Name: Hemos ID:

CSE-321 Programming Languages 2011
Midterm

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	18	7	12	21	28	14	100

- There are six problems on 26 pages in this exam.
- The maximum score for this exam is 100 points.
- Be sure to write your name and Hemos ID.
- In Problems 1 and 2, write your answers exactly as you would type on the screen. The grading for Problems 1 and 2 will be strict (*i.e.,* no partial points).
- When writing individual proof steps in Problems 3 and 5.2, 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 [18 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 factorial function.

```
f u n _ f a c t _ n _ = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ i f _ n _ < = _ 0 _ t h e n _ 1 _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ e l s e _ n _ * _ f a c t _ ( n _ - _ 1 ) _ _ _ _ _ _ _ _
```

Question 1. [**3 pts**] Give a *recursive* implementation of `power`. You may use the function `List.map` and the operator `@`.

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

(Description) `power` l returns a list consisting of all possible sublists of l. The order of the returned list does not matter.

(Invariant) Elements in an input list are all distinct.

(Example) `power [1, 2, 3]` may return
`[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]`.

```
fun power [] = [nil]
  | power (h :: t) =

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

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

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

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

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

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

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

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

Question 2. [**3 pts**] Give a *tail recursive* implementation of `sumList`.

(Type) `sumList: int list -> int`

(Description) `sumList` l returns the sum of integers in l.

(Example) `sumList [1, 2, 3]` returns 6.

```
fun sumList l =

l e t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

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

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

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

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

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

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

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

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

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

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

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

e n d _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

In Questions 3 and 4, 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 3. [**3 pts**] Complete the following definition of the function `map` using `foldr`.

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

(Description) `map` f l takes a function f and a list l and applies f to each element of l. That is, `map` f $[x_1, x_2, ..., x_n]$ computes $[f\ \ x_1, f\ \ x_2, ..., f\ \ x_n]$, or [ ] if the list is empty.

```
fun map f l =
    let
      fun g _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

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

      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    in
      foldr g _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

Question 4. [**3 pts**] Complete the following definition of the function `foldl` using `foldr`.

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

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

```
fun foldl f e l =
    let
      fun g _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

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

      _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    in
      foldr g _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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

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

Question 5. [**3 pts**] Consider the following definitions:

```
signature SET =
sig
  type 'a set
  (* create l returns a set consisting of the elements of l *)
  val create : ''a list -> ''a set
end

structure SetOne :> SET =
struct
  type 'a set = 'a list
  ...
end

structure SetTwo : SET =
struct
  type 'a set = 'a list
  ...
end
```

Give an implementation of the function `union` for the structure `SetOne`. If impossible, state "impossible." You may use the operator `@`.

(Type) `union: ''a list -> ''a SetOne.set -> ''a SetOne.set`

(Description) `union` l s takes a list l and a set s, and returns a set consisting of the elements of l and s.

(Example) `union` $[x_1, x_2]$ `(SetOne.create` $[y_1, y_2]$`)` returns a set consisting of x_1, x_2, y_1, and y_2.

```
fun union l s =
```

_ _

_ _

_ _

_ _

Question 6. [**3 pts**] Give an implementation of the function `union` for the structure `SetTwo`. If it is impossible to implement `union`, then state "impossible." You may use the operator `@`.

(Type) `union: ''a list -> ''a SetTwo.set -> ''a SetTwo.set`

(Description) `union` l s takes a list l and a set s, and returns a set consisting of the elements of l and s.

(Example) `union` $[x_1, x_2]$ `(SetTwo.create` $[y_1, y_2]$`)` returns a set consisting of x_1, x_2, y_1, and y_2.

```
fun union l s =
```

_ _

_ _

_ _

_ _

2 α-equivalence relation [7 pts]

Recall the definition of the α-equivalence relation for the untyped λ-calculus:

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

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

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

Here we write $e \equiv_\alpha e'$ to mean that expressions e and e' are α-equivalent.

The goal of this problem is to implement the α-equivalence relation as an SML function. Suppose that we have the following definitions of types and functions:

- Type definitions:

```
type var = string

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

- `fv : var -> exp -> bool`

  `fv` x e returns `true` if and only if variable x is a free variable of expression e.

- `swap : var -> var -> exp -> exp`

  `swap` x y e returns $[x \leftrightarrow y]e$.

Use these definitions and define the function `aequi`:

- `aequi: exp -> exp -> bool`

  `aequi` e_1 e_2 returns `true` if and only if $e_1 \equiv_\alpha e_2$ holds.

fun aequi

3 Inductive definitions [12 pts]

The goal in this problem is to prove the correctness of the following function `f` which invokes a tail recursive function fact to calculate the factorial of a given integer:

```
fun fact 0 a = a
  | fact n a = fact (n - 1) (n * a)
fun f n = fact n 1
```

To prove the correctness of `f`, we rewrite the definition of `fact` into the following judgment $\mathsf{fact}\ n\ a\ s$ and two inference rules:

$$\mathsf{fact}\ n\ a\ s \quad \Leftrightarrow \quad \texttt{fact}\ n\ a \text{ evaluates to } s$$

$$\frac{}{\mathsf{fact}\ 0\ a\ a}\ Fbase \qquad \frac{\mathsf{fact}\ (n-1)\ (n \times a)\ s \quad n > 0}{\mathsf{fact}\ n\ a\ s}\ Frec$$

Prove Theorem 3.1 shown below. We assume $n \geq 0$ and $0! = 1$.

You may introduce a lemma in your proof. If you choose to introduce a lemma, state the lemma and prove it separately.

Theorem 3.1. *If* $\mathsf{fact}\ n\ 1\ s$ *, then* $s = n!$.

4 λ-Calculus [21 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. **[4 pts]** Show the reduction sequence of a given expression under the call-by-value strategy and call-by-name strategy. In each step, underline the redex. Do *not* rename bound variables.

call-by-value:

$$(\lambda t.\, \lambda f.\, f)\ ((\lambda y.\, y)\ (\lambda z.\, z))\ ((\lambda y'.\, y')\ (\lambda z'.\, z'))$$

$\mapsto$ ____

$\mapsto$ ____

$\mapsto$ ____

$\mapsto$ ____

call-by-name:

$$(\lambda t.\, \lambda f.\, f)\ ((\lambda y.\, y)\ (\lambda z.\, z))\ ((\lambda y'.\, y')\ (\lambda z'.\, z'))$$

$\mapsto$ ____

$\mapsto$ ____

$\mapsto$ ____

Question 2. [2 pts] Give an expression whose reduction does not terminate under the call-by-value strategy.

____

Question 3. [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.

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

$$[e/x]y \quad = \quad \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \qquad \textit{if } x \neq y$$

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

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

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

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

where ____________________

____

____

____

In Questions 4, 5 and 6, you may use the following pre-defined constructs: zero, one, succ, $\mathsf{if/then/else}$, pair, fst, snd, pred, eq, and fix. You do not need to copy definitions of these constructs.

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

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

- succ finds the successor of a given natural number.

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

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

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

- pair creates a pair of 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.

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

- eq tests two natural numbers for equality.

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

- fix is the fixed point combinator.

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

Question 4. [2 pts] Define the function sum such that $\mathsf{sum}\ \hat{n}$ evaluates to the sum of natural numbers from $\hat{0}$ to $\hat{n}$. You may use the fixed point combinator.

sum = ____

____

____

Question 5. **[2 pts]** Define the function add such that add $\hat{m}$ $\hat{n}$ evaluates to $\widehat{m+n}$.

add $=$ ____

Question 6. **[2 pts]** Define the function mult such that mult $\hat{m}$ $\hat{n}$ evaluates to $\widehat{m*n}$.
Do not use the function add. *Do not copy the definition of* add.

mult $=$ ____

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.

$$\tau_i^n(N_1\ N_2) \quad = \quad \underline{\hspace{6cm}}$$

$$\tau_i^n(\lambda.\, N) \quad = \quad \underline{\hspace{6cm}}$$

$$\tau_i^n(m) \quad = \quad \underline{\hspace{6cm}} \qquad \text{if } m \geq i$$

$$\tau_i^n(m) \quad = \quad \underline{\hspace{6cm}} \qquad \text{if } m < i$$

5 Simply typed λ-calculus [28 pts]

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

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

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

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

The typing rules are as follows:

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

$$
\frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}}\ \textsf{True}
\qquad
\frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}}\ \textsf{False}
\qquad
\frac{\Gamma \vdash e : \textsf{bool} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2 : A}\ \textsf{If}
$$

Question 1. [2 pts] State the two theorems, progress and type preservation, constituting type safety:

Theorem 5.1. *(Progress).*

Theorem 5.2. *(Preservation).*

Question 2. **[10 pts]** Fill in the blank to complete the proof of the substitution lemma. Assume the definition of substitution $[e/x]e'$ as given in the course notes.

Lemma 5.3 (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 consider only two cases shown below. In the first 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 α-conversion to $\lambda y\!:\!C_1.\, e''$.

Case $\dfrac{\Gamma, x : A, y : C_1 \vdash e'' : C_2}{\Gamma, x : A \vdash \lambda y\!:\!C_1.\, e'' : C_1 \rightarrow C_2} \rightarrow\mathsf{I}$ where $e' = \lambda y\!:\!C_1.\, e''$ and $C = C_1 \rightarrow C_2$:

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

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

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

____________________________ ____________________________

$\square$

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

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

Complete the typing rules.

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\Gamma \vdash (e_1, e_2) :}$$

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\Gamma \vdash \textsf{fst}\ e :}$$

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\Gamma \vdash \textsf{snd}\ e :}$$

Question 4. [6 pts] Assume the call-by-value strategy as given in the course notes. Modify one of the typing rules so that the progress property is destroyed while the type preservation property is maintained. (You should modify only one typing rule.) Under the revised types system, give an example of a closed well-typed expression which does not reduce to another expression. Show the typing derivation of your expression under the revised type system.

modified rule:

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

expression:

---

typing derivation:

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{\cdot \vdash}$$

Question 5. [2 pts] Consider the extension of the simply-typed λ-calculus with sum types:

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

Write the reduction rules for these constructs under the call-by-value strategy. You should extend the definition of values and give reduction rules that maintain type safety.

$$\begin{array}{rrcl} \textsf{value} & v & ::= & \cdots \mid \underline{\hspace{8cm}} \end{array}$$

$$\frac{\hspace{12cm}}{\hspace{1cm}\mapsto\hspace{1cm}}$$

$$\frac{\hspace{12cm}}{\hspace{1cm}\mapsto\hspace{1cm}}$$

$$\frac{\hspace{12cm}}{\hspace{1cm}\mapsto\hspace{1cm}}$$

$$\frac{\hspace{12cm}}{\hspace{1cm}\mapsto\hspace{1cm}}$$

$$\frac{\hspace{12cm}}{\hspace{1cm}\mapsto\hspace{1cm}}$$

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.

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

$$\mapsto$$

Question 7. **[4 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 $(\mathsf{int} \to \mathsf{bool}) \times (\mathsf{int} \to \mathsf{bool})$ that encodes both `even` and `odd` in the simply-typed λ-calculus:

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

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

6 Evaluation contexts and environments [14 pts]

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

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

Question 1. [2 pts] Give the definition of evaluation contexts for the call-by-value strategy.

evaluation context κ ::= ____

Question 2. [2 pts] Give the definition of evaluation contexts for the call-by-name strategy.

evaluation context κ ::= ____

Question 3. [6 pts] Consider the following definition of the abstract machine C under the <u>*call-by-name*</u> strategy:

$$\begin{array}{lrcl} \textsf{frame} & \phi & ::= & \rule{8cm}{0.4pt} \\ \textsf{stack} & \sigma & ::= & \square \mid \sigma; \phi \\ \textsf{state} & s & ::= & \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v \end{array}$$

Complete the definition of frame ϕ and the reduction rules for the abstract machine C. We use the following reduction judgment:

$$s \mapsto_{\mathsf{C}} s' \quad \Leftrightarrow \quad \text{The state } s \text{ reduces to the state } s'$$

$\longmapsto$C

$\longmapsto$C

$\longmapsto$C

$\longmapsto$C

$\longmapsto$C

$\longmapsto$C

Question 4. [4 pts] The key idea behind the environment semantics is to postpone a substitution $[v/x]e$ by storing a pair of v and x in an *environment*. We use the following definition of environment:

$$\textsf{environment} \quad \eta \quad ::= \quad \cdot \mid \eta, x \hookrightarrow v$$

$\cdot$ denotes an empty environment, and $x \hookrightarrow v$ means that variable x is to be replaced by value v. We use an *environment evaluation judgment* of the form $\eta \vdash e \hookrightarrow v$:

$$\eta \vdash e \hookrightarrow v \qquad \Leftrightarrow \qquad e \textit{ evaluates to } v \textit{ under environment } \eta$$

Complete the definition of values. Then give rules for the environment evaluation judgment for variables, λ-abstractions, and applications under the *call-by-value* strategy. Your definition of values should include closures covered in class.

$$\textsf{value} \quad v \quad ::= \quad \underline{\hspace{8cm}}$$

$$\frac{}{\eta \vdash x \hookrightarrow}$$

$$\frac{}{\eta \vdash \lambda x : A.\, e \hookrightarrow}$$

$$\frac{}{\eta \vdash e1 \ e2 \hookrightarrow}$$