Name: Hemos ID:

CSE-321 Programming Languages 2010
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score					
Max	15	30	35	20	100

1 SML Programming [15 pts]

Question 1. [**5 pts**] Give a tail recursive implementation of `preorder` for preorder traversals of binary trees. Fill in the blank:

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

(* preorder : 'a tree -> 'a list *)
fun preorder t =
let

  ______________________________________________

  ______________________________________________

  ______________________________________________

  ______________________________________________
in

  ______________________________________________
end
```

Question 2. [**7 pts**] `DICT` is a signature for dictionaries:

```
signature DICT =
sig
  type key
  type 'a dict
  val empty : unit -> 'a dict
  val lookup : 'a dict -> key -> 'a option
  val delete : 'a dict -> key -> 'a dict
  val insert : 'a dict -> key * 'a -> 'a dict
end
```

- `key` denotes the type of keys in dictionaries.
- `'a dict` denotes the type of dictionaries for `'a` type values.
- `empty ()` returns an empty dictionary.
- `lookup d k` searches the key `k` in the dictionary `d`. If the key is found, it returns the associated item. Otherwise, it returns NONE.
- `delete d k` deletes the key `k` and its associated item in the dictionary `d` and returns the resultant dictionary `d'`. If the key does not exist in the dictionary `d`, it returns the given dictionary `d` without any modification.
- `insert d (k, v)` inserts the new key `k` and its associated item `v` in the dictionary `d`. If the key `k` already exists in the dictionary `d`, it just updates its associated item with the given item `v`.

Implement the functor `DictFn` which takes a `KEY` structure and generates a corresponding `DICT` structure that uses a 'functional representation' of dictionaries. Fill in the blank:

```
signature KEY =
sig
  type t

  (* eq k k' : true   k is equal to k'  *)
  (*           false  otherwise         *)
  val eq : t * t -> bool
end

functor DictFn (Key : KEY) :> DICT where type key = Key.t
=
struct
  type key = Key.t
  type 'a dict = key -> 'a option

  fun empty () =

      ________________________________________

  fun lookup d k =

      ________________________________________

  fun delete d k =

      ________________________________________

  fun insert d (k, v) =

      ________________________________________
end
```

Question 3. [**3 pts**] Give an implementation of `IntDict` whose `key` type is `int`. Fill in the blank. You may use the functor `DictFn` that you write in Question 2.

```
structure IntKey :> KEY where type t = int
=
struct

  ____________________________________________

  ____________________________________________

  ____________________________________________

  ____________________________________________

  ____________________________________________
end

structure IntDict = ______________________________
```

2 Inductive definitions [30 pts]

Consider the following system from the Course Notes where s mparen means that s is a string of matched parentheses.

$$\frac{}{\epsilon\ \mathsf{mparen}}\ \mathit{Meps} \qquad \frac{s\ \mathsf{mparen}}{(s)\ \mathsf{mparen}}\ \mathit{Mpar} \qquad \frac{s_1\ \mathsf{mparen} \quad s_2\ \mathsf{mparen}}{s_1\ s_2\ \mathsf{mparen}}\ \mathit{Mseq}$$

In order to show that if s mparen holds, s is indeed a string of matched parentheses, we introduce a new judgment $k \rhd s$ where k is a non-negative integer:

$$\begin{array}{lcl} k \rhd s & \Leftrightarrow & k \textit{ left parentheses concatenated with } s \textit{ form a string of matched parentheses} \\ & \Leftrightarrow & \underbrace{((\cdots(}_{k}\, s \textit{ is a string of matched parentheses} \end{array}$$

The idea is that we scan a given string from left to right and keep counting the number of left parentheses that have not yet been matched with corresponding right parentheses. Thus we begin with $k = 0$, increment k each time a left parenthesis is encountered, and decrement k each time a right parenthesis is encountered:

$$\frac{}{0 \rhd \epsilon}\ \mathit{Peps} \qquad \frac{k+1 \rhd s}{k \rhd (s}\ \mathit{Pleft} \qquad \frac{k-1 \rhd s \quad k > 0}{k \rhd)s}\ \mathit{Pright}$$

The second premise $k > 0$ in the rule *Pright* ensures that in any prefix of a given string, the number of right parentheses may not exceed the number of left parentheses. Now a judgment $0 \rhd s$ expresses that s is a string of matched parentheses.

Your task is to prove Theorem 2.1. If you need a lemma to complete the proof, state the lemma, prove it, and use it to complete the proof of Theorem 2.1.

For individual steps in the proof, please use the following format:

conclusion *justification*

Theorem 2.1. *If* s mparen*, then* $0 \rhd s$*.*

3 λ-Calculus [35 pts]

Question 1. **[5 pts]** 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 2. **[3 pts]** Complete the definition of $FV(e)$ that finds the set of free variables in e.

$$FV(x) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV(\lambda x.\, e) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV(e_1\ e_2) = \underline{\qquad\qquad\qquad\qquad}$$

Question 3. **[2 pts]** Fill in the blank with the set of free variables of the given expression.

$$FV(\lambda x.\, x) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV(x\ y) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV(\lambda x.\, x\ y) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV(\lambda x.\, \lambda y.\, x\ y) = \underline{\qquad\qquad\qquad\qquad}$$

$$FV((\lambda x.\, x\ y)\ (\lambda y.\, x\ y)) = \underline{\qquad\qquad\qquad\qquad}$$

Question 4. [5 pts] This question assumes types `var` and `exp` that we have seen in Assignment 4:

```
type var = string
datatype exp =
  Var of var
| Lam of var * exp
| App of exp * exp
```

Suppose that we have two functions `notFv` and `varSwap`:

- `notFv : var -> exp -> bool`
  `notFv` x e returns `true` if x is a free variable of e and `false` otherwise.

- `varSwap : var * var -> exp -> exp`
  `varSwap` (x, y) e returns $[x \leftrightarrow y]e$.

Below is a function `aEqual` of type `( exp * exp ) -> bool` such that `aEqual` (e_1, e_2) returns `true` if e_1 and e_2 are α-equivalent and `false` otherwise.

```
fun aEqual (Var x, Var y) = x = y
  | aEqual (App (e1, e2), App (e1', e2')) =
    aEqual (e1, e1') andalso aEqual (e2, e2')
  | aEqual (Lam (x, e), Lam (y, e')) =
    if x = y then aEqual (e, e')
    else if notFv x e' then aEqual (e, varSwap (y, x) e')
    else false
  | aEqual _ = false
```

We write $e \equiv_\alpha e'$ if e can be rewritten as e' by renaming bound variables in e and vice versa. Give exactly <u>four</u> inference rules corresponding to the above definition of `aEqual`. Use the notation $x \notin FV(e)$ for `notFv` x e and $[x \leftrightarrow y]e$ for `varSwap` (x, y) e.

$$\overline{\qquad\qquad\qquad \equiv_\alpha \qquad\qquad\qquad}$$

$$\overline{\qquad\qquad\qquad \equiv_\alpha \qquad\qquad\qquad}$$

$$\overline{\qquad\qquad\qquad \equiv_\alpha \qquad\qquad\qquad}$$

$$\overline{\qquad\qquad\qquad \equiv_\alpha \qquad\qquad\qquad}$$

Question 5. **[8 pts]** 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):

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

In this question, you will define three functions: sub for the subtraction operation, mod for the modulo operation, and, as an extra credit problem, div for the division operation.

Your answers may use the following pre-defined constructs: zero, one, succ, $\mathsf{if/then/else}$, $\mathsf{pair/fst/snd}$, pred, eq, and fix.

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

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

- succ finds the successor of a given natural number.

$$\mathsf{succ} \;=\; \lambda \widehat{n}.\, \lambda f.\, \lambda x.\, \widehat{n}\ f\ (f\ x)$$

- $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ is a conditional construct.

$$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \;=\; e\ e_1\ e_2$$

- pair creates a pair of two expressions, and fst and snd are projection operators.

$$\begin{array}{rcl} \mathsf{pair} &=& \lambda x.\, \lambda y.\, \lambda b.\, b\ x\ y \\ \mathsf{fst} &=& \lambda p.\, p\ (\lambda t.\, \lambda f.\, t) \\ \mathsf{snd} &=& \lambda p.\, p\ (\lambda t.\, \lambda f.\, f) \end{array}$$

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

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

- eq tests two natural numbers for equality.

$$\mathsf{eq} \;=\; \lambda x.\, \lambda y.\, \mathsf{and}\ (\mathsf{isZero}\ (x\ \mathsf{pred}\ y))\ (\mathsf{isZero}\ (y\ \mathsf{pred}\ x))$$

- fix is the fixed point combinator.

$$\mathsf{fix} \;=\; \lambda F.\, (\lambda f.\, F\ \lambda x.\, (f\ f\ x))\ (\lambda f.\, F\ \lambda x.\, (f\ f\ x))$$

These constructs use the following auxiliary constructs, which you do not need:

$$\begin{array}{rcl} \mathsf{tt} &=& \lambda t.\, \lambda f.\, t \\ \mathsf{ff} &=& \lambda t.\, \lambda f.\, f \\ \mathsf{and} &=& \lambda x.\, \lambda y.\, x\ y\ \mathsf{ff} \\ \mathsf{isZero} &=& \lambda x.\, x\ (\lambda y.\, \mathsf{ff})\ \mathsf{tt} \\ \mathsf{next} &=& \lambda p.\, \mathsf{pair}\ (\mathsf{snd}\ p)\ (\mathsf{succ}\ (\mathsf{snd}\ p)) \end{array}$$

Define a subtraction function sub such that sub $\hat{m}$ $\hat{n}$ evaluates to $\widehat{m-n}$ if $m > n$ and $\hat{0}$ otherwise.

sub $=$ ____

Define a modulo function mod such that mod $\hat{m}$ $\hat{n}$ evaluates to $\hat{r}$ if r is the remainder of division of m by n. mod never takes $\hat{0}$ as the second argument. Hence the result of evaluating mod $\hat{m}$ $\hat{0}$ is unspecified. You may use the subtraction function sub that you define above.

mod $=$

____

Extra credit question. [10 pts] Define a division function div such that div $\hat{m}$ $\hat{n}$ evaluates to $\hat{q}$ if q is the quotient of m divided by n. div never takes $\hat{0}$ as the second argument. Hence the result of evaluating div $\hat{m}$ $\hat{0}$ is unspecified. In this question, you are not allowed to use the fixed point combinator (and its definition), but you may use the subtraction function sub that you define above.

div $=$

____

Question 6. **[3 pts]** Give an expression whose reduction does not terminate.

____

Question 7. **[5 pts]** Following is the definition of de Bruijn expressions:

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

Suppose that you are given the definition of $\tau_i^n(N)$ 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.

Complete the definition of $\sigma_n(M, N)$ for substituting N for every occurrence of n in M where N may include free variables.

$$\begin{array}{rcll} \sigma_n(M_1\ M_2, N) & = & \underline{\hspace{6cm}} & \\ \sigma_n(\lambda.\, M, N) & = & \underline{\hspace{5cm}} & \\ \sigma_n(m, N) & = & \underline{\hspace{3cm}} & \text{if } m < n \\ \sigma_n(n, N) & = & \underline{\hspace{4cm}} & \\ \sigma_n(m, N) & = & \underline{\hspace{4cm}} & \text{if } m > n \end{array}$$

Question 8. **[4 pts]** Show the reduction of the given expression where the redex is underlined.

$$\lambda.\,\lambda.\,\underline{(\lambda.\,(\lambda.\,3\ 2\ 1\ 0)\ (2\ 1\ 0))\ (\lambda.\,0)} \quad \mapsto$$

$$\underline{(\lambda.\,(\lambda.\,1)\ 0)\ (\lambda.\,2\ 1\ 0)} \quad \mapsto$$

4 Simply-typed λ-calculus [20 pts]

Question 1. [**2 pts**] Consider the following simply-typed λ-calculus:

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

Write the typing rules for x, $\lambda x\!:\!A.\, e$, and $e\ e$:

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\Gamma \vdash x :}$$

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

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\Gamma \vdash e\ e' :}$$

Question 2. [**3 pts**] Consider the extension of the simply-typed λ-calculus with product types:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \cdots \mid A \times A \\ \textsf{expression} & e & ::= & \cdots \mid (e, e) \mid \textsf{fst}\ e \mid \textsf{snd}\ e \end{array}$$

Write the reduction rules for these constructs under *lazy* reduction strategy:

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

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

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

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

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

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

Write the typing rules:

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

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

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

Question 4. [5 pts] Consider the extension of the simply-typed λ-calculus with fixed point constructs

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

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

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

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

Question 5. [**5 pts**] Consider the following SML program:

```
fun even 0 = true
  | even 1 = false
  | even n = odd (n - 1)

and odd 0 = false
  | odd 1 = true
  | odd n = even (n - 1)
```

The function `even` calls the function `odd`, and the function `odd` calls the function `even`. We refer to these functions as mutually recursive functions.

Write an expression of type $(\mathsf{int} \rightarrow \mathsf{bool}) \times (\mathsf{int} \rightarrow \mathsf{bool})$ that encodes both `even` and `odd` in the simply-typed λ-calculus:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & \mathsf{int} \mid \mathsf{bool} \mid A \rightarrow A \mid A \times A \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid () \mid \\
& & & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{fix}\ x\!:\!A.\, e \\
& & & - \mid\ = \ \mid 0 \mid 1 \mid \cdots
\end{array}
$$

We assume that the infix operations $-$ and $=$ are given as primitive, which correspond to the integer substitution and equality test, respectively.