There are five problems on 9 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 Problem 2, please write conclusion in the left and justification in the right, as in the course notes.

For a ‘true or false’ question, a wrong answer has a penalty equal to the points assigned to it.

You have one and a half hours for this exam.
1 OCaml Programming [10 points]

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.

```ocaml
let rec sum n =
  if n = 0 then 0
  else sum (n-1)
```

Question 1. [5 points] Tail-recursive union for computing the union of two sets

(Type) union: ’a list -> ’a list -> ’a list

(Description) union \(S \cup T\) returns a set that includes all elements of \(S\) and \(T\) without duplication of any element. The order of elements in the return value does not matter. You may use the `List.exists` function:

```ocaml
# List.exists;;
- : ('a -> bool) -> 'a list -> bool = <fun>
```

union must be a tail-recursive function and should not introduce auxiliary functions other than `List.exists`.

(Invariant) Each input set consists of distinct elements.

(Example) union \([1; 2; 3]\) \([2; 4; 6]\) returns \([3; 1; 2; 4; 6]\).

```ocaml
let rec union l1 l2 =
  match l1 with
  | [] -> l2
  | x :: xs ->
    if List.exists (fun y -> y = x) l2 then union xs l2
    else union xs (x :: l2)
```

Question 2. [5 points] fold_left using fold_right

(Type) fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

(Description) Implement fold_left using the fold_right function:

```ocaml
# open List;;
# fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

fold_left and fold_right are specified as follows:

\[
\text{fold_left} \quad f \quad a_0 \quad [l_1, l_2, \ldots, l_n] = f(\ldots f(f(a_0, l_1)l_2)\ldots l_n) \quad (n \geq 0)
\]

\[
\text{fold_right} \quad f \quad [l_1, l_2, \ldots, l_n] \quad a_0 = f(l_1, \ldots f(l_{n-1}, f(l_n, a_0))) \quad (n \geq 0)
\]

(Hint) OCaml is a functional language :-)

```ocaml
let rec fold_left f a l =
  fold_right (fun x g a -> g (f a x)) l (fun x -> x) a
```
2 Inductive proof on strings of matched parentheses [20 points]

In this problem, we study a system of strings of matched parentheses. First we define a syntactic category \( \text{paren} \) for strings of parentheses:

\[
\text{paren} \\
\hspace{1cm} s := \epsilon \mid (s) s
\]

\( \epsilon \) stands for the empty string (i.e., \( \epsilon s = s s \epsilon = s \)). \( \text{paren} \) specifies a language of strings of parentheses with no constraint on the use of parentheses.

To identify strings of matched parentheses, we introduce a judgment \( s \mid \text{paren} \) with the following inference rules:

\[
\begin{array}{c}
\epsilon \mid \text{paren} \quad \text{Leps} \\
\frac{s_1 \mid \text{paren} \quad s_2 \mid \text{paren}}{(s_1) s_2 \mid \text{paren}} \quad \text{Lseq}
\end{array}
\]

We also introduce another judgment \( s \mid \text{tparen} \) for identifying strings of matched parentheses:

\[
\begin{array}{c}
\epsilon \mid \text{tparen} \quad \text{Teps} \\
\frac{s_1 \mid \text{tparen} \quad s_2 \mid \text{tparen}}{s_1 (s_2) \mid \text{tparen}} \quad \text{Tseq}
\end{array}
\]

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 \( s \mid \text{tparen} \), then \( s \mid \text{paren} \).

**Lemma 2.2.** If \( s \mid \text{paren} \), then \( s' \mid \text{paren} \) implies \( s s' \mid \text{paren} \).

**Proof.** By rule induction on \( s \mid \text{paren} \).

**Case** \( \epsilon \mid \text{paren} \) \( \text{Leps} \) where \( s = \epsilon \):

\( s' \mid \text{paren} \) \hspace{2cm} assumption

\( s s' = \epsilon s' = s' \)

**Case** \( s_1 \mid \text{paren} \quad s_2 \mid \text{paren} \) \( \text{Lseq} \) where \( s = (s_1) s_2 \):

\( s' \mid \text{paren} \) \hspace{2cm} assumption

\( s s' = (s_1) s_2 s' \)

\( s' \mid \text{paren} \) implies \( s_2 \mid \text{paren} \) \hspace{2cm} by induction hypothesis on \( s_2 \mid \text{paren} \)

\( s_2 s' \mid \text{paren} \) \hspace{2cm} from \( s' \mid \text{paren} \)

\( (s_1) s_2 s' \mid \text{paren} \) \hspace{2cm} by the rule \( \text{Lseq} \) with \( s_1 \mid \text{paren} \) and \( s_2 \mid \text{paren} \) \( \square \)

**Proof of Theorem 2.1.** By rule induction on \( s \mid \text{tparen} \).

**Case** \( \epsilon \mid \text{tparen} \) \( \text{Teps} \) where \( s = \epsilon \):

\( s \mid \text{paren} \) \hspace{2cm} by the rule \( \text{Leps} \)

**Case** \( s_1 \mid \text{tparen} \quad s_2 \mid \text{tparen} \) \( \text{Tseq} \) where \( s = s_1 (s_2) \):

\( s_1 \mid \text{paren} \) \hspace{2cm} by induction hypothesis on \( s_1 \mid \text{tparen} \)
3 Untyped \(\lambda\)-Calculus [30 points]

The abstract syntax of the untyped \(\lambda\)-calculus is given as follow:

\[
\text{expression } e ::= x \mid \lambda x. e \mid e \quad e
\]

We may use other names for variables (e.g., \(z, s, t, f, \text{arg, accum, and so on}\)). The scope of a \(\lambda\)-abstraction \(\lambda x. e\) extends as far to the right as possible. We use a reduction judgment of the form \(e \rightarrow e'\):

\[
e \rightarrow e' \Leftrightarrow e \text{ reduces to } e'
\]

**Question 1.** [2 points] 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?

True

**Question 2.** [5 points] Complete the inductive definition of substitution. You may use \([e/x]e\) for the expression obtained by replacing all occurrences of \(x\) in \(e\) by \(y\) and all occurrences of \(y\) in \(e\) by \(x\).

\[
[e/x]x = e
\]

\[
[e/x]y = y \quad \text{if } x \neq y
\]

\[
[e/x](e_1 e_2) = [e/x]e_1 [e/x]e_2
\]

\[
[e'/x]\lambda x. e = \lambda x. e
\]

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

\[
[e'/x]\lambda y. e = \lambda z. [e'/x][y \leftrightarrow z]e \quad \text{if } x \neq y, y \notin \text{FV}(e'), z \neq x, z \notin \text{FV}(e')
\]

**Question 3.** [3 points] Show the reduction sequence under the call-by-name strategy. Underline the redex at each step.
\[(\lambda x. \lambda y. x) \ ((\lambda x. \lambda y. y) ) ( \lambda z. z )\]

\[\mapsto ( \lambda y. y \ ((\lambda x. \lambda y.y)) \ ) ( \lambda z. z )\]

\[\mapsto ( \lambda z. z ) \ ((\lambda x. \lambda y.y)) \]

\[\mapsto ( \lambda x. x ) \ (\lambda y.y)\]

\[\mapsto (\lambda y.y)\]

**Question 4. [2 points]** Give an expression whose reduction neither gets stuck nor terminates under the call-by-value strategy.

\[(\lambda x.x) \ (\lambda x.x)\]

**Question 5. [3 points]** Complete the reduction rules for the call-by-value strategy. You may use the substitution that you defined earlier:

\[
\begin{align*}
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} & \quad \text{Lam} \\
\frac{e_2 \rightarrow e'_2}{(\lambda x. e) e_2 \rightarrow (\lambda x. e) e'_2} & \quad \text{Arg} \\
\frac{(\lambda x. e) \ v \rightarrow [v/x]e}{\text{App}} & \quad \text{App}
\end{align*}
\]

**Programming in the untyped \(\lambda\)-calculus**

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

\[
\tilde{n} = \lambda f. f^n = \lambda f. \lambda x. f f f \cdots f x
\]

**Question 6. [5 points]** Define the function \(\text{exp}\) for exponentiation such that \(\text{exp} \ \tilde{m} \ \tilde{n}\) evaluates to a church numeral for the product of \(n\) copies of \(m\). In other words, \(\text{exp} \ \tilde{m} \ \tilde{n} \rightarrow^* \tilde{m} \tilde{n}\).

\(\text{exp} = \lambda \tilde{m}. \lambda \tilde{n}. \tilde{n} \ (\text{mult} \ \tilde{m}) \ \hat{1},\) or \(\text{exp} = \lambda \tilde{m}. \lambda \tilde{n}. \tilde{n} \ \tilde{m}\)
de Bruijn expressions

Following is the definition of de Bruijn expressions:

\[
\text{de Bruijn expression } M ::= n \mid \lambda. M \mid M \; M
\]

\[
\text{de Bruijn index } n ::= 0 \mid 1 \mid 2 \mid \cdots
\]

**Question 7. [5 points]** 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 \( \tau^n_i(N) \).

\[
\sigma_n(M_1 \; M_2, N) = \sigma_n(M_1, N) \; \sigma_n(M_2, N)
\]

\[
\sigma_n(\lambda. M, N) = \lambda. \sigma_{n+1}(M, N)
\]

\[
\sigma_n(m, N) = m \quad \text{if } m < n
\]

\[
\sigma_n(n, N) = \tau^n_0(N)
\]

\[
\sigma_n(m, N) = m - 1 \quad \text{if } m > n
\]

**Question 8. [5 points]** Complete the definition of \( \tau^n_i(N) \), 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.

\[
\tau^n_i(N_1 \; N_2) = \tau^n_i(N_1) \; \tau^n_i(N_2)
\]

\[
\tau^n_i(\lambda. N) = \lambda. \tau^n_{i+1}(N)
\]

\[
\tau^n_i(m) = m + n \quad \text{if } m \geq i
\]

\[
\tau^n_i(m) = m \quad \text{if } m < i
\]
4 Simply-typed λ-calculus [15 points]

In this section, we assume the simply-typed λ-calculus. We use $A$, $B$, $C$ for metavariables for types, $e$ for expressions, and $\Gamma$ for typing contexts. We use a typing judgment $\Gamma \vdash e : A$ to mean that under typing context $\Gamma$, expression $e$ has type $A$. We use a reduction judgment $e \mapsto e'$ to mean that expression $e$ reduces to expression $e'$.

The abstract syntax for the simply typed λ-calculus is given as follows:

$$\begin{align*}
\text{type} & \quad A ::= \ P \mid A \to A \\
\text{base type} & \quad P ::= \ \text{bool} \\
\text{expression} & \quad e ::= \ x \mid \lambda x : A. e \mid e \ e \mid \text{true} \mid \text{false} \mid \text{if} \ e \ \text{then} \ e \ \text{else} \ e \\
\text{value} & \quad v ::= \ \lambda x : A. e \mid \text{true} \mid \text{false} \\
\text{typing context} & \quad \Gamma ::= \cdot \mid \Gamma, x : A
\end{align*}$$

**Question 1.** [3 points] Type safety guarantees that evaluating a well-typed expression (i.e., running a well-typed program) eventually terminates, never producing non-termination. True or false?

False.

**Question 2.** [3 points] State the weakening property of typing judgments:

(Weakening).

If $\Gamma \vdash e : C$, then $\Gamma, x : A \vdash e : C$.

**Question 3.** [6 points] State two theorems, progress and type preservation, constituting type safety:

(Progress).

If $\cdot \vdash e : A$ for some type $A$,

then either $e$ is a value or there exists $e'$ such that $e \mapsto e'$.

(Type preservation).

If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$.

**Question 4.** [3 points] Consider the extension of the simply-typed λ-calculus with sum types:

$$\begin{align*}
\text{type} & \quad A ::= \cdots \mid A + A \\
\text{expression} & \quad e ::= \cdots \mid \text{inl}_A e \mid \text{inr}_A e \mid \text{case} \ e \ \text{of} \ \text{inl} \ x. e \mid \text{inr} \ x. e
\end{align*}$$

Write the typing rule for case $e$ of inl $x. e \mid \text{inr} \ x. e$:

$$\frac{\Gamma \vdash e : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : C \quad \Gamma, x_2 : A_2 \vdash e_2 : C}{\Gamma \vdash \text{case} \ e \ \text{of} \ \text{inl} \ x_1. e_1 \mid \text{inr} \ x_2. e_2 : C} + E$$
5 Evaluation contexts and abstract machine $C$ [25 points]

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

<table>
<thead>
<tr>
<th>type</th>
<th>$A$ ::= $P$</th>
<th>$A \rightarrow A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>base type</td>
<td>$P$ ::= bool</td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>$e$ ::= $x$</td>
<td>$\lambda x : A. e$</td>
</tr>
</tbody>
</table>

**Question 1.** [4 points] Give the definition of evaluation contexts for the call-by-name strategy.

\[
\text{evaluation context} \quad \kappa ::= \square \mid \kappa \ e \mid \text{if } \kappa \text{ then } e \text{ else } e
\]

**Question 2.** [3 points] Under the call-by-value strategy, give an expression $e$ such that

- $e = \kappa[e']$ where $e'$ is the redex, and
- $e$ reduces to $e_0$ that is decomposed to $\kappa'[e'']$ where $e''$ is the redex for the next reduction and $\kappa \neq \kappa'$.

\[(\lambda x : \text{bool} \rightarrow \text{bool}. \ x) \ (\lambda y : \text{bool}. \ y) \ \text{true}\]

**Abstract machine $C$**

Consider the simply-typed $\lambda$-calculus under the call-by-value reduction strategy. The abstract machine $C$ uses two states:

<table>
<thead>
<tr>
<th>state</th>
<th>$s$ ::= $\sigma \triangleright e$</th>
<th>$\sigma \triangleright v$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\sigma \triangleright e$ means that the machine is currently reducing $\sigma[e]$, but has yet to analyze $e$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sigma \triangleright v$ means that the machine is currently reducing $\sigma[v]$ and has already analyzed $v$.</td>
<td></td>
</tr>
</tbody>
</table>

**Question 3.** [3 points] Give the definitions of frames and stacks:

<table>
<thead>
<tr>
<th>frame</th>
<th>$\phi ::= \square e \mid \square \text{ if } \square \text{ then } e_1 \text{ else } e_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>$\sigma ::= \square \mid \sigma ; \phi$</td>
</tr>
</tbody>
</table>
Question 4. [10 points] Give the rules for the reduction judgment $s \rightarrow_C s'$ for the abstract machine $C$.

$$
\begin{align*}
\sigma \triangleright v & \rightarrow_C \sigma \triangleleft v & Val_C \\
\sigma;\Box e_2 \triangleleft e_1 & \rightarrow_C \sigma; (\lambda x : A.e) \square \triangleright e_2 & Arg_C \\
\sigma; \Box (\lambda x : A.e) \square \triangleleft v & \rightarrow_C \sigma \triangleright [v/x]e & App_C \\
\sigma \triangleright \text{if } e \text{ then } e_1 \text{ else } e_2 & \rightarrow_C \sigma; \text{if } \boxempty \text{ then } e_1 \text{ else } e_2 \triangleright e & If_C \\
\sigma; \Box \text{if } e \text{ then } e_1 \text{ else } e_2 & \rightarrow_C \sigma \triangleright \text{true} & If_{true_C} \\
\sigma; \Box \text{if } e \text{ then } e_1 \text{ else } e_2 & \rightarrow_C \sigma \triangleright \text{false} & If_{false_C}
\end{align*}
$$

Question 5. [5 points] State the correctness of the abstract machine $C$ in terms of the reduction judgment $e \rightarrow e'$. You may use $\rightarrow^*$ and $\rightarrow^*_C$ for the reflexive and transitive closures of $\rightarrow$ and $\rightarrow_C$.

$e \rightarrow^* v$ if and only if $\Box \triangleright e \rightarrow^*_C \Box \triangleleft v$. 

9