<table>
<thead>
<tr>
<th></th>
<th>Problem 1</th>
<th>Problem 2</th>
<th>Problem 3</th>
<th>Problem 4</th>
<th>Problem 5</th>
<th>Problem 6</th>
<th>Problem 7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Score</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>
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:

```sml
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 =
    | inorder' (Node (left, x, 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.

<table>
<thead>
<tr>
<th>with if/then/else</th>
<th>without if/then/else</th>
</tr>
</thead>
<tbody>
<tr>
<td>if e then false else true</td>
<td>not e</td>
</tr>
<tr>
<td>if x then true else y</td>
<td>x andalso y</td>
</tr>
<tr>
<td>if x then y else false</td>
<td>not x orelse y</td>
</tr>
<tr>
<td>if x then false else y</td>
<td>x orelse y</td>
</tr>
<tr>
<td>if x then y else true</td>
<td>not x andalso y</td>
</tr>
</tbody>
</table>
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 = ________________

    fun singleton x = ________________

    fun member s = __________

    fun insert s x = ________________

    fun remove s x = ________________

    fun union s s' = ________________
  end
```
Reductions in the $\lambda$-calculus [15 pts]

Let us abbreviate an identity function $\lambda x_i. x_i$ as $id_i$. You will show the reduction sequence of $(id_1 id_2) (id_3 (\lambda z. id_4 z))$ 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 $\lambda x_i. x_i$.

$$(id_1 id_2) (id_3 (\lambda z. id_4 z))$$

$$\rightarrow$$

$$\rightarrow$$

$$\rightarrow$$

**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 $\lambda x_i. x_i$.

$$(id_1 id_2) (id_3 (\lambda z. id_4 z))$$

$$\rightarrow$$

$$\rightarrow$$

$$\rightarrow$$

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

- (example) $\lambda x. x \equiv_\alpha \lambda y. y$

  $$\lambda x. \lambda x'. x' \ x' \equiv_\alpha \lambda x'. \underline{\phantom{x}}$$

  $$\lambda x. \lambda x'. x' \ x'' \equiv_\alpha \lambda x'. \underline{\phantom{x}}$$

  $$\lambda x. \lambda x'. x' \ x'' \equiv_\alpha \lambda x''. \underline{\phantom{x}}$$
3 Programming in the $\lambda$-calculus $[15$ pts$]$

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

\[
\begin{align*}
\hat{0} &= \lambda f. f^0 = \lambda f. \lambda x. x \\
\hat{1} &= \lambda f. f^1 = \lambda f. \lambda x. f x \\
\cdots \\
\hat{n} &= \lambda f. f^n = \lambda f. \lambda x. f f f \cdots f x
\end{align*}
\]

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

\[
\text{double} = \lambda \hat{n}. \quad \text{-------------------------------------------}
\]

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

- $\text{halve} \hat{2} \times k$ returns $\hat{k}$.
- $\text{halve} \hat{2} \times k + 1$ returns $\hat{k}$.

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

\[
\begin{align*}
\text{pair} &= \lambda x. \lambda y. \lambda b. b x y \\
\text{fst} &= \lambda p. p (\lambda t. \lambda f. t) \\
\text{snd} &= \lambda p. p (\lambda t. \lambda f. t)
\end{align*}
\]

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:

\[
\begin{align*}
\text{zero} &= \hat{0} = \lambda f. \lambda x. x \\
\text{succ} &= \lambda \hat{n}. \lambda f. \lambda x. \hat{n} f (f x)
\end{align*}
\]

Fill in the blank:

\[
\text{halve} = \lambda \hat{n}. \quad \text{-------------------------------------------}
\]
4 A weird reduction strategy [15 pts]

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

<table>
<thead>
<tr>
<th>Type</th>
<th>( A ) := ( P \mid A \rightarrow A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Type</td>
<td>( P )</td>
</tr>
<tr>
<td>Expression</td>
<td>( e := x \mid \lambda x : A . e \mid e e )</td>
</tr>
<tr>
<td>Value</td>
<td>( v := \lambda x : A . e )</td>
</tr>
</tbody>
</table>

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 \( \lambda \)-abstraction, we apply the \( \beta \)-reduction.

**Question 1.** [5 pts] Give the rules for the reduction judgment \( e \rightarrow e' \) under the weird reduction strategy. You need three rules.

\[ e_2 \rightarrow e_0 \rightarrow e_1 \rightarrow e_0 \] \[ \lambda x : A . e \rightarrow [v = x] e \]

**Question 2.** [5 pts] Give the rules for the evaluation judgment \( e \rightarrow v \) under the weird reduction strategy. You need two rules.

\[ e_1 , \! \! v \rightarrow e_2 \rightarrow v \rightarrow v \rightarrow v \]

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

\[ \kappa ::= \]
5 Substitution theorem [15 pts]

Prove the substitution theorem for the following fragment of the simply typed $\lambda$-calculus:

<table>
<thead>
<tr>
<th>type</th>
<th>$A ::= P \mid A \to A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>base type</td>
<td>$P$</td>
</tr>
<tr>
<td>expression</td>
<td>$e ::= x \mid \lambda x : A. e \mid e \ e$</td>
</tr>
<tr>
<td>typing context</td>
<td>$\Gamma ::= \cdot \mid \Gamma, x : A$</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
x : A \in \Gamma \\
\Gamma \vdash x : A \quad \text{Var} \\
\Gamma, x : A \vdash e : B \\
\Gamma \vdash x : A \to B \\
\Gamma \vdash e : A \to B \\
\Gamma \vdash e' : A \to B \\
\Gamma \vdash e : A \to B \\
\Gamma \vdash e' : A \to B \\
\end{array} \to E
\]

**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 $\Gamma, x : A \vdash e : B$. We assume that all variables in a typing context are distinct. We also assume that variable clashes never occur in the rule $\to l$. That is, $x$ in the rule $\to l$ is always a fresh variable.

Fill in the blank:

Case $y : C \in \Gamma$

$\Gamma, x : A \vdash y : C$ Var where $e' = y$

$\Gamma \vdash y : C$

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

\[\text{from } \quad \text{and the rule } \quad \text{from } x \neq y\]

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

$\Gamma \vdash e : A$

$\Gamma \vdash [e/x]x = e$

\[\text{from the assumption}\]

Case $\Gamma, x : A, y : B_1 \vdash e'' : B_2$

$\Gamma, x : A \vdash \lambda y : B_1. e'' : B_1 \to B_2$ $\to l$ where $e' = \lambda y : B_1. e''$ and $C = B_1 \to B_2$

\[\text{by IH on the premise}\]

\[\text{by the rule } \to l\]

$\Gamma \vdash \lambda y : B_1. e'' = \lambda y : B_1. [e/x]e''$

$\Gamma \vdash \lambda y : B_1. e''$

\[\text{from } \quad \text{and } \quad \]

7
Case \[ \frac{\Gamma, x : A \vdash e_1 : B \to C \quad \Gamma, x : A \vdash e_2 : B}{\Gamma, x : A \vdash e_1 \ e_2 : C} \rightarrow \text{E} \] where \( e' = e_1 \ e_2 \)

\[ \frac{\text{by IH on the first premise}}{\text{by IH on the second premise}} \]

\[ \frac{\Gamma \vdash [e/x]e_1 \ [e/x]e_2 : C}{\text{by the rule } \rightarrow \text{E}} \]

\[ \frac{\text{by the definition of substitution}}{\square} \]
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:

\[
\begin{align*}
\text{Ref} & \quad e \mapsto^* e \quad \text{Re} \\
\text{Trans} & \quad e \mapsto^* e' \mapsto^* e'' \Rightarrow e \mapsto^* e''
\end{align*}
\]

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 \______________.

Case \______________ \( \langle \quad \rangle \) where \______________

\______________ \qquad \text{assumption}

\______________ \qquad \text{from} \______________

Case \______________ \( \mapsto^* \langle \quad \rangle \)

\______________ \qquad \text{assumption}

\______________ \qquad \text{by} \______________

\______________ \qquad \text{from} \______________
7 Abstract machine C [10 pts]

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

- **type**
  - \( A \) ::= \( P \mid A \rightarrow A \)
- **base type**
  - \( P \)
- **expression**
  - \( e \) ::= \( x \mid \lambda x : A . \ e \mid e \ e \mid \text{fix } x : A . \ e \)
- **value**
  - \( v \) ::= \( \lambda x : A . \ e \)
- **frame**
  - \( \phi \) ::= \( \square e \mid v \square \)
- **stack**
  - \( \sigma \) ::= \( \square \mid \sigma ; \phi \)
- **state**
  - \( s \) ::= \( \sigma \uparrow e \mid \sigma \downarrow v \)

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

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

Fill in the blank and complete each rule:

\[
\begin{align*}
\frac{\sigma \uparrow v \mapsto_C}{\text{Val}_C} \\
\frac{\sigma \uparrow e_1 e_2 \mapsto_C}{\text{Lam}_C} \\
\frac{\sigma ; \square e_2 \downarrow v \mapsto_C}{\text{Arg}_C} \\
\frac{\sigma ; (\lambda x : A . \ e) \square \downarrow v \mapsto_C}{\text{App}_C} \\
\frac{\sigma \uparrow \text{fix } x : A . \ e \mapsto_C}{\text{Fix}_C}
\end{align*}
\]