Name: Hemos ID:

CSE-321 Programming Languages 2014
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Total
Score						
Max	10	20	30	15	25	100

- There are five problems on 10 pages in this exam.
- The maximum score for this exam is 100 points.
- Be sure to write your name and Hemos ID.
- In Problem 1, write your answers exactly as you would type on the screen. The grading for Problem 1 will be strict (*i.e.,* no partial points).
- When writing individual proof steps in Problem 2, please write *conclusion* in the left and *justification* in the right, as in the course notes.
- For a 'true or false' question, a wrong answer has a penalty equal to the points assigned to it.
- You have one and a half hours for this exam.

1 OCaml Programming [10 points]

In this problem, you will implement a number of functions satisfying given descriptions. You should write one character per blank. For example, the following code implements a sum function.

```
l e t _ r e c _ s u m _ n _ = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ i f _ n _ = _ 0 _ t h e n _ 0 _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ e l s e _ n _ + _ s u m _ ( n _ - _ 1 ) _ _ _ _ _ _ _ _ _
```

Question 1. [5 points] Tail-recursive `union` for computing the union of two sets

(Type) `union: 'a list -> 'a list -> 'a list`

(Description) `union` S T returns a set that includes all elements of S and T without duplication of any element. The order of elements in the return value does not matter. You may use the `List.exists` function:

```
# List.exists;;
- : ('a -> bool) -> 'a list -> bool = <fun>
```

`union` must be a <u>tail</u>-recursive function and should not introduce auxiliary functions other than `List.exists`.

(Invariant) Each input set consists of distinct elements.

(Example) `union [1; 2; 3] [2; 4; 6]` returns `[3; 1; 2; 4; 6]`.

```
let rec union l1 l2 =

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Question 2. [5 points] `fold_left` using `fold_right`

(Type) `fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

(Description) Implement `fold_left` using the `fold_right` function:

```
# open List;;
# fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

`fold_left` and `fold_right` are specified as follows:

$$\begin{array}{rcll} \texttt{fold\_left}\ f\ a_0\ [l_1, l_2, \cdots, l_n] & = & f\ (\cdots f\ (f\ (a_0, l_1), l_2) \cdots, l_n) & (n \geq 0) \\ \texttt{fold\_right}\ f\ [l_1, l_2, \cdots, l_n]\ a_0 & = & f\ (l_1, \cdots f\ (l_{n-1}, f\ (l_n, a_0)) \cdots) & (n \geq 0) \end{array}$$

(Hint) OCaml is a <u>function</u>al language :-)

```
let fold_left f a l =
```

_ _

_ _

2 Inductive proof on strings of matched parentheses [20 points]

In this problem, we study a system of strings of matched parentheses. First we define a syntactic category paren for strings of parentheses:

$$\mathsf{paren} \qquad s \ ::= \ \epsilon \mid (s \mid)s$$

ϵ stands for the empty string ($i.e.$, $\epsilon s = s = s\epsilon$). paren specifies a language of strings of parentheses with no constraint on the use of parentheses.
To identify strings of matched parentheses, we introduce a judgment $s\ \mathsf{lparen}$ with the following inference rules:

$$\frac{}{\epsilon\ \mathsf{lparen}}\ Leps \qquad \frac{s_1\ \mathsf{lparen} \quad s_2\ \mathsf{lparen}}{(s_1)\ s_2\ \mathsf{lparen}}\ Lseq$$

We also introduce another judgment $s\ \mathsf{tparen}$ for identifying strings of matched parentheses:

$$\frac{}{\epsilon\ \mathsf{tparen}}\ Teps \qquad \frac{s_1\ \mathsf{tparen} \quad s_2\ \mathsf{tparen}}{s_1\ (s_2)\ \mathsf{tparen}}\ Tseq$$

Our goal is to prove Theorem 2.1. If you need a lemma to complete the proof, state the lemma, prove it, and use it in your proof of Theorem 2.1.

Theorem 2.1. *If $s\ \mathsf{tparen}$, then $s\ \mathsf{lparen}$.*

3 Untyped λ-Calculus [30 points]

The abstract syntax of the untyped λ-calculus is given as follow:

$$\text{expression} \quad e \quad ::= \quad x \mid \lambda x.\, e \mid e\ e$$

We may use other names for variables (*e.g.*, z, s, t, f, *arg*, *accum*, and so on). The scope of a λ-abstraction $\lambda x.\, e$ extends as far to the right as possible. We use a reduction judgment of the form $e \mapsto e'$:

$$e \mapsto e' \qquad \Leftrightarrow \qquad e \textit{ reduces to } e'$$

Question 1. [2 points] Given a function f in the untyped λ-calculus, we write f^n for the function applying f exactly n times, *i.e.*, $f^n = f \circ f \cdots \circ f$ (n times). A fixed point of f is also a fixed point of f^n if $n \geq 1$. True or false?

____

Question 2. [5 points] Complete the inductive definition of substitution. You may use $[x \leftrightarrow y]e$ for the expression obtained by replacing all occurrences of x in e by y and all occurrences of y in e by x.

$[e/x]x \quad = \quad$ _______________

$[e/x]y \quad = \quad$ _______________ $\quad$ *if* $x \neq y$

$[e/x](e_1\ e_2) \quad = \quad$ _________________________

$[e'/x]\lambda x.\, e \quad = \quad$ __________________

$[e'/x]\lambda y.\, e \quad = \quad$ ______________________ $\quad$ *if* $x \neq y, y \notin FV(e')$

$[e'/x]\lambda y.\, e \quad = \quad \lambda z.$ ________________________ $\quad$ *if* $x \neq y, y \in FV(e')$
where $z \neq y$, $z \notin FV(e)$, $z \neq x$, $z \notin FV(e')$

Question 3. [3 points] Show the reduction sequence under the call-by-name strategy. Underline the redex at each step.

$(\lambda x.\, \lambda y.\, y\ x)\ ((\lambda x.\, x)\ (\lambda y.\, y))\ (\lambda z.\, z)$

$\mapsto$

$\mapsto$

$\mapsto$

$\mapsto$

Question 4. [2 points] Give an expression whose reduction neither gets stuck nor terminates under the call-by-value strategy.

____

Question 5. [3 points] Complete the reduction rules for the call-by-value strategy. You may use the substitution that you defined earlier:

$$\frac{\qquad\qquad}{\quad \mapsto \quad}$$

$$\frac{\qquad\qquad}{\quad \mapsto \quad}$$

$$\frac{\qquad\qquad}{\quad \mapsto \quad}$$

Programming in the untyped λ-calculus

A Church numeral encodes a natural number n as a λ-abstraction $\hat{n}$ which takes a function f and returns $f^n = f \circ f \cdots \circ f$ (n times):

$$\hat{n} \quad = \quad \lambda f.\, f^n \quad = \quad \lambda f.\, \lambda x.\, f\ f\ f\ \cdots f\ x$$

Question 6. [5 points] Define the function exp for exponentiation such that exp $\hat{m}\ \hat{n}$ evaluates to a church numeral for the product of n copies of m. In other words, exp $\hat{m}\ \hat{n} \mapsto^* \widehat{m^n}$.

exp = ______________________________

de Bruijn expressions

Following is the definition of de Bruijn expressions:

$$\begin{array}{rrcl} \textsf{de Bruijn expression} & M & ::= & n \mid \lambda.\, M \mid M\ M \\ \textsf{de Bruijn index} & n & ::= & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

Question 7. [5 points] Complete the definition of $\sigma_n(M, N)$ for substituting N for every occurrence of n in M where N may include free variables. You may use $\tau_i^n(N)$.

$\sigma_n(M_1\ M_2, N) \quad = \quad$ ____________________________

$\sigma_n(\lambda.\, M, N) \quad = \quad$ ____________________________

$\sigma_n(m, N) \quad = \quad$ ____________________________ if $m < n$

$\sigma_n(n, N) \quad = \quad$ ____________________________

$\sigma_n(m, N) \quad = \quad$ ____________________________ if $m > n$

Question 8. [5 points] Complete the definition of $\tau_i^n(N)$, as given in the course notes, for shifting by n (*i.e.*, incrementing by n) all de Bruijn indexes in N corresponding to free variables, where a de Bruijn index m in N such that $m < i$ does not count as a free variable.

$\tau_i^n(N_1\ N_2) \quad = \quad$ ____________________________

$\tau_i^n(\lambda.\, N) \quad = \quad$ ____________________________

$\tau_i^n(m) \quad = \quad$ ____________________________ if $m \geq i$

$\tau_i^n(m) \quad = \quad$ ____________________________ if $m < i$

4 Simply-typed λ-calculus [15 points]

In this section, we assume the simply-typed λ-calculus. We use A, B, C for metavariables for types, e for expressions, and Γ for typing contexts. We use a typing judgment $\Gamma \vdash e : A$ to mean that under typing context Γ, expression e has type A. We use a reduction judgment $e \mapsto e'$ to mean that expression e reduces to expression e'.

The abstract syntax for the simply typed λ-calculus is given as follows:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \\
\textsf{base type} & P & ::= & \textsf{bool} \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\, e \mid \textsf{true} \mid \textsf{false} \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Question 1. [**3 points**] Type safety guarantees that evaluating a well-typed expression (*i.e.*, running a well-typed program) eventually terminates, never producing non-termination. True or false?

____

Question 2. [**3 points**] State the weakening property of typing judgments:
(Weakening).

____

Question 3. [**6 points**] State two theorems, progress and type preservation, constituting type safety:

(Progress).

____

____

(Type preservation).

____

Question 4. [**3 points**] Consider the extension of the simply-typed λ-calculus with sum types:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & \cdots \mid A + A \\
\textsf{expression} & e & ::= & \cdots \mid \textsf{inl}_A\ e \mid \textsf{inr}_A\ e \mid \textsf{case}\ e\ \textsf{of}\ \textsf{inl}\ x.\, e \mid \textsf{inr}\ x.\, e
\end{array}
$$

Write the typing rule for $\textsf{case}\ e\ \textsf{of}\ \textsf{inl}\ x.\, e \mid \textsf{inr}\ x.\, e$:

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad}{\qquad\qquad\qquad\qquad\qquad\qquad}\ {+}\textsf{E}$$

5 Evaluation contexts and abstract machine C [25 points]

Consider the following fragment of the simply-typed λ-calculus:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \rightarrow A \\
\textsf{base type} & P & ::= & \textsf{bool} \\
\textsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e
\end{array}
$$

Question 1. [4 points] Give the definition of evaluation contexts for the call-by-name strategy.

evaluation context κ ::= ____

Question 2. [3 points] Under the call-by-value strategy, give an expression e such that

- $e = \kappa[\![e']\!]$ where e' is the redex, and
- e reduces to e_0 that is decomposed to $\kappa'[\![e'']\!]$ where e'' is the redex for the next reduction and $\kappa \neq \kappa'$.

____

Abstract machine C

Consider the simply-typed λ-calculus under the call-by-value reduction strategy. The abstract machine C uses two states:

$$\textsf{state} \quad s \quad ::= \quad \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v$$

- $\sigma \blacktriangleright e$ means that the machine is currently reducing $\sigma[\![e]\!]$, but has yet to analyze e.
- $\sigma \blacktriangleleft v$ means that the machine is currently reducing $\sigma[\![v]\!]$ and has already analyzed v.

Question 3. [3 points] Give the definitions of frames and stacks:

frame ϕ ::= ____

stack σ ::= ____

Question 4. [10 points] Give the rules for the reduction judgment $s \mapsto_{\mathsf{C}} s'$ for the abstract machine C.

Question 5. [5 points] State the correctness of the abstract machine C in terms of the reduction judgment $e \mapsto e'$. You may use $\mapsto^*$ and $\mapsto_{\mathsf{C}}^*$ for the reflexive and transitive closures of $\mapsto$ and $\mapsto_{\mathsf{C}}$.