Name: Hemos ID:

CSE-321 Programming Languages 2012
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	14	15	29	20	7	15	100

- There are six problems on 24 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 Problems 2 and 6, please write *conclusion* in the left blank and *justification* in the right blank, as in the course notes.
- You have three hours for this exam.

1 SML Programming [14 pts]

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.

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

Question 1. [**4 pts**] The definition of `'a tree` for binary trees is as follows:

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
```

Give a *tail-recursive* implementation of `inorder` for an inorder traversal of binary trees.

(Type) `inorder: 'a tree -> 'a list`

(Description) `inorder` t returns a list of elements produced by an inorder traversal of the tree t.

(Example) `inorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[1, 3, 2, 7, 4]`.

(Hint) `inorder` can be implemented as follows:

```
fun inorder t =
    let
       fun inorder' (t' : 'a tree) (post : 'a list) : 'a list = ...
    in
       inorder' t [ ]
    end
```

`post` will be a list of elements to be appended to the result of an inorder traversal of `t'`. For example, when `inorder'` visits the node marked `2` in the tree below, `post` will be bound to `[1, 6, 3, 7]`.

```
       1

   2       3

 4   5   6   7
```

```
fun inorder t =
```

In Questions 2, assume the following function `foldr`:

(Type) `foldr: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

(Description) `foldr` f e l takes e and the last item of l and applies f to them, feeds the function with this result and the penultimate item, and so on. That is, `foldr` f e $[x_1, x_2, ..., x_n]$ computes $f(x_1, f(x_2, ..., f(x_n, e)...))$, or e if the list is empty.

Question 2. [5 pts] Complete the function `lrev` using `foldr`. You may use the operator @ for list concatenation.

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

(Description) `lrev` l returns the reversed list of an input list l.

(Example) `lrev [1, 2, 3, 4]` returns `[4, 3, 2, 1]`

```
fun lrev l =
    let
      val f = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

    in

      foldr f _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

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

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

    end
```

Question 3. [**5 pts**] A signature `SET` for sets is given as follows:

```
signature SET =
sig
    type 'a set
    val empty : ''a set
    val member : ''a set -> ''a -> bool
    val insert : ''a set -> ''a -> ''a set
    val intersection : ''a set -> ''a set -> ''a set
    val difference : ''a set -> '' a set -> ''a set
end
```

- `empty` is an empty set.
- `member` s x returns `true` if x is a member of s; otherwise it returns `false`.
- `insert` s x adds x to the set s and returns the resultant set.
- `intersection` s t returns the intersection of s and t.
- `difference` s t returns the set of elements which are members of s, but not members of t.

Give a functional representation of sets by implementing a structure `SetFun` of signature `SET`. In your answer, do not use the `if/then/else` construct. You may use `true`, `false`, `not`, `andalso`, and `orelse`.

```
structure SetFun : SET where type 'a set = 'a -> bool =
   struct
      type 'a set = 'a -> bool

      val empty = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

      fun member s = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

      fun insert s x = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

      fun intersection s t = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

      fun difference s t = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
   end
```

2 Inductive proof on strings of matched parentheses [15 pts]

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

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

To identify strings of matched parentheses, we introduce a judgment s 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 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 lparen, then s tparen.*

3 λ-Calculus [29 pts]

In this problem, we study the properties of the untyped λ-calculus:

$$\begin{array}{rrcl} \textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \\ \textsf{value} & v & ::= & \lambda x.\, e \end{array}$$

The reduction judgment is as follows:

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

Question 1. [5 pts] 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.

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

$$[e/x]y \quad = \quad \underline{\hspace{6cm}} \qquad \textit{if } x \neq y$$

$$[e/x](e_1\ e_2) \quad = \quad \underline{\hspace{6cm}}$$

$$[e'/x]\lambda x.\, e \quad = \quad \underline{\hspace{6cm}}$$

$$[e'/x]\lambda y.\, e \quad = \quad \underline{\hspace{6cm}} \qquad \textit{if } x \neq y, y \notin FV(e')$$

$$[e'/x]\lambda y.\, e \quad = \quad \underline{\hspace{6cm}} \qquad \textit{if } x \neq y, y \in FV(e')$$

where ______________________________

____

____

____

Question 2. **[3 pts]** Complete the reduction rules for the call-by-value strategy. You may use the substitution which you defined in the previous question:

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

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

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

Question 3. **[3 pts]** Show the reduction sequence of a given expression under the call-by-name strategy. *Do not* rename bound variables.

call-by-name:

$$(\lambda t.\, \lambda f.\, f)\ ((\lambda x.\, x)\ (\lambda y.\, y))\ ((\lambda z.\, z)\ (\lambda w.\, w))$$

$\mapsto$ ______________________________

$\mapsto$ ______________________________

$\mapsto$ ______________________________

$\mapsto$ ______________________________

$\mapsto$ ______________________________

In Questions 4 and 5, you may use the following pre-defined constructs: zero, one, tt, ff, and, or, and pred. You do not need to copy definitions of these constructs.

- zero and one encode the natural numbers zero and one, respectively.

$$\begin{array}{rcl} \mathsf{zero} & = & \hat{0} = \lambda f.\, \lambda x.\, x \\ \mathsf{one} & = & \hat{1} = \lambda f.\, \lambda x.\, f\ x \end{array}$$

- tt and ff represent the boolean values true and false, respectively.

$$\begin{array}{rcl} \mathsf{tt} & = & \lambda t.\, \lambda f.\, t \\ \mathsf{ff} & = & \lambda t.\, \lambda f.\, f \end{array}$$

- and and or encode the boolean operators 'and' and 'or', respectively.

$$\begin{array}{rcl} \mathsf{and} & = & \lambda x.\, \lambda y.\, x\ y\ \mathsf{ff} \\ \mathsf{or} & = & \lambda x.\, \lambda y.\, x\ \mathsf{tt}\ y \end{array}$$

- pred computes the predecessor of a given natural number where the predecessor of 0 is 0.

$$\mathsf{pred} = \lambda \widehat{n}.\, \mathsf{fst}\ (\widehat{n}\ \mathsf{next}\ (\mathsf{pair}\ \mathsf{zero}\ \mathsf{zero}))$$

Question 4. [3 pts] Define the function exp for exponentiation such that $\mathsf{exp}\ \hat{m}\ \hat{n}$ evaluates to a church numeral for the product of n copies of m. In other words, $\mathsf{exp}\ \hat{m}\ \hat{n} \mapsto^* \widehat{m^n}$.

exp = ____

____

____

Question 5. [**3 pts**] Define the function $\mathsf{isZero} = \lambda \hat{n}. \cdots$ which tests if a given Church numeral is $\hat{0}$. That is, $\mathsf{isZero}\ \hat{0}$ reduces to tt, and $\mathsf{isZero}\ \hat{n}$ evaluates to ff for any non-zero number n.

$\mathsf{isZero}\ =$ ____

Question 6. [**3 pts**] Define the function $\mathsf{eq} = \lambda \hat{m}. \lambda \hat{n}. \cdots$ which tests if two given Church numerals are equal. You may use the function isZero.

$\mathsf{eq}\ =$ ____

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. [4 pts] 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.

$$\tau_i^n(N_1\ N_2) \quad = \quad \underline{\hspace{8cm}}$$

$$\tau_i^n(\lambda.\, N) \quad = \quad \underline{\hspace{8cm}}$$

$$\tau_i^n(m) \quad = \quad \underline{\hspace{8cm}} \qquad \text{if } m \geq i$$

$$\tau_i^n(m) \quad = \quad \underline{\hspace{8cm}} \qquad \text{if } m < i$$

Question 8. **[5 pts]** 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 $\tau_i^n(N)$.

$$\sigma_n(M_1\ M_2, N) \quad = \quad \underline{\hspace{8cm}}$$

$$\sigma_n(\lambda.\, M, N) \quad = \quad \underline{\hspace{8cm}}$$

$$\sigma_n(m, N) \quad = \quad \underline{\hspace{8cm}} \qquad \text{if } \ m < n$$

$$\sigma_n(n, N) \quad = \quad \underline{\hspace{8cm}}$$

$$\sigma_n(m, N) \quad = \quad \underline{\hspace{8cm}} \qquad \text{if } \ m > n$$

4 Simply typed λ-calculus [20 pts]

In this problem, we study the properties of the simply typed λ-calculus.

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

The reduction judgment and typing judgment are of the following forms:

$$\begin{array}{rcl}
e \mapsto e' & \Leftrightarrow & e \text{ reduces to } e' \\
\Gamma \vdash e : A & \Leftrightarrow & e \text{ has type } A \text{ under typing context } \Gamma
\end{array}$$

Question 1. [3 pts] Give the typing rules for the simply typed λ-calculus:

Question 2. [2 pts] State the canonical forms lemma, which is necessary to prove progress:

Question 3. [**2 pts**] State the inversion property, which is necessary to prove type preservation:

Question 4. [**4 pts**] State the two theorems, progress and type preservation, constituting type safety:

Theorem 4.1. *(Progress).*

Theorem 4.2. *(Preservation).*

Question 5. [3 pts] Consider the extension of the simply-typed λ-calculus with sum types:

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

Complete the typing rules.

$$\frac{}{\Gamma \vdash \textsf{inl}_A\ e :}$$

$$\frac{}{\Gamma \vdash \textsf{inr}_A\ e :}$$

$$\frac{}{\Gamma \vdash \textsf{case}\ e\ \textsf{of}\ \textsf{inl}\ x_1.\,e_1 \mid \textsf{inr}\ x_2.\,e_2 :}$$

Question 6. [2 pts] Consider the extension of the simply-typed λ-calculus with fixed point constructs

$$\text{expression} \quad e \quad ::= \quad \cdots \mid \mathsf{fix}\ x\!:\!A.\,e$$

Write the typing rule for $\mathsf{fix}\ x\!:\!A.\,e$ and its reduction rule.

$$\frac{\qquad\qquad\qquad\qquad}{\Gamma \vdash \mathsf{fix}\ x\!:\!A.\,e :}$$

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

Question 7. [4 pts] Explain how to encode two mutually recursive functions f_1 of type $A_1 \to B_1$ and f_2 of type $A_2 \to B_2$ using product types.

5 Mutable references [7 pts]

Consider the following simply-typed λ-calculus extended with mutable references.

$$\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \mid \textsf{int} \mid \textsf{ref}\ A \mid \textsf{unit} \\
\textsf{base type} & P & ::= & \textsf{bool} \\
\textsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \mid \\
 & & & \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \\
 & & & \textsf{ref}\ e \mid\ !e \mid e := e \mid () \mid \\
 & & & + \mid - \mid * \mid \div \mid = \mid \\
 & & & 0 \mid 1 \mid \cdots \\
\textsf{value} & v & ::= & \lambda x : A.\, e \mid () \mid \textsf{true} \mid \textsf{false} \mid 0 \mid 1 \mid \cdots
\end{array}$$

Question 1. **[3 pts]** We want to represent an array of integers as a function taking an index (of type int) and returning a corresponding element of the array. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$\textsf{iarray} = \textsf{ref}\ (\textsf{int} \to \textsf{int})$$

We need the following constructs for arrays:

- new : unit $\to$ iarray for creating a new array.
  new () returns a new array of indefinite size; all elements are initialized as 0.
- access : iarray $\to$ int $\to$ int for accessing an array.
  access a i returns the i-th element of array a.
- update : iarray $\to$ int $\to$ int $\to$ unit for updating an array.
  update a i n updates the i-th element of array a with integer n.

Exploit the constructs for mutable references to implement new, access and update. Fill in the blank:

$$\textsf{new} = \lambda\_ : \textsf{unit}.\ \textsf{ref}\ \lambda i : \textsf{int}.\, 0$$

$$\textsf{access} = \lambda a : \textsf{iarray}.\, \lambda i : \textsf{int}.\, (!a)\ i$$

$$\textsf{update} = \lambda a : \textsf{iarray}.\, \lambda i : \textsf{int}.\, \lambda n : \textsf{int}.$$

____

____

____

____

Question 2. **[4 pts]** Use the constructs for mutable references to implement a recursive function fact for factorials such that fact n evaluates to $n!$.

fact $=$ ________________________________

6 Symmetry of the α-equivalence relation [15 pts]

In this problem, we prove the symmetry of the α-equivalence relation in the untyped λ-calculus (Theorem 6.4). We use the following inference rules, where $FV(e)$ computes the set of free variables in e and $[x \leftrightarrow y]e$ denotes the expression obtained by replacing all occurrences of x in e by y and all occurrences of y in e by x.

$$\frac{}{x \equiv_\alpha x}\ Var_\alpha \qquad \frac{e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2'}{e_1\ e_2 \equiv_\alpha e_1'\ e_2'}\ App_\alpha$$

$$\frac{e \equiv_\alpha e'}{\lambda x.\, e \equiv_\alpha \lambda x.\, e'}\ Lam_\alpha \qquad \frac{x \neq y \quad y \notin FV(e) \quad [x \leftrightarrow y]e \equiv_\alpha e'}{\lambda x.\, e \equiv_\alpha \lambda y.\, e'}\ Lam_\alpha'$$

In the proof of Theorem 6.4, you may use the following lemmas on the α-equivalence relation without proofs:

Lemma 6.1. $[x \leftrightarrow y][x \leftrightarrow y]e = [y \leftrightarrow x][x \leftrightarrow y]e = e$.

Lemma 6.2. *If* $e_1 \equiv_\alpha e_2$, *then* $[x \leftrightarrow y]e_1 \equiv_\alpha [x \leftrightarrow y]e_2$.

Lemma 6.3. *If* $e_1 \equiv_\alpha e_2$, *then* $FV(e_1) = FV(e_2)$.

Complete the proof of Theorem 6.4.

Theorem 6.4. *If* $e_1 \equiv_\alpha e_2$, *then* $e_2 \equiv_\alpha e_1$

Proof.

$\square$

7 (Extra-credit) Transitivity of the α-equivalence relation

In this problem, we prove the transitivity of the α-equivalence relation from the previous problem:

Theorem 7.1. *If* $e_1 \equiv_\alpha e_2$ *and* $e_2 \equiv_\alpha e_3$*, then* $e_1 \equiv_\alpha e_3$*.*

In your proof, you may use the following lemmas without proofs (Lemmas 7.2 to 7.5). Lemma 7.2 shows how two variable swappings $[p \leftrightarrow q]$ and $[x \leftrightarrow y]$ commute. Lemma 7.3 shows how a variable swapping affects the set of free variables in a given expression. Lemma 7.4 states the symmetry of the α-equivalence relation. Lemma 7.5 is the inversion property of the α-equivalence relation and holds because the inference rules for the α-equivalence relation are syntax-directed.

Lemma 7.2. $[p \leftrightarrow q][x \leftrightarrow y]e = [[p \leftrightarrow q]x \leftrightarrow [p \leftrightarrow q]y][p \leftrightarrow q]e$.

Lemma 7.3.
$x \notin FV(e)$ *if and only if* $[p \leftrightarrow q]x \notin FV([p \leftrightarrow q]e)$*.*
$x \in FV(e)$ *if and only if* $[p \leftrightarrow q]x \in FV([p \leftrightarrow q]e)$*.*

Lemma 7.4. *If* $e_1 \equiv_\alpha e_2$*, then* $e_2 \equiv_\alpha e_1$*.*

Lemma 7.5 (Inversion)**.**
If $x \equiv_\alpha e$*, then* $e = x$*.*
If $e'_1\ e''_1 \equiv_\alpha e$*, then* $e = e'_2\ e''_2$*,* $e'_1 \equiv_\alpha e'_2$*, and* $e''_1 \equiv_\alpha e''_2$ *for some* e'_2 *and* e''_2*.*
If $\lambda x.\, e_1 \equiv_\alpha \lambda x.\, e_2$*, then* $e_1 \equiv_\alpha e_2$*.*
If $\lambda x.\, e_1 \equiv_\alpha \lambda y.\, e_2$ *and* $x \neq y$*, then* $y \notin FV(e_1)$*, and* $[x \leftrightarrow y]e_1 \equiv_\alpha e_2$*.*

Prove Theorem 7.1.

Hint: Perhaps the proof should proceeed by rule induction on $e_1 \equiv_\alpha e_2$. You will need a lemma that shows how variable swappings affect the α-equivalence relation. Identifying this lemma is critical to the proof of Theorem 7.1. If you introduce such a lemma, you should give a proof of it as well.