

Name:

Hemos ID:

CSE-321 Programming Languages 2007  
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Prob 7	Total
Score								
Max	13	11	26	5	10	20	15	100

## 1 SML Programming [13 pts]

**Question 1.** [3 pts] Give a tail-recursive implementation of `fact` for the factorial function. Perhaps you will need two lines of code.

```
fun fact n =  
  let  
  
  _____  
  
  _____  
  
  in  
    fact' n 1  
  end
```

**Question 2.** [5 pts] Exploit mutable references in SML to implement a factorial function of type `int -> int`. You may use the `fn` keyword, but not the `fun` keyword. That is, do not use the built-in mechanism for building recursive functions in SML. Your program should evaluate to a factorial function that returns  $n!$  if its argument  $n$  is positive, *i.e.*,  $n > 0$ . Perhaps you will need three or four lines of code.

```
let
```

```
  _____  
  
  _____
```

```
in
```

```
  _____  
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 singleton : ''a -> ''a set
  val union : ''a set -> '' a set -> ''a set
  val intersection : ''a set -> '' a set -> ''a set
  val diff : ''a set -> '' a set -> ''a set
end
```

- `empty` is an empty set.
- `singleton x` returns a singleton set consisting of  $x$ .
- `union s s'` returns the union of  $s$  and  $s'$ .
- `intersection s s'` returns the intersection of  $s$  and  $s'$ .
- `diff s s'` returns the difference of  $s$  and  $s'$ : the set of elements in  $s$  but not in  $s'$ .

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

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

    val empty = _____

    fun singleton x = _____

    fun union s s' = _____

    fun intersection s s' = _____

    fun diff s s' = _____
  end
```

## 2 True/false questions [11 pts]

For true/false questions, a wrong answer gives a penalty equal to the points assigned to the question. Given an answer only if you are convinced!

**Question 1. [1 pts]** A derivable rule is always admissible. True or false?

\_\_\_\_\_

**Question 2. [1 pts]** When reducing a closed expression, we may need to use  $\alpha$ -conversions. True or false?

\_\_\_\_\_

**Question 3. [1 pts]** We can prove  $\lambda x. e \equiv_\alpha \lambda y. e'$  when  $x \neq y$  and  $y \in FV(e)$  where  $FV(e)$  calculates the set of free variables in  $e$ . True or false?

\_\_\_\_\_

**Question 4. [1 pts]** Given a function  $f$  in the untyped  $\lambda$ -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 5. [1 pts]** In the presence of an abort expression  $\text{abort}_A e$ , type safety of the simply typed  $\lambda$ -calculus continues to hold. True or false?

\_\_\_\_\_

**Question 6. [2 pts]** The fixed point construct  $\text{fix } x:A. e$  makes every type inhabited in the simply typed  $\lambda$ -calculus. True or false?

\_\_\_\_\_

**Question 7. [1 pts]** If an algorithmic typing judgment covers all possible cases of well-typed expressions, it is said to be “sound.” True or false?

\_\_\_\_\_

**Question 8. [3 pts]** If a language has no static type system, it cannot be a safe language. True or false?

\_\_\_\_\_

### 3 Short answers [26 pts]

**Question 1.** [4 pts] Show the reduction sequence of the following expression under the call-by-name strategy.

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

$\mapsto$  \_\_\_\_\_  
 $\mapsto$  \_\_\_\_\_  
 $\mapsto$  \_\_\_\_\_  
 $\mapsto$  \_\_\_\_\_

**Question 2.** [2 pts] Suppose that  $v_1$ ,  $v_2$ , and  $v_3$  are all values of type  $A$  in the simply typed  $\lambda$ -calculus. Assuming the *lazy* reduction strategy, how many steps does it take to fully reduce to a value the following expression?

$$\text{fst} \left( (\lambda x : A \times A. \text{fst } x) ((\lambda x : A. x) v_1, v_2), v_3 \right)$$

(Given a reduction sequence  $e \mapsto e' \mapsto e'' \mapsto v$ , we say that it takes three steps to fully reduce  $e$ , for example.)

\_\_\_\_\_

**Question 3.** [3 pts] Encode the boolean type `bool` and its constructs `true`, `false`, and `if e then e1 else e2` using the sum type  $A + A$ , the unit type `unit`, and their constructs.

`bool` = \_\_\_\_\_

`true` = \_\_\_\_\_

`false` = \_\_\_\_\_

`if e then e1 else e2` = \_\_\_\_\_

**Question 4.** [2 pts] Give an expression in the extended simply typed  $\lambda$ -calculus that denotes a recursive function  $f$  of type  $A \rightarrow B$  whose formal argument is  $x$  and whose body is  $e$ .

\_\_\_\_\_

**Question 5. [3 pts]** Show the reduction sequence of the expression  $\text{!ref } (\lambda x:A. x)$  in the simply typed  $\lambda$ -calculus with mutable references. The reduction begins with an empty store and uses a location  $l$  when allocating a reference. The reduction judgment has the form  $e \mid \psi \mapsto e' \mid \psi'$  where a store  $\psi$  is a collection of bindings of the form  $l \mapsto v$ .

$\text{!ref } (\lambda x:A. x) \mid \cdot \mapsto$  \_\_\_\_\_

**Question 6. [3 pts]** Complete the rule for the store typing judgment  $\psi :: \Psi$  in the simply typed  $\lambda$ -calculus with mutable references.

$\frac{\text{dom}(\Psi) = \text{dom}(\psi)}{\psi :: \Psi}$  Store

**Question 7. [6 pts]** Consider the environment semantics (using the environment evaluation judgment  $\eta \vdash e \hookrightarrow v$ ) for the simply typed  $\lambda$ -calculus with a base type `bool`:

type	$A ::= P \mid A \rightarrow A$
base type	$P ::= \text{bool}$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e$
environment	$\eta ::= \cdot \mid \eta, x \hookrightarrow v$

Give an inductive definition of values:

\_\_\_\_\_

Write the environment evaluation rule for applications:

\_\_\_\_\_  $\text{App}_e$   
 $\eta \vdash e_1 e_2 \hookrightarrow v$

**Question 8. [3 pts]** What is the language construct in C++ that realizes parametric polymorphism, although it is “a terribly hacked and inadequate feature” from our point of view?

\_\_\_\_\_

## 4 Inductive definition [5 pts]

Suppose that we use a sequence of digits **0** and **1** as a binary representation of a natural number. As usual, the rightmost digit corresponds to the least significant bit and the leftmost digit corresponds to the most significant bit. For example, **1101** denotes a natural number  $2^3 + 2^2 + 2^0 = 13$ . A syntactic category **bin** for such sequences of digits can be inductively defined in several ways, but we use the following definition:

$$\text{bin } b ::= \mathbf{0} \mid \mathbf{1} \mid b\mathbf{0} \mid b\mathbf{1}$$

We wish to inductively define a syntactic category **pbin** for sequences of digits that denote *positive* natural numbers and also do not have a leading **0**. For example, **1101** belongs to **pbin**, but **01101** does not because it has a leading **0**. **0** does not belong to **pbin**, either, because it does not denote a positive natural number.

**Question 1. [3 pts]** Give an inductive definition of **pbin**. You may not introduce auxiliary syntactic categories.

$$\text{pbin } p ::= \underline{\hspace{10em}}$$

**Question 2. [2 pts]** Give an inductive definition of a function *num* which takes a sequence *p* belonging to **pbin** and returns its corresponding decimal number. For example, we have  $\text{num}(\mathbf{10}) = 2$  and  $\text{num}(\mathbf{1101}) = 13$ .

## 5 Programming in the $\lambda$ -calculus [10 pts]

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

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

The goal of this problem is to define a logarithm function  $\log$  which finds the logarithm in base 2 of a given non-zero natural number (encoded as a Church numeral).

- $\log \hat{k}$  evaluates to  $\hat{n}$  if  $2^n \leq k < 2^{n+1}$ .
- $\log$  never takes  $\hat{0}$  as an argument. Hence the result of evaluating  $\log \hat{0}$  is unspecified.

Your answers may use the following pre-defined constructs: `zero`, `one`, `succ`, `if/then/else`, `pair`, `eq`, `halve`, and `fix`.

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

$$\begin{aligned} \text{zero} &= \hat{0} = \lambda f. \lambda x. x \\ \text{one} &= \hat{1} = \lambda f. \lambda x. f x \end{aligned}$$

- `succ` finds the successor of a given natural number.

$$\text{succ} = \lambda \hat{n}. \lambda f. \lambda x. \hat{n} f (f x)$$

- `if e then e1 else e2` is a conditional construct.

$$\text{if } e \text{ then } e_1 \text{ else } e_2 = e e_1 e_2$$

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

$$\begin{aligned} \text{pair} &= \lambda x. \lambda y. \lambda b. b x y \\ \text{fst} &= \lambda p. p (\lambda t. \lambda f. t) \\ \text{snd} &= \lambda p. p (\lambda t. \lambda f. f) \end{aligned}$$

- `eq` tests two natural numbers for equality.

$$\text{eq} = \lambda x. \lambda y. \text{and} (\text{isZero } (x \text{ pred } y)) (\text{isZero } (y \text{ pred } x))$$

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

$$\text{halve} = \lambda \hat{n}. \text{fst } (\hat{n} (\lambda p. \text{pair } (\text{snd } p) (\text{succ } (\text{fst } p)))) (\text{pair zero zero}))$$

- `fix` is the fixed point combinator.

$$\text{fix} = \lambda F. (\lambda f. F \lambda x. (f f x)) (\lambda f. F \lambda x. (f f x))$$

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

$$\begin{aligned} \text{tt} &= \lambda t. \lambda f. t \\ \text{ff} &= \lambda t. \lambda f. f \\ \text{and} &= \lambda x. \lambda y. x y \text{ ff} \\ \text{isZero} &= \lambda x. x (\lambda y. \text{ff}) \text{ tt} \\ \text{next} &= \lambda p. \text{pair} (\text{snd } p) (\text{succ} (\text{snd } p)) \\ \text{pred} &= \lambda \hat{n}. \text{fst} (\hat{n} \text{ next} (\text{pair zero zero})) \end{aligned}$$

**Question 1.** [3 pts] Use the fixed point combinator to define `log`. You may use the above pre-defined constructs, but do not expand them into their definitions.

`log` = \_\_\_\_\_

**Question 2.** [7 pts] Define `log` without using the fixed point combinator. You may use the above pre-defined constructs, but do not expand them into their definitions. (You are not allowed to rewrite your answer to the previous question by expanding `fix` into its definition!)

`log` =  $\lambda \hat{n}.$  \_\_\_\_\_

## 6 Complete call-by-name reduction [20 pts]

Consider the following fragment of the simply typed  $\lambda$ -calculus:

type	$A ::= P \mid A \rightarrow A$
base type	$P$
expression	$e ::= x \mid \lambda x:A. e \mid e e$
value	$v ::= \lambda x:A. e$

Under the *call-by-name* (CBN) strategy, an expression reduces to a value using two reduction rules below:

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{Lam} \quad \frac{}{(\lambda x:A. e) e' \mapsto [e'/x]e} \text{App}$$

Note that the second subexpression in an application (*e.g.*,  $e_2$  in  $e_1 e_2$ ) is not reduced immediately.

In this problem, we will consider a variant of the CBN strategy, called the *complete CBN strategy*, in which we attempt to reduce the expression  $e$  in  $\lambda x:A. e$  before applying the rule *App*. As a result, we reduce  $(\lambda x:A. e) e'$  to  $[e'/x]e$  by the rule *App* only when the function body  $e$  is a normal form. Recall that an expression  $e$  is said to be a normal form if no reduction rule is applicable, *i.e.*, if there is no  $e'$  such that  $e \mapsto e'$ . Thus, if a reduction sequence terminates, it must end up with a normal form.

Under the complete CBN strategy, a normal form is not necessarily a value. For example,  $\lambda x:A. x (\lambda y:B. y)$  is a normal form (because there is no  $e'$  such that  $\lambda x:A. x (\lambda y:B. y) \mapsto e'$ ) and also a value, whereas  $x y$  is a normal form (because there is no  $e'$  such that  $x y \mapsto e'$ ) but not a value. Conversely a value is not necessarily a normal form. For example,  $\lambda x:A. (\lambda y:B. y) x$  is a value but not a normal form because its body  $(\lambda y:B. y) x$  reduces to another expression  $x$ , as shown in  $\lambda x:A. (\lambda y:B. y) x \mapsto \lambda x:A. x$ . We call a normal form that is a value as a *value normal form*, and a normal form that is not a value as a *non-value normal form*.

In order to syntactically distinguish the two kinds of normal forms, we introduce two new syntactic categories:

non-value normal form	$xf ::= x \mid xf e$
normal form	$nf ::= xf \mid \lambda x:A. nf$

Examples of non-value normal forms are  $x (\lambda y:A. y)$  and  $x y$ . Note that a non-value normal form can always be written as  $x e_1 e_2 \cdots e_n$ . A normal form  $nf$  is either a non-value normal form  $xf$  or a value normal form  $\lambda x:A. nf'$ . Note that the body of a value normal form  $\lambda x:A. nf$  is just a normal form, not necessarily another value normal form.

**Question 1.** [6 pts] Give the rules for the reduction judgment  $e \mapsto e'$ . You need three rules.

$\mapsto$	$\mapsto$
$\mapsto$	

**Question 2. [4 pts]** Give the rules for the evaluation judgment  $e \hookrightarrow nf$  which means that an expression  $e$  evaluates to a normal form  $nf$ . You need three rules and we provide one.

$$\frac{}{nf \hookrightarrow nf} \quad \frac{}{\hookrightarrow} \quad \frac{}{\hookrightarrow}$$

**Question 3. [4 pts]** Give the definition of evaluation contexts corresponding to the complete CBN strategy.

evaluation context  $\kappa ::=$  \_\_\_\_\_

**Question 4. [6 pts]** Give the definition of frames and the rules for the state transition judgment  $s \mapsto_{\mathcal{C}} s'$  for the abstract machine  $\mathcal{C}$ .  $\sigma \blacktriangleright e$  means that the machine is currently reducing  $\sigma[e]$ , but has yet to analyze  $e$ .  $\sigma \blacktriangleleft nf$  means that the machine is currently reducing  $\sigma[nf]$  and has already analyzed  $nf$ ; that is, it is returning  $nf$  to the top frame of  $\sigma$ . Fill in the blank:

frame  $\phi ::=$  \_\_\_\_\_  
 stack  $\sigma ::=$   $\square \mid \sigma; \phi$   
 state  $s ::=$   $\sigma \blacktriangleright e \mid \sigma \blacktriangleleft nf$

$$\frac{}{\sigma \blacktriangleright nf \mapsto_{\mathcal{C}}} \quad Nf_{\mathcal{C}}$$

$$\frac{}{\sigma \blacktriangleright e_1 e_2 \mapsto_{\mathcal{C}}} \quad Lam_{\mathcal{C}}$$

$$\frac{}{\sigma \blacktriangleright \lambda x:A. e \mapsto_{\mathcal{C}}} \quad BodyA_{\mathcal{C}}$$

$$\frac{}{\sigma; \lambda x:A. \square \blacktriangleleft nf \mapsto_{\mathcal{C}}} \quad BodyR_{\mathcal{C}}$$

$$\frac{}{\sigma; \square e_2 \blacktriangleleft xnf \mapsto_{\mathcal{C}}} \quad Xnf_{\mathcal{C}}$$

$$\frac{}{\sigma; \square e_2 \blacktriangleleft \lambda x:A. nf \mapsto_{\mathcal{C}}} \quad App_{\mathcal{C}}$$

## 7 Type preservation [15 pts]

In this problem, we use the following fragment of the simply typed  $\lambda$ -calculus. We do not consider base types.

type	$A ::= P \mid A \rightarrow A$
base type	$P$
expression	$e ::= x \mid \lambda x:A. e \mid e e$
value	$v ::= \lambda x:A. e$
typing context	$\Gamma ::= \cdot \mid \Gamma, x : A$

  

$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$	Var	$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x:A. e : A \rightarrow B}$	$\rightarrow I$	$\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e e' : B}$	$\rightarrow E$
$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}$	Lam	$\frac{e_2 \mapsto e'_2}{(\lambda x:A. e) e_2 \mapsto (\lambda x:A. e) e'_2}$	Arg	$\frac{}{(\lambda x:A. e) v \mapsto [v/x]e}$	App

**Question 1. [5 pts]** Fill in the blank in the next page to complete the proof of the substitution lemma. We assume that a typing context is an unordered set and that variables in a typing context are all distinct.

**Question 2. [10 pts]** Fill in the blank in the page after to complete the proof of the type preservation theorem. Unlike the proof given in the Course Notes, we apply rule induction to  $\Gamma \vdash e : A$  instead of  $e \mapsto e'$ . You may use Lemmas 7.2 (Substitution) and 7.1 (Inversion).

**Lemma 7.1 (Inversion).** *Suppose  $\Gamma \vdash e : C$ .*

*If  $e = x$ , then  $x : C \in \Gamma$ .*

*If  $e = \lambda x:A. e'$ , then  $C = A \rightarrow B$  and  $\Gamma, x : A \vdash e' : B$  for some type  $B$ .*

*If  $e = e_1 e_2$ , then  $\Gamma \vdash e_1 : A \rightarrow C$  and  $\Gamma \vdash e_2 : A$  for some type  $A$ .*

**Lemma 7.2 (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  $\Gamma, x : A \vdash e' : C$ . In the third case, we assume (without loss of generality) that  $y$  is a fresh variable such that  $y \notin FV(e)$  and  $y \neq x$ . If  $y \in FV(e)$  or  $y = x$ , we can always choose a different variable by applying an  $\alpha$ -conversion to  $\lambda y : C_1. e''$ .

**Case**  $\frac{y : C \in \Gamma, x : A}{\Gamma, x : A \vdash y : C} \text{Var}$  where  $e' = y$  and  $y : C \in \Gamma$ :  
 $\Gamma \vdash y : C$  from  $y : C \in \Gamma$   
 $[e/x]y = y$  from  $x \neq y$

---

**Case**  $\overline{\Gamma, x : A \vdash x : A} \text{Var}$  where  $e' = x$  and  $C = A$ :  
assumption  
 $[e/x]x = e$

---

**Case**  $\frac{\Gamma, x : A, y : C_1 \vdash e'' : C_2}{\Gamma, x : A \vdash \lambda y : C_1. e'' : C_1 \rightarrow C_2} \rightarrow I$  where  $e' = \lambda y : C_1. e''$  and  $C = C_1 \rightarrow C_2$ :  
by induction hypothesis  
by the rule  $\rightarrow I$   
 $[e/x]\lambda y : C_1. e'' =$  from  $y \notin FV(e)$  and  $x \neq y$

---

**Case**  $\frac{\Gamma, x : A \vdash e_1 : B \rightarrow C \quad \Gamma, x : A \vdash e_2 : B}{\Gamma, x : A \vdash e_1 e_2 : C} \rightarrow E$  where  $e' = e_1 e_2$ :  
by induction hypothesis on \_\_\_\_\_  
by induction hypothesis on \_\_\_\_\_  
by the rule  $\rightarrow E$   
 $\Gamma \vdash [e/x](e_1 e_2) : C$  from \_\_\_\_\_  $\square$

**Theorem 7.3 (Type preservation).** *If  $\Gamma \vdash e : A$  and  $e \mapsto e'$ , then  $\Gamma \vdash e' : A$ .*

*Proof.* By rule induction on the judgment  $\Gamma \vdash e : A$ .

**Case**  $\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{Var}$  where  $e = x$ :

There is no expression  $e'$  such that  $x \mapsto e'$ , so we do not need to consider this case.

**Case**  $\frac{\Gamma, x : A_1 \vdash e'' : A_2}{\Gamma \vdash \lambda x : A_1. e'' : A_1 \rightarrow A_2} \rightarrow I$  where  $e = \lambda x : A_1. e''$  and  $A = A_1 \rightarrow A_2$ :

There is no expression  $e'$  such that  $\lambda x : A_1. e'' \mapsto e'$ , so we do not need to consider this case.

**Case**  $\frac{\Gamma \vdash e_1 : C \rightarrow A \quad \Gamma \vdash e_2 : C}{\Gamma \vdash e_1 e_2 : A} \rightarrow E$  where  $e = e_1 e_2$ :

There are three subcases depending on the reduction rule used in the derivation of  $e \mapsto e'$ . Note that if  $e_1$  is a  $\lambda$ -abstraction, it must have the form  $\lambda x : C. e''$  by Lemma 7.1 with  $\Gamma \vdash e_1 : C \rightarrow A$ .

**Subcase**  $\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{Lam}$  where  $e' = e'_1 e_2$ :

\_\_\_\_\_ by induction hypothesis on \_\_\_\_\_ with \_\_\_\_\_

\_\_\_\_\_ from \_\_\_\_\_

**Subcase**  $\frac{e_2 \mapsto e'_2}{(\lambda x : C. e'') e_2 \mapsto (\lambda x : C. e'') e'_2} \text{Lam}$  where  $e_1 = \lambda x : C. e''$  and  $e' = (\lambda x : C. e'') e'_2$ :

\_\_\_\_\_ by induction hypothesis on \_\_\_\_\_ with \_\_\_\_\_

\_\_\_\_\_ from \_\_\_\_\_

**Subcase**  $\frac{}{(\lambda x : C. e'') v \mapsto [v/x]e''} \text{App}$  where  $e_1 = \lambda x : C. e''$  and  $e_2 = v$  and  $e' = [v/x]e''$ :

\_\_\_\_\_ by Lemma 7.1 with \_\_\_\_\_

\_\_\_\_\_ by Lemma 7.2 with \_\_\_\_\_ and \_\_\_\_\_

□