CSE 526: Principles of Programming Languages

Spring 2012 **Final Exam** Max: 40 points

May 10, 2012 Duration: 2h 30m

Name: ______________________________

USB ID Number: ______________________________

INSTRUCTIONS

Read the following carefully before answering any question.

- Make sure you have filled in your name and USB ID number in the space above.
- Write your answers in the space provided; Keep your answers brief and precise.
- The exam consists of **6** questions, in **13** pages (including this page) for a total of **40** points.

GOOD LUCK!

Question	**Max.**	**Score**
1.	8	
2.	5	
3.	6	
4.	10	
5.	5	
6.	6	
Total:	40	

1. [8 points] Recall the definition of the language of natural numbers and booleans, **NB**. The syntax and the inference rules for single-step semantics of **NB** is shown below.

Terms and Values:

$$
\begin{array}{rcl}
t & ::= & \boxed{\textit{\textbf{Terms:}}} \\
 & & 0 \\
 & | & \texttt{succ}(t) \\
 & | & \texttt{pred}(t) \\
 & | & \texttt{iszero}(t) \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \texttt{if}(t, t, t)
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \boxed{\textit{\textbf{Values:}}} \\
 & & \texttt{true} \\
 & & \texttt{false} \\
 & | & nv \\
nv & ::= & \texttt{0} \\
 & | & \texttt{succ}(nv)
\end{array}
$$

Evaluation Rules:

$$\frac{t_1 \to t_1'}{\texttt{succ}\ t_1 \to \texttt{succ}\ t_1'} \quad \text{E-SUCC} \qquad \frac{t_1 \to t_1'}{\texttt{iszero}\ t_1 \to \texttt{iszero}\ t_1'} \quad \text{E-ISZERO}$$

$$\frac{t_1 \to t_1'}{\texttt{pred}\ t_1 \to \texttt{pred}\ t_1'} \quad \text{E-PRED} \qquad \texttt{iszero}\ \texttt{succ}\ nv_1 \to \texttt{false} \quad \text{E-ISZEROSUCC}$$

$$\texttt{pred}\ \texttt{0} \to \texttt{0} \quad \text{E-PREDZERO} \qquad \frac{t_1 \to t_1'}{\texttt{if}(t_1, t_2, t_3) \to \texttt{if}(t_1', t_2, t_3)} \quad \text{E-IF}$$

$$\texttt{pred}\ \texttt{succ}\ nv_1 \to nv_1 \quad \text{E-PREDSUCC} \qquad \texttt{if}(\texttt{true}, t_2, t_3) \to t_2 \quad \text{E-IFTRUE}$$

$$\texttt{iszero}\ \texttt{0} \to \texttt{true} \quad \text{E-ISZEROZERO} \qquad \texttt{if}(\texttt{false}, t_2, t_3) \to t_3 \quad \text{E-IFFALSE}$$

Extend **NB** to a language, called **ZB**, that can represent and compute with *integer* values (i.e. positive numbers, zero, as well as negative numbers) instead of just natural numbers. **ZB** should be such that every term in **NB** is in **ZB** as well.

(a) Define the language (set of terms) of **ZB**. If you want, you may state it as an extension to **NB** by stating only the newly introduced terms.

[Contd. on next page]

(b) Define the set of values of **ZB**. If you want, you may state it as an extension to **NB** by stating only the newly introduced values.

(c) Define the set of evaluation rules for **ZB**. If you want, you may fully state only the newly introduced/modified rules, and simply name all the rules that are carried over from **NB** unchanged.

[Contd. on next page]

(d) Using your evaluation rules, find the normal form of `succ(succ(pred(pred(0))))`. Show the evaluation sequence.

2. [5 points] The `let` construct in the extended lambda calculus is of the form $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$. The "$x = t_1$" part is called a "let binding". In the `let` construct defined in the textbook, each `let` expression has exactly one let binding.

Consider a further extension that allows a *sequence* of let bindings to be used within a `let`. More formally, the extended `let` construct is of the form

$$\texttt{let}\ x_1 = t_1\texttt{;}\ x_2 = t_2\texttt{;}\ \cdots\texttt{;}\ x_n = t_n\ \texttt{in}\ t$$

The single step semantics for the extended construct is given by the following rules:

$$\frac{[x_1 \mapsto v_1, \ldots x_k \mapsto v_k]t_{k+1} \longrightarrow t'_{k+1}}{\begin{array}{r}\texttt{let}\ x_1 = v_1\ \texttt{;}\ \ldots\ \texttt{;}\ x_k = v_k\ \texttt{;}\ x_{k+1} = t_{k+1}\ \texttt{;}\ \ldots x_n = t_n\ \texttt{in}\ t \longrightarrow \\ \texttt{let}\ x_1 = v_1\ \texttt{;}\ \ldots\ \texttt{;}\ x_k = v_k\ \texttt{;}\ x_{k+1} = t'_{k+1}\ \texttt{;}\ \ldots x_n = t_n\ \texttt{in}\ t\end{array}} \qquad \text{T-LETK}$$

$$\texttt{let}\ x_1 = v_1\ \texttt{;}\ \ldots\ \texttt{;}\ x_n = v_n\ \texttt{;}\ \texttt{in}\ t \longrightarrow [x_1 \mapsto v_1, \ldots x_n \mapsto v_n]t \qquad \text{T-LET}$$

(An aside: this form of `let` is SML, but not in OCAML.)

Can the extended let construct be obtained as a derived form of the simpler let construct introduced in the text? If so, give the definition of the derived form. If not, justify why a derived form is not possible.

3. [6 points] Consider the extensions to lambda calculus with **NB**, datatypes and recursion (i.e. contents of Chapter 11). The `letrec` construct was introduced to enable easier specification of recursive functions. In particular, `letrec` was described as a derived form, in terms of `fix` as follows:

$$\begin{array}{l} \texttt{letrec}\ x : T_1 = t_1\ \texttt{in}\ t_2 \\ \stackrel{def}{=}\ \texttt{let}\ x =\ \texttt{fix}\ (\lambda\ x : T_1.\ t_1)\ \texttt{in}\ t_2 \end{array}$$

Using `letrec`, one can define directly recursive functions such as plus:

$$\begin{array}{l} \texttt{letrec}\ \text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat} = \\ \qquad \lambda m{:}\text{Nat}.\ \lambda n{:}\text{Nat}.\ \texttt{if}(\texttt{iszero}(m), n, \texttt{succ}(\text{plus}\ (\texttt{pred}(m))\ n))\ \texttt{in} \ldots \end{array}$$

OCAML and SML have a more expressive construct that permits definition of mutually recursive functions using the "`and`" connective. Along the same lines, consider extending the language with a `letmrec` construct that permits definition of pairs of mutually exclusive functions. The syntax of `letmrec` construct is:

$$\texttt{letmrec}\ x_1 : T_1 = t_1\ \texttt{and}\ x_2 : T_2 = t_2\ \texttt{in}\ t_3$$

For example the following is a definition of mutually recursive even and odd functions using the `letmrec` construct:

$$\begin{array}{rcl} \texttt{letmrec}\ \text{even: Nat} \to \text{Bool} & = & (\lambda m.\ \texttt{if}(\texttt{iszero}(m), \texttt{true}, \text{odd}(\texttt{pred}(m))) \\ \texttt{and}\ \text{odd: Nat} \to \text{Bool} & = & (\lambda m.\ \texttt{if}(\texttt{iszero}(m), \texttt{false}, \text{even}(\texttt{pred}(m))\ \text{in} \ldots \end{array}$$

Define the semantics and typing rules of terms with `letmrec`. If possible, define `letmrec` as a derived form based on existing constructs such as `letrec`. Alternatively, you may define additional evaluation and typing rules.

[You may continue the answer, if necessary, on next page]

4. [10 points] Consider the extensions to lambda calculus with **NB** and records. (i.e. part of the contents of Chapter 11). The typing rules for typed lambda calculus with **NB** and records is summarized below. The evaluation rules for call-by-value lambda calculus and records are also summarized below (the evaluation rules for **NB** are given in Question 1).

Terms, Values and Types:

t ::=	$\ldots$ Terms from **NB**		
$\mid$	x	Variables	
$\mid$	$t\ t$	Application	
$\mid$	$\lambda x : T.t$	Abstraction	
$\mid$	$\{l_i = t_i{}^{i \in 1 \ldots n}\}$	Record	
$\mid$	$t\,.\,l$	Projection	

v ::=	$\ldots$ Values from **NB**	
$\mid$	$\lambda x : T.t$	
$\mid$	$\{l_i = v_i{}^{i \in 1 \ldots n}\}$	record values
T ::=	$\texttt{Nat}$	
$\mid$	$\texttt{Bool}$	
$\mid$	$T \to T$	
$\mid$	$\{l_i : T_i{}^{i \in 1 \ldots n}\}$	type of records

Evaluation Rules (in addition to those of **NB**):

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \quad \text{E-App1}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \quad \text{E-Abs2}$$

$$(\lambda x : T.\ t_1)\ v_2 \to [x \mapsto v_2]t_1 \quad \text{E-AppAbs}$$

$$\{l_i = v_i{}^{i \in 1 \ldots n}\}\,.\,l_j \to v_j \quad \text{E-ProjRcd}$$

$$\frac{t_1 \to t_1'}{t_1\,.\,l \to t_1'\,.\,l} \quad \text{E-Proj}$$

$$\frac{t_j \to t_j'}{\begin{array}{c}\{l_i = v_i{}^{i \in 1..(j-1)}, l_j = t_j, l_k = t_k{}^{k \in (j+1) \ldots n}\} \to \\ \{l_i = v_i{}^{i \in 1..(j-1)}, l_j = t_j', l_k = t_k{}^{k \in (j+1) \ldots n}\}\end{array}} \quad \text{E-Rcd}$$

Typing Rules:

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \quad \text{T-True}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \quad \text{T-False}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if}(t_1, t_2, t_3) : T} \quad \text{T-If}$$

$$\Gamma \vdash \texttt{0} : \texttt{Nat} \quad \text{T-Zero}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ t_1 : \texttt{Nat}} \quad \text{T-Succ}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ t_1 : \texttt{Nat}} \quad \text{T-Pred}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ t_1 : \texttt{Bool}} \quad \text{T-IsZero}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 \ :\ T_1 \to T_2} \quad \text{T-Abs}$$

$$\frac{\Gamma \vdash s : T_1 \to T_2 \qquad \Gamma \vdash t : T_1}{\Gamma \vdash (s\ t) : T_2} \quad \text{T-App}$$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i{}^{i \in 1 \ldots n}\} : \{l_i : T_i{}^{i \in 1 \ldots n}\}} \quad \text{T-Rcd}$$

$$\frac{\Gamma \vdash t : \{l_i : T_i{}^{i \in 1 \ldots n}\}}{\Gamma \vdash t\,.\,l_j : T_j} \quad \text{T-Proj}$$

Consider further extending this language with terms of the form $\texttt{isequal}(t_1, t_2)$ where t_1 and t_2 are terms.

At a high level, the intent of `isequal` is to determine whether or not the two terms have the same normal form or not. A term of the form $\texttt{isequal}(t_1, t_2)$ is evaluated by first evaluating t_1 to a value v_1, then t_2 to a value v_2, and then evaluating to `true` if v_1 and v_2 are identical, and to `false` otherwise.

Note that once we have `isequal`, we can treat $\texttt{iszero}(t)$ as a derived form, defined as $\texttt{isequal}(t, \texttt{0})$.

[Contd. on next page]

(a) Give the evaluation rules that need to be added when `isequal`(t, t) is added to the language.

[Contd. on next page]

(b) Give the typing rules that need to be added when $\texttt{isequal}(t, t)$ is added to the language (i.e. the set of terms).

(c) The ***progress*** *property* states that if t is well-typed, then either t is a value or there is a term t' such that $t \to t'$. Does the progress property hold when your typing and evaluation rules are added to treat the addition of `isequal`? For this part, if the property holds, you need to give a detailed justification but not give a formal proof. If the property does not hold, you need to give a counter example.

5. [5 points] Consider the addition of references to extended lambda calculus (Chapter 13). The typing rules for the calculus with references is summarized below. (See Question 4 for the typing rules for the calculus with **NB** and records.)

Terms, Values and Types:

t	::=	… Terms from Q.4	
	$\mid$	$\texttt{ref}\ t$	Reference Creation
	$\mid$	$\texttt{!}\ t$	Dereference
	$\mid$	$t\ \texttt{:=}\ t$	Assignment
	$\mid$	$\texttt{unit}$	Unit constant
	$\mid$	$t\ \texttt{;}\ t$	Sequence

v	::=	… Values from Q.4	
	$\mid$	$\texttt{unit}$	Unit value
	$\mid$	l	Locations
T	::=	… Types from Q.4	
	$\mid$	$\texttt{Unit}$	Unit type
	$\mid$	$\texttt{Ref}\ T$	Reference types

Additional Typing Rules:

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \quad \text{T-UNIT}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_2} \quad \text{T-REF}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ \texttt{;}\ t_2 : T} \quad \text{T-SEQ}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_2}{\Gamma \vdash \texttt{!}\ t_1 : T_2} \quad \text{T-DEREF}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ \texttt{:=}\ t_2 : \texttt{Unit}} \quad \text{T-ASSIGN}$$

(a) For each of the following terms, state its type if it is well typed; if it is not well-typed, give a brief justification.

i. $\lambda x : \texttt{Nat}.\ \texttt{ref}\ x$

ii. $\lambda x : \texttt{Ref Nat}.\ \texttt{ref !}\ x$

iii. $\lambda x : \texttt{Ref Ref Nat}.\ x\ \texttt{:= ref !}x$

iv. $\lambda x : \texttt{Ref Ref Nat}.\ (x\ \texttt{:= !}x)\ \texttt{;}\ \texttt{!}\ x$

[Contd. on next page]

(b) For each of the following terms, determine if there exist types $T_1, T_2, \ldots$ such that the term is well-typed. If so, state the most general type of the term (i.e. its principal type). If not, give a brief justification.

i. $\lambda x : \mathtt{T}_1.\ x \ \mathtt{:=}\ 0$

ii. $\lambda x : \mathtt{T}_1.\ \lambda y : \mathtt{T}_2.\ (\mathtt{!}\ x)\ \mathtt{:=}\ \mathtt{succ}(\mathtt{!}\ y)$

iii. $\lambda x : \mathtt{T}_1.\ (\mathtt{!}\ (\mathtt{!}\ x))\ \mathtt{:=}\ \mathtt{!}\ x$

6. [6 points] Consider the following Prolog program:

```
p(A, S, []) :- f(A, S).
p(A, S, [X|Xs]) :- t(A, S, X, T), p(A, T, Xs).

t(1, 1, a, 1).
t(1, 1, b, 2).
t(2, 1, a, 2).
t(2, 1, b, 1).
t(2, 2, b, 2).
t(3, 1, a, 1).
t(3, 1, b, 2).
t(3, 2, a, 1).
t(3, 2, b, 2).

f(1, 2).
f(2, 2).
f(3, 2).
```

(a) What are the answers to query `p(1, Q, [a,a,b])`?

(b) What are the answers to query `p(L, 1, [a,a,b])`?

(c) What are the answers to query `p(1, 1, X), p(2, 1, X)`?

(d) What are the answers to query `p(1, 1, X), p(3, 1, X)`?

END OF EXAM