<table>
<thead>
<tr>
<th></th>
<th>Prob 1</th>
<th>Prob 2</th>
<th>Prob 3</th>
<th>Prob 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>15</td>
<td>30</td>
<td>35</td>
<td>20</td>
<td>100</td>
</tr>
</tbody>
</table>
1 SML Programming [15 pts]

Question 1. [5 pts] Give a tail recursive implementation of \texttt{preorder} for preorder traversals of binary trees. Fill in the blank:

\begin{verbatim}
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
(* preorder : 'a tree -> 'a list *)
fun preorder t = let
    fun preorder' (Leaf x) pre = pre @ [x]
    | preorder' (Node (left, x, right)) pre = preorder' right (preorder' left (pre @ [x]))
in
end
\end{verbatim}

Question 2. [7 pts] DICT is a signature for dictionaries:

\begin{verbatim}
signature DICT =
sig
    type key
    type 'a dict
    val empty : unit -> 'a dict
    val lookup : 'a dict -> key -> 'a option
    val delete : 'a dict -> key -> 'a dict
    val insert : 'a dict -> key * 'a -> 'a dict
end
\end{verbatim}

- \texttt{key} denotes the type of keys in dictionaries.
- \texttt{'}a dict denotes the type of dictionaries for \texttt{'a} type values.
- \texttt{empty ()} returns an empty dictionary.
- \texttt{lookup d k} searches the key \texttt{k} in the dictionary \texttt{d}. If the key is found, it returns the associated item. Otherwise, it returns NONE.
- \texttt{delete d k} deletes the key \texttt{k} and its associated item in the dictionary \texttt{d} and returns the resultant dictionary \texttt{d’}. If the key does not exist in the dictionary \texttt{d}, it returns the given dictionary \texttt{d} without any modification.
- \texttt{insert d (k, v)} inserts the new key \texttt{k} and its associated item \texttt{v} in the dictionary \texttt{d}. If the key \texttt{k} already exists in the dictionary \texttt{d}, it just updates its associated item with the given item \texttt{v}.

2
Implement the functor `DictFn` which takes a `KEY` structure and generates a corresponding `DICT` structure that uses a ‘functional representation’ of dictionaries. Fill in the blank:

```ocaml
signature KEY =
  sig
    type t
    (* eq k k' : true k is equal to k' *)
    (* false otherwise *)
    val eq : t * t -> bool
  end

functor DictFn (Key : KEY) :> DICT where type key = Key.t =
  struct
    type key = Key.t
    type 'a dict = key -> 'a option

    fun empty () =

    fun lookup d k =

    fun delete d k =

    fun insert d (k, v) =

  end
```
Question 3. [3 pts] Give an implementation of IntDict whose key type is int. Fill in the blank. You may use the functor DictFn that you write in Question 2.

```ocaml
structure IntKey :> KEY where type t = int
 =
  struct

    eq (k, k') = k = k'

  end

structure IntDict = ____________________________
```
2 Inductive definitions [30 pts]

Consider the following system from the Course Notes where $s \text{ mparen}$ means that $s$ is a string of matched parentheses.

$$
\begin{array}{c}
\epsilon \text{ mparen} \quad M_{\text{eps}} \\
(s) \text{ mparen} \quad M_{\text{par}} \\
\frac{s_1 \text{ mparen} \quad s_2 \text{ mparen}}{s_1 \quad s_2 \text{ mparen}} \quad M_{\text{seq}}
\end{array}
$$

In order to show that if $s \text{ mparen}$ holds, $s$ is indeed a string of matched parentheses, we introduce a new judgment $k \triangleright s$ where $k$ is a non-negative integer:

$$
k \triangleright s \iff \begin{cases}
\text{k left parentheses concatenated with s form a string of matched parentheses} \\
\text{\underbrace{\cdots}_{k}} \text{s is a string of matched parentheses}
\end{cases}
$$

The idea is that we scan a given string from left to right and keep counting the number of left parentheses that have not yet been matched with corresponding right parentheses. Thus we begin with $k = 0$, increment $k$ each time a left parenthesis is encountered, and decrement $k$ each time a right parenthesis is encountered:

$$
0 \triangleright \epsilon \quad P_{\text{eps}} \\
\frac{k + 1 \triangleright s}{k \triangleright (s)} \quad P_{\text{left}} \\
\frac{k - 1 \triangleright s \quad k > 0}{k \triangleright s} \quad P_{\text{right}}
$$

The second premise $k > 0$ in the rule $P_{\text{right}}$ ensures that in any prefix of a given string, the number of right parentheses may not exceed the number of left parentheses. Now a judgment $0 \triangleright s$ expresses that $s$ is a string of matched parentheses.

Your task is to prove Theorem 2.1. If you need a lemma to complete the proof, state the lemma, prove it, and use it to complete the proof of Theorem 2.1.

For individual steps in the proof, please use the following format:

**conclusion**

**justification**

**Theorem 2.1.** If $s \text{ mparen}$, then $0 \triangleright s$. 

---

---

---
3 \( \lambda \)-Calculus [35 pts]

**Question 1.** [5 pts] Show the reduction sequence under the call-by-name strategy. Underline the redex at each step.

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

\[\rightarrow\]

\[\rightarrow\]

\[\rightarrow\]

\[\rightarrow\]

**Question 2.** [3 pts] Complete the definition of \( FV(e) \) that finds the set of free variables in \( e \).

\[
FV(x) = \underline{\{x\}}
\]

\[
FV(\lambda x. e) = FV(e) - \{x\}
\]

\[
FV(e_1 e_2) = FV(e_1) \cup FV(e_2)
\]

**Question 3.** [2 pts] Fill in the blank with the set of free variables of the given expression.

\[
FV(\lambda x. x) = \underline{\{\}}
\]

\[
FV(x \ y) = \underline{\{x, y\}}
\]

\[
FV(\lambda x. x \ y) = \underline{\{y\}}
\]

\[
FV(\lambda x. \lambda y. x \ y) = \underline{\{\}}
\]

\[
FV((\lambda x. x \ y) (\lambda y. x \ y)) = \underline{\{y, x\}}
\]
**Question 4. [5 pts]** This question assumes types `var` and `exp` that we have seen in Assignment 4:

```haskell
type var = string
datatype exp =
    Var of var
  | Lam of var * exp
  | App of exp * exp
```

Suppose that we have two functions `notFv` and `varSwap`:

- `notFv : var -> exp -> bool`
  - `notFv x e` returns `true` if `x` is a free variable of `e` and `false` otherwise.

- `varSwap : var * var -> exp -> exp`
  - `varSwap (x, y) e` returns `[x↔y]e`.

Below is a function `aEqual` of type `(exp * exp) -> bool` such that `aEqual (e1, e2)` returns `true` if `e1` and `e2` are α-equivalent and `false` otherwise.

```haskell
fun aEqual (Var x, Var y) = x = y
| aEqual (App (e1, e2), App (e1', e2')) = aEqual (e1, e1') andalso aEqual (e2, e2')
| aEqual (Lam (x, e), Lam (y, e')) = if x = y then aEqual (e, e')
  else if notFv x e' then aEqual (e, varSwap (y, x) e')
  else false
| aEqual _ = false
```

We write `e ≡_α e'` if `e` can be rewritten as `e'` by renaming bound variables in `e` and vice versa. Give exactly four inference rules corresponding to the above definition of `aEqual`. Use the notation `x ∉ FV(e)` for `notFv x e` and `[x↔y]e` for `varSwap (x, y) e`.

\[
\begin{align*}
\frac{}{\equiv_\alpha} \\
\frac{}{\equiv_\alpha} \\
\frac{}{\equiv_\alpha} \\
\frac{}{\equiv_\alpha}
\end{align*}
\]
Question 5. [8 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):

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

In this question, you will define three functions: sub for the subtraction operation, mod for the modulo operation, and, as an extra credit problem, div for the division operation.

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

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

\[
\begin{align*}
\text{zero} &= \hat{0} = \lambda f. \lambda x. x \\
\text{one} &= \hat{1} = \lambda f. \lambda x. f x
\end{align*}
\]

- 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 $e_1$ else $e_2$ 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.

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

- pred computes the predecessor of a given natural number where the predecessor of 0 is 0.

\[
\text{pred} = \lambda \hat{n}. \text{fst} (\hat{n} \text{ next} (\text{pair zero zero}))
\]

- eq tests two natural numbers for equality.

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

- 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:

\[
\begin{align*}
\text{tt} &= \lambda t. \lambda f. t \\
\text{ff} &= \lambda t. \lambda f. f \\
\text{and} &= \lambda x. \lambda y. x y \text{ ff} \\
\text{isZero} &= \lambda x. x (\lambda y. \text{ ff}) \text{ tt} \\
\text{next} &= \lambda p. \text{pair} (\text{snd} p) (\text{succ} (\text{snd} p))
\end{align*}
\]
Define a subtraction function $\text{sub}$ such that $\text{sub} \hat{m} \hat{n}$ evaluates to $\hat{m} - \hat{n}$ if $m > n$ and $\hat{0}$ otherwise.

$$\text{sub} = \lambda \hat{m}. \lambda \hat{n}. (\hat{n} \text{pred}) \hat{m}$$

Define a modulo function $\text{mod}$ such that $\text{mod} \hat{m} \hat{n}$ evaluates to $\hat{r}$ if $r$ is the remainder of division of $m$ by $n$. $\text{mod}$ never takes $\hat{0}$ as the second argument. Hence the result of evaluating $\text{mod} \hat{m} \hat{0}$ is unspecified. You may use the subtraction function $\text{sub}$ that you define above.

$$\text{mod} = \text{fix}(\lambda f. \lambda \hat{m}. \lambda \hat{n}. \text{if eq} \hat{m} \hat{n} \text{then zero} \text{else if eq} (\text{sub} \hat{m} \hat{n}) \text{then zero} \text{else } (f (\text{sub} \hat{m} \hat{n})) \hat{n}))$$

**Extra credit question. [10 pts]** Define a division function $\text{div}$ such that $\text{div} \hat{m} \hat{n}$ evaluates to $\hat{q}$ if $q$ is the quotient of $m$ divided by $n$. $\text{div}$ never takes $\hat{0}$ as the second argument. Hence the result of evaluating $\text{div} \hat{m} \hat{0}$ is unspecified. In this question, you are not allowed to use the fixed point combinator (and its definition), but you may use the subtraction function $\text{sub}$ that you define above.

$$\text{div} = \lambda \hat{m}. \lambda \hat{n}. \text{snd}(\hat{m}(\lambda p. \text{if eq} (\text{fst} p) \hat{n} \text{then pair zero } (\text{succ} (\text{snd} p)) \text{else if eq} (\text{sub} (\text{fst} p) \hat{n}) \text{then zero} \text{else pair} (\text{sub} (\text{fst} p) \hat{n}) (\text{succ} (\text{snd} p)))) (\text{pair} \hat{m} \text{zero}))$$

Question 7. [5 pts] Following is the definition of de Bruijn expressions:

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

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

Suppose that you are given the definition of \( \tau_i^n(N) \) 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.

Complete the definition of \( \sigma_n(M, N) \) for substituting \( N \) for every occurrence of \( n \) in \( M \) where \( N \) may include free variables.

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

\[
\sigma_n(\lambda. M, N) =
\]

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

\[
\sigma_n(n, N) =
\]

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

Question 8. [4 pts] Show the reduction of the given expression where the redex is underlined.

\[
\lambda. \lambda. \underline{(\lambda. (\lambda. 3 2 1 0) (2 1 0)) (\lambda. 0)} \quad \mapsto
\]

\[
(\lambda. (\lambda. 1) 0) (\lambda. 2 1 0) \quad \mapsto
\]
4 Simply-typed $\lambda$-calculus [20 pts]

**Question 1. [2 pts]** Consider the following simply-typed $\lambda$-calculus:

| type        | $A$ ::= bool $|$ $A \rightarrow A$ |
|-------------|-----------------------------------|
| expression  | $e ::= x | \lambda x: A. e | e e | true | false | if $e$ then $e$ else $e$ |

Write the typing rules for $x$, $\lambda x: A. e$, and $e e$:

---

$\Gamma \vdash x : A$

---

$\Gamma \vdash \lambda x: A. e$:

---

$\Gamma \vdash e e' : B$

---

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

| type        | $A ::= \cdots | A \times A$ |
|-------------|-------------------------------|
| expression  | $e ::= \cdots | (e, e) | \text{fst } e | \text{snd } e$ |

Write the reduction rules for these constructs under *lazy* reduction strategy:

---

$\rightarrow$

---

$\rightarrow$

---

$\rightarrow$

---

$\rightarrow$
Question 3. [5 pts] Consider the extension of the simply-typed $\lambda$-calculus with sum types:

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

Write the typing rules:

\[
\begin{align*}
\Gamma \vdash \text{inl}_A e : & \\
\Gamma \vdash \text{inr}_A e : & \\
\Gamma \vdash \text{case } e \text{ of } \text{inl } x_1 . e_1 | \text{inr } x_2 . e_2 :
\end{align*}
\]

Question 4. [5 pts] Consider the extension of the simply-typed $\lambda$-calculus with fixed point constructs

\[
\begin{align*}
\text{expression} & \quad e ::= \cdots | \text{fix } x : A . e
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash \text{fix } x : A . e :
\end{align*}
\]
Question 5. [5 pts] Consider the following SML program:

\[
\text{fun even } 0 = \text{true} \\
| \text{even } 1 = \text{false} \\
| \text{even } n = \text{odd } (n - 1)
\]

\[
\text{and odd } 0 = \text{false} \\
| \text{odd } 1 = \text{true} \\
| \text{odd } n = \text{even } (n - 1)
\]

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 \(\text{int} \rightarrow \text{bool} \times \text{int} \rightarrow \text{bool}\) that encodes both even and odd in the simply-typed \(\lambda\)-calculus:

\[
\begin{align*}
\text{type} & \quad A ::= \text{int} | \text{bool} | A \rightarrow A | A \times A \\
\text{expression} & \quad e ::= x | \lambda x : A. e | e \ e | (e, e) | \text{fst } e | \text{snd } e | () | \\
& \quad \text{true} | \text{false} | \text{if } e \text{ then } e \text{ else } e | \text{fix } x : A. e \\
& \quad - | = | 0 | 1 | \ldots
\end{align*}
\]

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