There are six problems on 24 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 Problems 2 and 6, 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 [14 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 sum function.

```sml
fun sum n = if n = 0 then 1 else sum (n - 1)
```

Question 1. [4 pts] The definition of a tree for binary trees is as follows:

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

Give a tail-recursive implementation of inorder for an inorder traversal of binary trees.

(Type) inorder: 'a tree -> 'a list

(Description) inorder t returns a list of elements produced by an inorder traversal of the tree t.

(Example) inorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4)) returns [1, 3, 2, 7, 4].

(Hint) inorder can be implemented as follows:

```sml
fun inorder t = 
  let 
    fun inorder' (t': 'a tree) (post : 'a list) : 'a list = ... 
  in 
    inorder' t [ ] 
  end
```

post will be a list of elements to be appended to the result of an inorder traversal of t'. For example, when inorder' visits the node marked 2 in the tree below, post will be bound to [1, 6, 3, 7].

```
      1
     /   
    2     3
   /     / 
  4     5   6 7
```
fun inorder t =
In Questions 2, 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 2. [5 pts]** Complete the function `lrev` using `foldr`. You may use the operator `@` for list concatenation.

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

(Description)`lrev l` returns the reversed list of an input list `l`.

(Example)`lrev [1, 2, 3, 4]` returns `[4, 3, 2, 1]`

```ocaml
fun lrev l =
    let
        val f = ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
        ___________
Question 3. [5 pts] A signature SET for sets is given as follows:

signature SET =
  sig
    type 'a set
    val empty : ''a set
    val member : ''a set -> ''a -> bool
    val insert : ''a set -> ''a -> ''a set
    val intersection : ''a set -> ''a set -> ''a set
    val difference : ''a set -> ''a set -> ''a set
  end

- empty is an empty set.
- 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.
- intersection s t returns the intersection of s and t.
- difference s t returns the set of elements which are members of s, but not members of t.

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. You may use true, false, not, and also, and or else.

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

    val empty = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    fun member s = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    fun insert s x = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    fun intersection s t = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

    fun difference s t = _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
                   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
  end
2 Inductive proof on strings of matched parentheses [15 pts]

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

\[
\texttt{paren} \quad s ::= \epsilon | (s) s
\]

To identify strings of matched parentheses, we introduce a judgment \texttt{s lparen} with the following inference rules:

\[
\frac{}{\epsilon \texttt{lparen}} \quad \frac{s_1 \texttt{lparen} \quad s_2 \texttt{lparen}}{(s_1) \texttt{lparen} s_2 \texttt{lparen}} \quad \text{Lseq}
\]

We also introduce another judgment \texttt{s tparen} for identifying strings of matched parentheses:

\[
\frac{}{\epsilon \texttt{tparen}} \quad \frac{s_1 \texttt{tparen} \quad s_2 \texttt{tparen}}{s_1 (s_2) \texttt{tparen}} \quad \text{Tseq}
\]

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 \(s \texttt{lparen}\), then \(s \texttt{tparen}\).
3  \( \lambda \)-Calculus [29 pts]

In this problem, we study the properties of the untyped \( \lambda \)-calculus:

\[
\begin{align*}
\text{expression} & \quad e ::= x \mid \lambda x. e \mid e \, e \\
\text{value} & \quad v ::= \lambda x. e
\end{align*}
\]

The reduction judgment is as follows:

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

**Question 1.** [5 pts] Complete the inductive definition of substitution. You may use \([(x \mapsto y)e]e\) for the expression obtained by replacing all occurrences of \(x\) in \(e\) by \(y\) and all occurrences of \(y\) in \(e\) by \(x\).

\[
\begin{align*}
[e/x]x & = e \\
[e/x]y & = \text{__________________________ if } x \neq y \\
[e/x](e_1 \, e_2) & = \text{__________________________} \\
[e'/x]\lambda x. e & = \text{__________________________} \\
[e'/x]\lambda y. e & = \text{__________________________ if } x \neq y, y \notin \text{FV}(e') \\
[e'/x]\lambda y. e & = \text{__________________________ if } x \neq y, y \in \text{FV}(e') \\
\end{align*}
\]

where __________________________

__________________________

__________________________

__________________________
Question 2. [3 pts] Complete the reduction rules for the call-by-value strategy. You may use the substitution which you defined in the previous question:

\[
\begin{align*}
&\text{e}_1 \text{e}_2 \mapsto \text{e}'_1 \text{e}_2 \\
&\text{(λx.e) } \text{e}_2 \mapsto \text{(λx.e)'_2} \\
&\text{(λx.e)} \text{v} \mapsto [\text{v/x}] \text{e}
\end{align*}
\]

Question 3. [3 pts] Show the reduction sequence of a given expression under the call-by-name strategy. Do not rename bound variables.

call-by-name:

\[
(\lambda t. \lambda f. f) \ ((\lambda x. x) \ (\lambda y. y)) \ ((\lambda z. z) \ (\lambda w. w))
\]

\[
\mapsto \text{__________________________}
\]

\[
\mapsto \text{__________________________}
\]

\[
\mapsto \text{__________________________}
\]

\[
\mapsto \text{__________________________}
\]

\[
\mapsto \text{__________________________}
\]

\[
\mapsto \text{__________________________}
\]
In Questions 4 and 5, you may use the following pre-defined constructs: zero, one, tt, ff, and, or, and pred. You do not need to copy definitions of these constructs.

- **zero** and **one** encode the natural numbers zero and one, respectively.
  \[
  \text{zero} = 0 = \lambda f. \lambda x. x \\
  \text{one} = 1 = \lambda f. \lambda x. f \ x
  \]

- **tt** and **ff** represent the boolean values true and false, respectively.
  \[
  \text{tt} = \lambda t. \lambda f. t \\
  \text{ff} = \lambda t. \lambda f. f
  \]

- **and** and **or** encode the boolean operators ‘and’ and ‘or’, respectively.
  \[
  \text{and} = \lambda x. \lambda y. x \ y \ \text{ff} \\
  \text{or} = \lambda x. \lambda y. x \ \text{tt} \ y
  \]

- **pred** computes the predecessor of a given natural number where the predecessor of 0 is 0.
  \[
  \text{pred} = \lambda \hat{n}. \text{fst} (\hat{n} \ \text{next} (\text{pair} \ \text{zero} \ \text{zero}))
  \]

**Question 4. [3 pts]** Define the function exp for exponentiation such that \(\exp \hat{m} \hat{n}\) evaluates to a church numeral for the product of \(n\) copies of \(m\). In other words, \(\exp \hat{m} \hat{n} \mapsto^{*} \tilde{m}^n\).

\[
\text{exp} = \text{fun} \hat{m} \hat{n} \rightarrow \text{letrec} \\
\text{pair} = \lambda x. \lambda y. (x, y) \\
\text{next} = \lambda (x, y). (y, x) \\
\text{fst} = \lambda (x, y). x \\
\text{exp} = \\
\text{return exp}
\]

---
Question 5. [3 pts] Define the function isZero = λn. · · · which tests if a given Church numeral is 0. That is, isZero 0 reduces to tt, and isZero n evaluates to ff for any non-zero number n.

isZero =

Question 6. [3 pts] Define the function eq = λm. λn. · · · which tests if two given Church numerals are equal. You may use the function isZero.

eq =
Following is the definition of de Bruijn expressions:

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

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

**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.

\[
\tau_i^n(N_1 \ N_2) = \ldots
\]

\[
\tau_i^n(\lambda. N) = \ldots
\]

\[
\tau_i^n(m) = \ldots \quad \text{if } m \geq i
\]

\[
\tau_i^n(m) = \ldots \quad \text{if } m < i
\]
Question 8. [5 pts] 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 $\tau_i^n(N)$.

\[
\sigma_n(M_1 \ M_2, N) = \quad \text{__________________________} \\
\sigma_n(\lambda \cdot M, N) = \quad \text{__________________________} \\
\sigma_n(m, N) = \quad \text{__________________________} \quad \text{if } m < n \\
\sigma_n(n, N) = \quad \text{__________________________} \\
\sigma_n(m, N) = \quad \text{__________________________} \quad \text{if } m > n
\]
4 Simply typed λ-calculus [20 pts]

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

\begin{align*}
\text{type} & : \quad A ::= P \mid A \to A \\
\text{expression} & : \quad e ::= x \mid \lambda x : A. e \mid e \ e \\
\text{value} & : \quad v ::= \lambda x : A. e \\
\text{typing context} & : \quad \Gamma ::= \cdot \mid \Gamma, x : A
\end{align*}

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

\begin{align*}
e \mapsto e' & \iff e \text{ reduces to } e' \\
\Gamma \vdash e : A & \iff e \text{ has type } A \text{ under typing context } \Gamma
\end{align*}

Question 1. [3 pts] Give the typing rules for the simply typed λ-calculus:

\begin{align*}
&
\end{align*}

Question 2. [2 pts] State the canonical forms lemma, which is necessary to prove progress:

\begin{align*}
&
\end{align*}
Question 3. [2 pts] State the inversion property, which is necessary to prove type preservation:

Theorem 4.1. (Progress).

Theorem 4.2. (Preservation).

Question 4. [4 pts] State the two theorems, progress and type preservation, constituting type safety:

Theorem 4.1. (Progress).

Theorem 4.2. (Preservation).
**Question 5. [3 pts]** Consider the extension of the simply-typed $\lambda$-calculus with sum types:

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

Complete the typing rules.

\[
\begin{align*}
\Gamma & \vdash \text{inl}_A e : \\
\Gamma & \vdash \text{inr}_A e : \\
\Gamma & \vdash \text{case } e \text{ of inl } x_1. e_1 & \mid \text{inr } x_2. e_2 : 
\end{align*}
\]
Question 6. [2 pts] Consider the extension of the simply-typed \(\lambda\)-calculus with fixed point constructs

\[
\text{expression} \quad e ::= \cdots \mid \text{fix} \, x : A. \, e
\]

Write the typing rule for \(\text{fix} \, x : A. \, e\) and its reduction rule.

\[
\Gamma \vdash \text{fix} \, x : A. \, e : \quad \rightarrow
\]

Question 7. [4 pts] Explain how to encode two mutually recursive functions \(f_1\) of type \(A_1 \rightarrow B_1\) and \(f_2\) of type \(A_2 \rightarrow B_2\) using product types.
5 Mutable references [7 pts]

Consider the following simply-typed λ-calculus extended with mutable references.

<table>
<thead>
<tr>
<th>type</th>
<th>$A$ ::= $P$</th>
<th>$A \rightarrow A$</th>
<th>int</th>
<th>ref $A$</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>base type</td>
<td>$P$ ::= bool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>$e$ ::= $x$</td>
<td>$\lambda x:A.e$</td>
<td>$e ; e$</td>
<td>let $x = e$ in $e$</td>
<td>$\text{true}$</td>
</tr>
<tr>
<td>value</td>
<td>$v$ ::= $\lambda x:A.e$</td>
<td>$()$</td>
<td>$\text{true}$</td>
<td>$\text{false}$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Question 1. [3 pts] We want to represent an array of integers as a function taking an index (of type int) and returning a corresponding element of the array. We choose a functional representation of arrays by defining type $iarray$ for arrays of integers as follows:

$$iarray = \text{ref} \ (\text{int} \rightarrow \text{int})$$

We need the following constructs for arrays:

- **new** : unit $\rightarrow$ $iarray$ for creating a new array.
  
  new () returns a new array of indefinite size; all elements are initialized as 0.

- **access** : $iarray$ $\rightarrow$ int $\rightarrow$ int for accessing an array.
  
  access $a$ $i$ returns the $i$-th element of array $a$.

- **update** : $iarray$ $\rightarrow$ int $\rightarrow$ int $\rightarrow$ unit for updating an array.
  
  update $a$ $i$ $n$ updates the $i$-th element of array $a$ with integer $n$.

Exploit the constructs for mutable references to implement $\text{new}$, $\text{access}$ and $\text{update}$. Fill in the blank:

$$\text{new} = \lambda \_\_\text{unit}. \text{ref} \; \lambda i:\text{int}.0$$

$$\text{access} = \lambda a:iarray. \lambda i:\text{int}. (!a) \; i$$

$$\text{update} = \lambda a:iarray. \lambda i:\text{int}. \lambda n:\text{int}.$$
Question 2. [4 pts] Use the constructs for mutable references to implement a recursive function fact for factorials such that fact n evaluates to n!.

fact = ________________________________
6 Symmetry of the $\alpha$-equivalence relation [15 pts]

In this problem, we prove the symmetry of the $\alpha$-equivalence relation in the untyped $\lambda$-calculus (Theorem 6.4). We use the following inference rules, where $FV(e)$ computes the set of free variables in $e$ and $[x \leftrightarrow y]e$ denotes the expression obtained by replacing all occurrences of $x$ in $e$ by $y$ and all occurrences of $y$ in $e$ by $x$.

\[
\begin{align*}
\frac{x \equiv_\alpha x}{e \equiv_\alpha e'} & \quad Var_\alpha \\
\frac{e_1 \equiv_\alpha e_1' \quad e_2 \equiv_\alpha e_2'}{e_1 \ e_2 \equiv_\alpha e_1' \ e_2'} & \quad App_\alpha \\
\frac{e \equiv_\alpha e'}{\lambda x. \ e \equiv_\alpha \lambda x. \ e'} & \quad Lam_\alpha
\end{align*}
\]

In the proof of Theorem 6.4, you may use the following lemmas on the $\alpha$-equivalence relation without proofs:

**Lemma 6.1.** $[x \leftrightarrow y]e = [y \leftrightarrow x][x \leftrightarrow y]e = e$.

**Lemma 6.2.** If $e_1 \equiv_\alpha e_2$, then $[x \leftrightarrow y]e_1 \equiv_\alpha [x \leftrightarrow y]e_2$.

**Lemma 6.3.** If $e_1 \equiv_\alpha e_2$, then $FV(e_1) = FV(e_2)$.

Complete the proof of Theorem 6.4.

**Theorem 6.4.** If $e_1 \equiv_\alpha e_2$, then $e_2 \equiv_\alpha e_1$

**Proof.**
7 (Extra-credit) Transitivity of the $\alpha$-equivalence relation

In this problem, we prove the transitivity of the $\alpha$-equivalence relation from the previous problem:

**Theorem 7.1.** If $e_1 \equiv_\alpha e_2$ and $e_2 \equiv_\alpha e_3$, then $e_1 \equiv_\alpha e_3$.

In your proof, you may use the following lemmas without proofs (Lemmas 7.2 to 7.5). Lemma 7.2 shows how two variable swappings $[p \leftrightarrow q]$ and $[x \leftrightarrow y]$ commute. Lemma 7.3 shows how a variable swapping affects the set of free variables in a given expression. Lemma 7.4 states the symmetry of the $\alpha$-equivalence relation. Lemma 7.5 is the inversion property of the $\alpha$-equivalence relation and holds because the inference rules for the $\alpha$-equivalence relation are syntax-directed.

**Lemma 7.2.** $[p \leftrightarrow q][x \leftrightarrow y]e = [[p \leftrightarrow q][x \leftrightarrow [p \leftrightarrow q]y][p \leftrightarrow q]e$.

**Lemma 7.3.**
- $x \notin FV(e)$ if and only if $[p \leftrightarrow q]x \notin FV([p \leftrightarrow q]e)$.
- $x \in FV(e)$ if and only if $[p \leftrightarrow q]x \in FV([p \leftrightarrow q]e)$.

**Lemma 7.4.** If $e_1 \equiv_\alpha e_2$, then $e_2 \equiv_\alpha e_1$.

**Lemma 7.5 (Inversion).**
- If $x \equiv_\alpha e$, then $e = x$.
- If $e_1' e_1'' \equiv_\alpha e$, then $e = e_1' e_2''$, $e_1' \equiv_\alpha e_1''$, and $e_1'' \equiv_\alpha e_2''$ for some $e_1'$ and $e_2''$.
- If $\lambda x. e_1 \equiv_\alpha \lambda x. e_2$, then $e_1 \equiv_\alpha e_2$.
- If $\lambda x. e_1 \equiv_\alpha \lambda y. e_2$ and $x \neq y$, then $y \notin FV(e_1)$, and $[x \leftrightarrow y]e_1 \equiv_\alpha e_2$.

Prove Theorem 7.1.

Hint: Perhaps the proof should proceed by rule induction on $e_1 \equiv_\alpha e_2$. You will need a lemma that shows how variable swappings affect the $\alpha$-equivalence relation. Identifying this lemma is critical to the proof of Theorem 7.1. If you introduce such a lemma, you should give a proof of it as well.