Name: Hemos ID:

CSE-321 Programming Languages 2006
Midterm — Sample Solution

	Problem 1	Problem 2	Problem 3	Problem 4	Problem 5	Problem 6	Problem 7	Total
Score								
Max	20	15	15	15	15	10	10	100

1 SML Programming [20 pts]

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

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

(* inorder : 'a tree -> 'a list *)
fun inorder t =
let
  (* inorder' : 'a tree -> 'a list -> 'a list *)
  fun inorder' (Leaf x) post =

    x :: post
  | inorder' (Node (left, x, right)) post =

    inorder' left (x :: (inorder' right post))
in
  inorder' t []
end
```

Question 2. **[5 pts]** Rewrite expressions in the left column *without* using the `if/then/else` construct. You may use `not`, `andalso`, and `orelse`.

with `if/then/else`	without `if/then/else`
`if e then false else true`	`not e`
`if x then true else y`	`x orelse y`
`if x then y else false`	`x andalso y`
`if x then false else y`	`not x andalso y`
`if x then y else true`	`not x orelse y`

Question 3. **[10 pts]** A signature `SET` for sets is given as follows:

```
signature SET =
sig
    type 'a set
    val empty : ''a set
    val singleton : ''a -> ''a set
    val member : ''a set -> ''a -> bool
    val insert : ''a set -> ''a -> ''a set
    val remove : ''a set -> ''a -> ''a set
    val union : ''a set -> '' a set -> ''a set
end
```

- `empty` is an empty set.
- `singleton` x returns a singleton set consisting of x.
- `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.
- `remove` s x removes x from the set s and returns the resultant set. If x is not a member of s, then `remove` s x returns s.
- `union` s s' returns the union of s and s'.

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; instead take advantage of the result from Question 2. Fill in the blank:

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

      val empty = fn _ => false

      fun singleton x = fn y => x = y

      fun member s = s

      fun insert s x = fn y => x = y orelse s y

      fun remove s x = fn y => x <> y andalso s y

      fun union s s' = fn x => s x orelse s' x
   end
```

2 Reductions in the λ-calculus [15 pts]

Let us abbreviate an identity function $\lambda x_i.\, x_i$ as id_i. You will show the reduction sequence of $(\underline{\mathsf{id}_1\ \mathsf{id}_2})\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z))$ under the call-by-name strategy (Question 1) and under the call-by-value strategy (Question 2).

Question 1. [5 pts] Show the reduction sequence under the call-by-name strategy. Underline the redex at each step. Do not expand id_i back to $\lambda x_i.\, x_i$.

$$
\begin{aligned}
& (\underline{\mathsf{id}_1\ \mathsf{id}_2})\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto\ & \underline{\mathsf{id}_2\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z))} \\
\mapsto\ & \underline{\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)} \\
\mapsto\ & \lambda z.\, \mathsf{id}_4\ z
\end{aligned}
$$

Question 2. [5 pts] Show the reduction sequence under the call-by-value strategy. Underline the redex at each step. Do not expand id_i back to $\lambda x_i.\, x_i$.

$$
\begin{aligned}
& (\underline{\mathsf{id}_1\ \mathsf{id}_2})\ (\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)) \\
\mapsto\ & \mathsf{id}_2\ (\underline{\mathsf{id}_3\ (\lambda z.\, \mathsf{id}_4\ z)}) \\
\mapsto\ & \underline{\mathsf{id}_2\ (\lambda z.\, \mathsf{id}_4\ z)} \\
\mapsto\ & (\lambda z.\, \mathsf{id}_4\ z)
\end{aligned}
$$

Question 3. [5 pts] Fill in the blank with the result of applying α-conversion to the expression in the left. We have supplied a variable to be used in the conversion. If it is impossible to apply α-conversion using the given variable, write "impossible."

- (example) $\lambda x.\, x \equiv_\alpha \lambda y.\, \underline{y}$

$$
\begin{aligned}
\lambda x.\, \lambda x'.\, x\ x' &\equiv_\alpha \lambda x'.\, \underline{\lambda x.\, x'\ x \qquad\qquad\qquad} \\
\lambda x.\, \lambda x'.\, x\ x'\ x'' &\equiv_\alpha \lambda x'.\, \underline{\lambda x.\, x'\ x\ x'' \qquad\qquad\qquad} \\
\lambda x.\, \lambda x'.\, x\ x'\ x'' &\equiv_\alpha \lambda x''.\, \underline{\text{impossible} \qquad\qquad\qquad}
\end{aligned}
$$

3 Programming in the λ-calculus [15 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):

$$\begin{array}{rcccl} \hat{0} & = & \lambda f.\, f^0 & = & \lambda f.\, \lambda x.\, x \\ \hat{1} & = & \lambda f.\, f^1 & = & \lambda f.\, \lambda x.\, f\ x \\ & & \cdots & & \\ \hat{n} & = & \lambda f.\, f^n & = & \lambda f.\, \lambda x.\, f\ f\ f\ \cdots f\ x \end{array}$$

Question 1. **[5 pts]** Define an operation double for doubling a given natural number. Specifically $\mathsf{double}\ \widehat{n}$ returns $\widehat{2 * n}$. Fill in the blank:

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

Question 2. **[10 pts]** Define an operation halve for halving a given natural number. Specifically $\mathsf{halve}\ \widehat{n}$ returns $\widehat{n/2}$:

- $\mathsf{halve}\ \widehat{2 * k}$ returns $\widehat{k}$.
- $\mathsf{halve}\ \widehat{2 * k + 1}$ returns $\widehat{k}$.

For defining halve, you want to exploit the encoding of pairs in the Course Notes:

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

Use pair, fst, and snd without expanding them into the above definition. To make your answer more readable, you also want to use zero for a natural number zero and succ for finding the successor to a given natural number:

$$\begin{array}{rcl} \mathsf{zero} & = & \hat{0} \;=\; \lambda f.\, \lambda x.\, x \\ \mathsf{succ} & = & \lambda \widehat{n}.\, \lambda f.\, \lambda x.\, \widehat{n}\ f\ (f\ x) \end{array}$$

Fill in the blank:

$$\mathsf{halve} \;=\; \lambda \widehat{n}.\, \underline{\mathsf{fst}\ (\widehat{n}\ (\lambda p.\, \mathsf{pair}\ (\mathsf{snd}\ p)\ (\mathsf{succ}\ (\mathsf{fst}\ p)))\ (\mathsf{pair}\ \mathsf{zero}\ \mathsf{zero}))}$$

4 A weird reduction strategy [15 pts]

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

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \rightarrow A \\
\textsf{base type} & P & & \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\,e
\end{array}
$$

We will develop a weird strategy specified as follows:

- Given an application $e_1\ e_2$, we first reduce e_2.
- After reducing e_2 to a value, we reduce e_1.
- When e_1 reduces to a λ-abstraction, we apply the β-reduction.

Question 1. [5 pts] Give the rules for the reduction judgment $e \mapsto e'$ under the weird reduction strategy. You need three rules.

$$
\frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'} \qquad \frac{e_1 \mapsto e_1'}{e_1\ v \mapsto e_1'\ v} \qquad \frac{}{(\lambda x\!:\!A.\,e)\ v \mapsto [v/x]e}
$$

Question 2. [5 pts] Give the rules for the evaluation judgment $e \hookrightarrow v$ under the weird reduction strategy. You need two rules.

$$
\frac{}{\lambda x\!:\!A.\,e \hookrightarrow \lambda x\!:\!A.\,e} \qquad \frac{e_2 \hookrightarrow v_2 \quad e_1 \hookrightarrow \lambda x\!:\!A.\,e \quad [v_2/x]e \hookrightarrow v}{e_1\ e_2 \hookrightarrow v}
$$

Question 3. [5 pts] Give the definition of evaluation contexts corresponding to the weird reduction strategy:

$$
\textsf{evaluation context} \qquad \kappa \ ::= \ \square \mid e\ \kappa \mid \kappa\ v
$$

5 Substitution theorem [15 pts]

Prove the substitution theorem for the following fragment of the simply typed λ-calculus:

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{Var} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x\!:\!A.\, e : A \to B}\ {\to}\textsf{I} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}\ {\to}\textsf{E}$$

Theorem (Substitution). *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$*, then* $\Gamma \vdash [e/x]e' : C$*.*

Proof. By rule induction on the judgment $\Gamma, x : A \vdash e' : C$. We assume that all variables in a typing context are distinct. We also assume that variable clashes never occur in the rule $\to$I. That is, x in the rule $\to$I is always a fresh variable.

Fill in the blank:

Case $\dfrac{y : C \in \Gamma}{\Gamma, x : A \vdash y : C}\ \textsf{Var}$ where $e' = y$

$\Gamma \vdash y : C$ from $y : C \in \Gamma$ and the rule Var

$[e/x]y = y$ from $x \neq y$

$\Gamma \vdash [e/x]y : C$

Case $\dfrac{}{\Gamma, x : A \vdash x : A}\ \textsf{Var}$ where $e' = x$ and $C = A$

$\Gamma \vdash e : A$ from the assumption

$[e/x]x = e$

$\Gamma \vdash [e/x]x : A$

Case $\dfrac{\Gamma, x : A, y : B_1 \vdash e'' : B_2}{\Gamma, x : A \vdash \lambda y\!:\!B_1.\, e'' : B_1 \to B_2}\ {\to}\textsf{I}$ where $e' = \lambda y\!:\!B_1.\, e''$ and $C = B_1 \to B_2$

$\Gamma, y : B_1 \vdash [e/x]e'' : B_2$ by IH on the premise

$\Gamma \vdash \lambda y\!:\!B_1.\, [e/x]e'' : B_1 \to B_2$ by the rule $\to$I

$[e/x]\lambda y\!:\!B_1.\, e'' = \lambda y\!:\!B_1.\, [e/x]e''$ from $y \notin FV(e)$ and $x \neq y$

$\Gamma \vdash [e/x]\lambda y\!:\!B_1.\, e'' : B_1 \to B_2$

Case $\dfrac{\Gamma, x : A \vdash e_1 : B \to C \quad \Gamma, x : A \vdash e_2 : B}{\Gamma, x : A \vdash e_1\ e_2 : C} \to\mathsf{E}$ where $e' = e_1\ e_2$

$\underline{\Gamma \vdash [e/x]e_1 : B \to C}$ by IH on the first premise

$\underline{\Gamma \vdash [e/x]e_2 : B}$ by IH on the second premise

$\Gamma \vdash [e/x]e_1\ [e/x]e_2 : C$ by the rule $\to\mathsf{E}$

$\underline{\Gamma \vdash [e/x](e_1\ e_2) : C}$ by the definition of substitution

$\square$

6 Transitivity [10 pts]

In a reduction sequence judgment $e \mapsto^* e'$, we use $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$. That is, $e \mapsto^* e'$ holds if $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ where $n \geq 0$. Then we would expect that $e \mapsto^* e'$ and $e' \mapsto^* e''$ together imply $e \mapsto^* e''$, since we obtain a proof of $e \mapsto^* e''$ simply by concatenating $e \mapsto e_1 \mapsto \cdots \mapsto e_n = e'$ and $e' \mapsto e'_1 \mapsto \cdots \mapsto e'_m = e''$:

$$e \mapsto e_1 \mapsto \cdots \mapsto e_n = e' \mapsto e'_1 \mapsto \cdots \mapsto e'_m = e''$$

You will prove this transitivity property of $\mapsto^*$ under the following inductive definition:

$$\frac{}{e \mapsto^* e}\ \mathit{Refl} \qquad \frac{e \mapsto e'' \quad e'' \mapsto^* e'}{e \mapsto^* e'}\ \mathit{Trans}$$

Theorem (Transitivity). *If $e \mapsto^* e'$ and $e' \mapsto^* e''$, then $e \mapsto^* e''$.*

Fill in the blank below and complete the proof:

Proof. By rule induction on the judgment $\underline{e \mapsto^* e'}$.

Case $\dfrac{}{e \mapsto^* e}\ \langle \mathit{Refl} \rangle$ where $\underline{e' = e}$

$\underline{e' \mapsto^* e''}$ — assumption

$\underline{e \mapsto^* e''}$ — from $\underline{e' \mapsto^* e'' \text{ and } e' = e}$

Case $\dfrac{e \mapsto e''' \quad e''' \mapsto^* e'}{e \mapsto^* e'}\ \langle \mathit{Trans} \rangle$

$\underline{e' \mapsto^* e''}$ — assumption

$\underline{e''' \mapsto^* e''}$ — by $\underline{\text{induction hypothesis on } e''' \mapsto^* e' \text{ with } e' \mapsto^* e''}$

$\underline{e \mapsto^* e''}$ — from $\dfrac{e \mapsto e''' \quad e''' \mapsto^* e''}{e \mapsto^* e''}\ \mathit{Trans}$

□

7 Abstract machine C [10 pts]

Consider the following fragment of the simply typed λ-calculus for the call-by-value strategy:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \\
\textsf{base type} & P & & \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{fix}\ x\!:\!A.\, e \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\, e \mid \\
\textsf{frame} & \phi & ::= & \Box\ e \mid v\ \Box \mid \\
\textsf{stack} & \sigma & ::= & \Box \mid \sigma; \phi \\
\textsf{state} & s & ::= & \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v
\end{array}
$$

The goal of this problem is to write the rules for the state transition judgment $s \mapsto_{\mathsf{C}} s'$ for the abstract machine C. For your reference, we give the rules for the reduction judgment $e \mapsto e'$ below:

$$
\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \qquad \frac{e_2 \mapsto e_2'}{v\ e_2 \mapsto v\ e_2'}\ \mathit{Arg} \qquad \frac{}{(\lambda x\!:\!A.\, e)\ v \mapsto [v/x]e}\ \mathit{App} \qquad \frac{}{\textsf{fix}\ x\!:\!A.\, e \mapsto [\textsf{fix}\ x\!:\!A.\, e/x]e}\ \mathit{Fix}
$$

Fill in the blank and complete each rule:

$$
\frac{}{\sigma \blacktriangleright v \mapsto_{\mathsf{C}} \sigma \blacktriangleleft v}\ \mathit{Val}_{\mathsf{C}}
$$

$$
\frac{}{\sigma \blacktriangleright e_1\ e_2 \mapsto_{\mathsf{C}} \sigma; \Box\ e_2 \blacktriangleright e_1}\ \mathit{Lam}_{\mathsf{C}}
$$

$$
\frac{}{\sigma; \Box\ e_2 \blacktriangleleft v \mapsto_{\mathsf{C}} \sigma; v\ \Box \blacktriangleright e_2}\ \mathit{Arg}_{\mathsf{C}}
$$

$$
\frac{}{\sigma; (\lambda x\!:\!A.\, e)\ \Box \blacktriangleleft v \mapsto_{\mathsf{C}} \sigma \blacktriangleright [v/x]e}\ \mathit{App}_{\mathsf{C}}
$$

$$
\frac{}{\sigma \blacktriangleright \textsf{fix}\ x\!:\!A.\, e \mapsto_{\mathsf{C}} \sigma \blacktriangleright [\textsf{fix}\ x\!:\!A.\, e/x]e}\ \mathit{Fix}_{\mathsf{C}}
$$