1. (Programming Languages, 12 pts)

(a) (4 pts) What is modularity? Briefly describe how this concept can be implemented in two different programming languages (i.e., OCaml, Ruby, Java, C, C++).

Modularity is the extent to which a program is composed of separate parts. It allows for abstraction and hiding implementation details.

<table>
<thead>
<tr>
<th>Language</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCaml</td>
<td>Modules group together related types, functions, and data. Aspects of a module exposed to the user are defined in a module signature.</td>
</tr>
<tr>
<td>Ruby</td>
<td>In addition to Ruby classes, explicit modules can be defined for implementing groups of functions and data.</td>
</tr>
<tr>
<td>Java</td>
<td>Java classes provide implementation for related functions and data, and interfaces describe available functions without including implementation.</td>
</tr>
<tr>
<td>C</td>
<td>.c files provide implementation for a related group of functions and .h files define available functions with no implementation</td>
</tr>
</tbody>
</table>

(b) (3 pts) What is the difference between dynamic and lexical scoping?

In dynamic scoping non-local variable names are bound to the closest binding in the runtime call stack. Lexical scoping binds non-local variables to the nearest binding in the source code.

(c) (5 pts) OCaml functions take only one parameter. What are the two ways of passing more than one value to an OCaml function? Briefly describe and give an example of each type of function call.

Two ways of passing multiple values:

1. Currying—function consumes one argument at a time, creating closures until all arguments are available. Example: f x y
2. Tuples—Multiple values are grouped together as a single input parameter and deconstructed using pattern matching. Example: f (x,y)
2. (OCaml Types and Type Inference, 16 pts)

(a) (5 pts) Give the types of the following OCaml expressions.

i. \[(2,3);(4,5)]\hspace{1cm} \text{TYPE: int*int list}

ii. let f x y = x + y \hspace{1cm} \text{TYPE: int -> int -> int}

(b) (6 pts) Write an OCaml expression with the following types.

i. \text{int -> int -> int list}
   \text{CODE:}
   \begin{verbatim}
   fun x -> fun y -> (x+1)::(y+1)::[]
   \end{verbatim}

ii. \text{bool -> (int -> int)}
   \text{CODE:}
   \begin{verbatim}
   fun x -> if x then fun y -> y + 1 else fun z -> z - 1
   \end{verbatim}

(c) (5 pts) Give the value of the following OCaml expressions or describe the error that exists.

i. let x = 6 in let x = x in x+6 \hspace{0.5cm} \text{VALUE: 12}

ii. (fun x -> fun y -> x * y) (5,6) \hspace{0.5cm} \text{VALUE: Error since (5,6) is type int*int but must be type int}

3. (Operational Semantics, 16 pts)

(a) (4 pts) Explain in English the meaning of the following operational semantics statement: \(\bullet, x:5 \ ; (\text{+ x 1}) \rightarrow 6\)

\begin{quote}
Given the empty environment extended with the binding \(x = 5\), the expression \((\text{+ x 1})\) evaluates to the value 6.
\end{quote}
(b) (12 pts) Using the operational semantics rules given in class, prove the following statement:

\[ \bullet; (\text{fun } x = + x 4) (+ 2 3) \rightarrow v \]

for some value \( v \). For full credit, you must show the complete tree of rule applications.

\[
\begin{align*}
\bullet; (\text{fun } x = + x 4) & \rightarrow (\bullet; \lambda x. (+ x 4)) & \text{// eval func} \\
& \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 3 & \text{// eval arg} \\
\& \rightarrow \bullet; x:5; (+ x 4) \rightarrow 9 & \text{// eval body} \\
\& \rightarrow \bullet; + 2 3 \rightarrow 5 \\
\bullet; (\text{fun } x = + x 4) (+ 2 3) & \rightarrow 9
\end{align*}
\]

4. (Lambda Calculus, 16 pts)

(a) (8 pts) Apply \( \beta \)-reduction to the following \( \lambda \)-expressions as much as possible.

i. \( (\lambda z. \lambda x. z x x) (\lambda z. z) z \)
   * \( \alpha \)-conversion: Rename \( z \) to \( a \)
   * \( \lambda a. \lambda x. a x x (\lambda z. z) z \)
   * \( \beta \)-reduction: Replace \( a \) with \( (\lambda z. z) \)
   * \( \lambda z. z x z \)
   * \( \beta \)-reduction: Replace \( x \) with \( z \)
   * \( z z \)
   * \( \beta \)-reduction: Replace \( z \) with \( z \)

ii. \( ((\lambda x. x) (\lambda y. y)) (\lambda y. y) \)
    * \( \beta \)-reduction: Replace \( x \) with \( \lambda y. y \)
    * \( ((\lambda y. y) (\lambda y. y)) (\lambda y. y) \)
    * \( \beta \)-reduction: Replace \( y \) with \( \lambda y. y \)
    * \( (\lambda y. y) (\lambda y. y) \)
    * \( \beta \)-reduction: Replace \( y \) with \( \lambda y. y \)

(b) (8 pts) Using the Church numeral encodings given below, prove that \( 0 \times 3 = 0 \)

\[
\begin{align*}
M \times N & = \lambda x. (M (N x)) \\
0 & = \lambda f. \lambda y. y \\
1 & = \lambda f. \lambda y. f y \\
2 & = \lambda f. \lambda y. f (f y) \\
3 & = \lambda f. \lambda y. f (f (f y))
\end{align*}
\]
\[ \lambda x. (M \ (N \ x)) \]
\[ \lambda x. (0 \ (3 \ x)) \]
\[ \lambda x. ((\lambda f. \lambda y. f \ (f \ y)) \ (x)) \]
\[ \lambda x. ((\lambda f. \lambda y. y) \ ((\lambda f. \lambda y. f \ (f \ y)) \ (x))) \]
\[ \lambda x. ((\lambda f. \lambda y. f) \ ((\lambda f. \lambda y. f) \ (f \ (f \ y)))) \]
\[ \lambda f. \lambda y. y = 0 \]

Replace M and N with 0 and 3
Replace 0 and 3 with encodings
\[ \beta\text{-reduction: Replace f with x} \]
\[ \beta\text{-reduction: Replace f with (\lambda y. x (x y))} \]
\[ \alpha\text{-conversion: Rename x to f} \]

5. (OCaml Programming, 40 pts)

(a) (12 pts) Functional Programming Basics
Write an OCaml function `intersect` that takes two lists `s1` and `s2` as input and returns a list that is the intersection of these two lists.

Example: `intersect [1;2;3;4] [2;4;6;8;10] = [2;4]`

You can assume that the elements of the lists can be compared using `=` or `==`, but the functions should be polymorphic. You may only use basic functional OCaml (no modules allowed!). You may also write helper functions if necessary.

```ocaml
let rec contains x s = match s with
  | [] -> false
  | h::t -> (x = h) || contains x t;;

let rec intersect s1 s2 = match s1 with
  | [] -> []
  | h::t -> if (contains h s2) then h::(intersect t s2) else intersect t s2;;
```

(b) (12 pts) Higher Order and Anonymous Functions
Using fold (given below) and an anonymous function, write a function `get_greater` that takes an int `n` and a list `l` and returns a list containing all elements of `l` that are greater than `n`.

Example: `get_greater 2 [1;2;3;4;5] = [3;4;5]`

Partial credit will be given for solutions that do not use fold.

```ocaml
let rec fold f a l = match l with
  | [] -> a
  | (h::t) -> fold f (f a h) t
```
let get_greater x l = fold (fun a h -> if h > x then h::a else a) [] l;;

(c) (16 pts) OCaml Polymorphic Datatypes

A map or associative array is a data structure comprised of a collection of key-value pairs (in Ruby this was called a hash). Consider the following OCaml polymorphic type ('a,'b) map that implements a map with keys of type 'a and values of type 'b:

```
type ('a,'b) map =
   Nil
 | Map of ('a * 'b) * ('a,'b) map;;
```

i. (3 pts) Write an OCaml expression that creates map with the following key=>value pairs:

```
1=>"foo"
2=>"bar"
```

Map ((1, "foo"), Map ((2, "bar"), Nil))

ii. (5 pts) Write an OCaml function keys that takes a map and returns a list of the map’s keys.

```
let rec keys m = match m with
   Nil ->[]
 | Map((k,..),t) -> k::(keys t);;
```

iii. (8 pts) Write an OCaml function put that takes a key-value pair (k,v) and a map m and returns a new map with (k,v) added to m. If the key k already exists, then put will just update the value. You can use helper functions if necessary.

```
let rec put_helper (x,y) a m = match m with
   Nil -> Map((x,y),a)
 | Map((k,v),t) -> if k = x then put_helper (x,y) a t else put_helper (x,y) (Map((k,v),a)) t;;

let put (x,y) m = put_helper (x,y) Nil m;;
```