pages are provided in the back. Partial credit will be given where appropriate.

Maximum score: 120
Attained score: ________________________
Question 1. (10 points)

1. (5 points) Write down the nameless representation (i.e., the representation via deBruijn indices) of the untyped λ term, λx.λy.x(λz.x(yz)).
   \( \lambda.\lambda.1(\lambda.2(1\ 0)) \)

2. (5 points) Let \( t_1 = \lambda.\lambda.((2\ 0)(1\ 0)) \) and let \( t_2 = \lambda(2\ 1) \). What is the result of substituting \( t_2 \) for \( t_1 \)'s only free variable?
   \( \lambda.\lambda.(\lambda.(4\ 3)\ 0)\ (1\ 0)) \)
Question 2. (10 points)

The subject expansion property states that for any terms $t$ and $t'$, if $t \rightarrow t'$ and $\Gamma \vdash t' : \tau$ for type $\tau$, then $\Gamma \vdash t : \tau$. Does this property hold for the simply-typed $\lambda$-calculus? If yes, prove it. If not, present a counter-example.

Counter-example: $(\lambda x:\textbf{bool}.\lambda y:\textbf{int}.y) (\lambda z.z)$
Question 3. (25 points)

The Fixed-Point Theorem for the untyped  \( \lambda \)-calculus states that for any well-formed \( \lambda \) term \( F \), there also exists a well-formed term \( X \) such that \( F \ X = X \).

The proof of this theorem is as follows:

\[
\text{Let } W \equiv \lambda x. F(x \ x) \text{ and } X \equiv W \ W. \text{ Then, } X \equiv W \ W \equiv ((\lambda x. F(x \ x)) \ W) \equiv F(W \ W) \equiv F \ X
\]

(In the following, consider \( \lambda xy. e \) as shorthand for \( \lambda x. \lambda y. e \).)

1. Using this result, determine the fixpoint for the following terms, and validate your answer:

(a) (5 points) \( \lambda \ x. \ x \)

\[(\lambda \ x. (x \ x))(\lambda \ x. (x \ x))\]

(b) (10 points) \( \lambda \ xy. (x \ y) \)

\[(\lambda \ xy. (x \ x) \ y)(\lambda \ xy. (x \ x) \ y)\]

2. (10 points) A fixed-point combinator is any term \( M \) such that \( \forall F. MF \to^* F(M \ F) \). Consider the term \( \Theta \equiv A \ A \) where \( A \equiv \lambda xy. y(x \ x \ y) \). Is \( \Theta \) a fixed-point combinator? If yes, prove it; if not, present a counter-example.

\[
\Theta \ F \equiv (\lambda \ xy. y(x \ x \ y)) \ A \ F \\
\to (\lambda \ y. y(A \ A \ y))F \\
\to F(A \ A \ F) \\
\equiv F(\Theta \ F)
\]
Question 4. (20 points)

The IMP language is given by the following grammar:

\[ e \in \text{AExp} ::= n | x | e_1 + e_2 | e_1 - e_2 | e_1 \ast e_2 \]
\[ b \in \text{BExp} ::= \text{true} | \text{false} | e_1 = e_2 | e_1 \leq e_2 | \neg b | b_1 \land b_2 | b_1 \lor b_2 \]
\[ c \in \text{Com} ::= \text{skip} | x := e | c_1 ; c_2 | \text{if } b \text{ then } c_1 \text{ else } c_2 | \text{while } b \text{ do } c \]

Prove the following statement by structural induction: For any boolean command \( b \) and any initial state \( \sigma \), such that \( \sigma(x) \) is even, if

\[ \text{while } b \text{ do } x := x + 2, \sigma \downarrow \sigma' \]

then \( \sigma'(x) \) is even.

**Reference:** The natural semantics for \( \text{while} \) loops is given by the following two commands:

\[
\begin{align*}
(b, \sigma) &\Downarrow \text{false} \\
(b, \sigma) &\Downarrow \sigma' \\
\text{(while } b \text{ do } c, \sigma) &\Downarrow \sigma' \\
\text{(while } b \text{ do } c, \sigma) &\Downarrow \sigma'
\end{align*}
\]

Consider a derivation \( D \) of this expression. There are two possible rules used at the top of \( D \):

1. \( \text{while } \text{false} \Downarrow \sigma = \sigma \) and \( \sigma(x) \) is even by the induction hypothesis.

2. \( \text{while } \text{true} \Downarrow \sigma = \sigma' \) and \( \sigma(x) \) is even by the induction hypothesis.

We know that \( \sigma'' = \sigma[x \mapsto \sigma(x) + 2] \) and thus, \( \sigma''(x) \) is even. Apply the induction hypothesis to the derivation rooted at \( D_1 \) to complete the proof.
**Question 5.** (30 points)

Certain languages allow *non-local* control-flow operations to be expressed using labels and jumps. Consider adding two new operators called `catch` and `throw` to the untyped call-by-value λ calculus. The operator `catch` labels the place to jump to, and `throw` jumps to the nearest catch with the same name, supplying a value. When a `throw` occurs, the value supplied by `throw` is the value returned by the `catch` expression which it matches.

For example, the following expression (written in ML syntax) returns 0 without evaluating `really-big-comp()`.

```ml
let fun p l = (case l of
  [] => 1
| 0::l' => throw(stop,0)
| l1::l2 => l1 * p(l2))
in catch(stop, p([1,2,0,really-big-comp()])
end
```

1. (10 points) Define a contextual grammar for the untyped call-by-value λ-calculus augmented with `catch` and `throw`.

   \[ E ::= \emptyset | v_1(v_2 \ldots E e_j \ldots e_n) | catch(\ell, E) | throw(\ell, E) \]

   \[ v ::= \lambda(x_1, \ldots, x_n).e \mid \overline{\text{throw}}(\ell, v) \]

2. (20 points) Define reduction rules based on this grammar to specify the desired semantics.

   **Local reduction rules:**

   \[ v(v_1, \ldots, v_n) \rightarrow e[v_1/x_1, \ldots, v_n/x_n] \text{ if } v = \lambda(x_1, \ldots, x_n).e \]

   \[ \text{catch}(\ell, v) \rightarrow v \]

   \[ \text{catch}(\ell, \text{throw}(\ell, v)) \rightarrow v \]

   \[ \text{catch}(\ell, \text{throw}(\ell', v)) \rightarrow \text{throw}(\ell', v) \]

   \[ \text{throw}(\ell, v) \rightarrow \overline{\text{throw}}(\ell, v) \]

   **Global reduction rules:**

   \[ E[v(v_1, \ldots, \overline{\text{throw}}(\ell, v), e_j, \ldots, e_n)] \rightarrow E[\overline{\text{throw}}(\ell, v)] \]

   \[ E[r] \rightarrow E[r'] \text{ if } r \rightarrow r' \]
Question 6. (25 points)

Some call-by-value languages provide explicit constructs to support lazy evaluation. One common approach is to define operations called delay and force:

- The delay procedure is used together with the procedure force to implement lazy evaluation or call by need. The expression: delay(expression) returns an object called a promise which at some point in the future may be asked (by the force procedure) to evaluate expression, and deliver the resulting value.
- The expression force(expression) evaluates expression to a promise, and returns the value of the promise. If no value has been computed for the promise, then a value is computed and returned. Significantly, the value of the promise forced is memoized or cached, so that if it is forced a second time, the previously computed value is returned.

For example:

```plaintext
val count = ref 0;
val p = delay( count := (!count) + 1;
                if ((!count) > x)
                    then count
                    else force(p));
val x = 5; (* x = 5 *)
p; (* a promise *)
force(p); (* 6 *)
p; (* a promise *)
(x:=10; force(p)); (* 6 *)
```

Consider the incorporation of delay and force into the simply-typed λ calculus.

1. (5 points) Show the typing rules for delay and force. You may assume the existence of a new type constructor prom such that τ prom denotes the type of a promise holding a value of type τ.

   \[
   \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{delay}(e) : \tau \text{ prom}} \quad \frac{\Gamma \vdash e : \tau \text{ prom}}{\Gamma \vdash \text{force}(e) : \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{promise}(e) : \tau \text{ prom}}
   \]

2. (10 points) Using a small-step operational semantics, formalize the informal description given above for the simply typed λ-calculus.

   (Do these evaluation rules adequately capture the memoization property of a promise?)

   \[
   \frac{\text{delay}(e) \rightarrow \text{promise}(e)}{e \rightarrow e', e \neq \text{promise}(e')} \quad \frac{\text{force}(e) \rightarrow \text{force}(e')}{\text{force}(\text{promise}(e)) \rightarrow \text{force}(\text{promise}(e'))} \quad \frac{\text{force}(\text{promise}(v)) \rightarrow v}{
   }
   \]


3. (10 points) The type preservation theorem can be stated as:

\[
\text{If } \vdash e : \tau \text{and } e \rightarrow e' \text{then } \vdash e' : \tau
\]

One way to prove this theorem is by induction on the derivation of \( e \rightarrow e' \). Show the cases for the new constructs (\texttt{delay} and \texttt{force}) in this proof.

The case for \texttt{delay}(e) is straightforward. A \texttt{delay} expression yields an object of type \( \tau \text{prom} \) as does the evaluation of the promise that the \texttt{delay} yields.

To prove the theorem for \texttt{force}(e) we examine the remaining three rules. When \( e \) is not a promise, the theorem holds trivially by the induction hypothesis. When \( e \) is a promise, but the promise does not hold a value, the theorem is satisfied again by the induction hypothesis applied to the rule’s antecedent. In the case where it is a value, we know by the typing rules for \texttt{force} and \texttt{prom} that \texttt{promise}(v) must be of type \( \tau \text{prom} \), and thus \( v \) must be of type \( \tau \).