# CMSC330 Spring 2011 Midterm #2 - SOLUTION

**Name** _____

**Discussion Time (circle one): 9am 10am 11am 12pm 1pm 2pm**

**Do not start this exam until you are told to do so!**

**Instructions**

- You have 75 minutes to take this midterm.
- This is a closed-book exam. No notes or other aids are allowed.
- If you have a question, please raise your hand and wait for the instructor.
- Answer essay questions concisely using 2-3 sentences. Longer answers are not necessary and a penalty may be applied.
- To be eligible for partial credit, show your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

| | Problem | Score |
|---|---|---|
| 1 | Parsing | /20 |
| 2 | OCaml: types | /20 |
| 3 | OCaml: functions | /20 |
| 4 | Scoping, parameters | /20 |
| 5 | OCaml programming | /20 |
| | Total | /100 |

1. (20 pts) **Parsing**
   a. (4 pts) Explain the difference between top-down and bottom-up parsing.

   **In top-down parsing, a parse tree is constructed starting from the root; new nodes are added as children of existing nodes based on the productions whose left-hand sides match an existing node. In bottom-up parsing, a parse tree is constructed starting from the leaves; new nodes are added based the matching of the right-hand side of a production against an existing sequence of nodes.**

   b. (4 points) What is "predictive parsing"? Is recursive-descent parsing predictive?

   **Predictive parsers use a lookahead to determine which productions to apply next. Recursive-descent parsing as covered in class is predictive.**

   The next parts of this question refer to the following CFG, which defines a language of anonymous functions and applications of functions.

   E → *x* | fun *x* = E | E E | (E)

   The terminals in the grammar include all (single-letter) identifiers *x*, the keyword `fun`, and the symbol =, ( and ). The first rule says that any identifier is an expression; the second says that an anonymous function with parameter *x* and body E is an expression; the third says that an application is an expression; and the last says that parentheses may be used.

   c. (8 pts) Compute the first sets for the right-hand sides of each of the productions in the CFG.

   **First (x) = {x}**
   **First (fun x = E) = {fun}**
   **First (E E) = {x, fun, ( }**
   **First ( (E) ) = { ( }**

   d. (4 pts) Is this grammar a candidate for recursive descent parsing? Why or why not?

   **No, it is not. There are two reasons. First, the First sets of two different productions overlap. Second, the grammar is left-recursive.**

2. (20 pts) **OCaml and types**

a. (12 pts) Give types that OCaml would compute for each of the following expressions, or write ERROR if OCaml would report a type error. In what follows `map` and `fold` refer to the curried function definitions given in class.

```
[]
```
**TYPE = 'a list**

```
map (fun x -> x+1) []
```
**TYPE = int list**

```
let f x y = x::y in
  f (x,y)
```
**TYPE = ERROR**

```
let f x y = fun y -> y in
  f 0 0 0
```
**TYPE = int**

```
let f g x = x g in
  f
```
**TYPE = 'a -> ('a -> 'b) -> 'b**

```
fold (fun a h -> a + !h)
```
**TYPE = int -> (int ref list) -> int**

b. (3 pts) What is a "signature" in OCaml?

**A signature is an interface specification for a module. It contains definitions of types and specifications of the names of values and their types. (For grading purposes, we allowed "A signature is a type", even though technically it is different from a type.)**

c. (5 points) Give an OCaml data type declaration for a type `Prop` of propositions. Your type should include constructors `Var` (for variables), which takes a string as an argument, a constructor `Not` (for negation), which may be applied to single proposition, and constructors `And`, `Or` and `Implies`, which are applied to two propositions. Here are some example propositions, and the corresponding values that should be in your data type.

| Proposition | Corresponding Data-Type Value |
|---|---|
| $p \vee q$ | `Or (Var "p", Var "q")` |
| $p \rightarrow \neg q$ | `Implies (Var "p", Not (Var "q"))` |

```
type prop =
      Var of string
    | Not of prop
    | Or of prop * prop
    | And of prop * prop
    |  Implies of prop * prop
```

3. (20 pts) **OCaml functions**
   a. (5 pts) Write an OCaml expression having the following type:
   ```
   ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
   ```

   **fun f -> fun g -> fun a -> g (f a)**

   b. (5 pts) Consider the following type of binary trees in OCaml
   ```
   type 'a binTree =
       NullT
     | Node of ('a * 'a binTree * 'a binTree)
   ```

   Write function `treeMap` of type
   ```
   ('a -> 'b) -> 'a binTree -> 'b binTree
   ```
   that, given function f and tree t, returns the tree resulting from applying f to every node in t.

**let rec treeMap f = function**
**NullT -> Null T**
**| Node of (x, l, r) -> Node (f x, treeMap f l, treeMap f r)**

   c. (5 pts) Consider the binary tree type in part b above, and the following fold operation on these trees:

```
let rec treeFold f a t =
  match t with
    NullT -> a
  | Node (n,lt,rt) -> treeFold f (treeFold f (f a n) lt) rt
```

   Using `treeFold`, define a function sumT of type `int binTree -> int` that returns the sum of all the integer labels in a tree.

**let sumT = treeFold (+) 0**

   d. (5 pts) Explain how closures in OCaml may be used to implement objects.

**The environment component of a closure may be used to store private instance variables that are only accessible by calling the closure. An object with private instance variables may then be constructed by implementing methods as closures that contain these instance variable in their environments.**

4. (20 pts) **Scope rules and parameter passing**

Consider the following OCaml declarations.

```
let f x y = x + y;;
let g y = f 1;;
let x = 0;;
```

a. (4 pts) In the closure that is associated with `g`, what value is assigned to `x`?

**1 (This question was discarded due to typo: should have been let g = f 1;;)**
2

b. (4 pts) Suppose OCaml uses dynamic scoping instead of static scoping. What value would be returned by evaluating the expression `(g 2)`?

**2 (This question was also discarded due to typo in g.)**

Parts c and d refer to the following declarations

```
let x = ref 0;;
let inc x = (x := !x + 1; !x);;
let g x = 0;;
```

c. (4 pts) Under the usual OCaml parameter-passing semantics, what is the value of expression `!x` after evaluating the call `g (inc x)`?

**1**

d. (4 pts) Now suppose that OCaml uses call-by-name parameter passing. What would the value of expression `!x` after evaluating the call `g (inc x)`?

**0**

e. (4 pts) Consider the following declarations.

```
let x = ref 0;;
let incX () = (x := !x + 1; !x);;
let x = ref 0;;
```

Suppose a user now evaluates the expression `incX ()`. What would the value of expression `!x` be afterwards?

**0**

5. (20 pts) **OCaml programming**
In this question you will be asked to write OCaml functions implementing different operations on *bit vectors*. Traditionally, bit vectors are sequences of bits. In this problem, bit vectors will be lists of booleans:

```
type bitVector = bool list;;
```

a. (5 pts) Write an OCaml function `vand` of type `bitVector -> bitVector -> bitVector` that implements a bitwise conjunction operation on bit vectors. Thus, `vand [true; false] [true; true]` should return `[true; false]`. If the arguments are of different lengths, an exception should occur.

```
let rec vand v1 v2 =
  match (v1, v2) with
    ([], []) -> []
  | (h1::t1, h2::t2) -> (h1 && h2)::(vand t1 t2);;
```

b. (5 pts) Write an OCaml function `std` of type

```
bool -> int -> bitVector -> bitVector
```

that standardizes the length of a bit vector by removing or adding elements at the end. Specifically, `std b len l` returns a list containing exactly `len` elements that is constructed from `l` as follows: if `l` is too long, elements at the end of `l` are removed; if `l` is too short, enough copies of Boolean `b` are added to make the resulting list the correct length. Thus, `std true 2 [true; false; true]` should return `[true; false]`, while `std true 3 [false]` should return `[false; true; true]`.

```
let rec std (b : bool) i v =
 if i <= 0 then []
 else match v with
    [] -> b::(std b (i-1) [])
  | (h::t) -> h::(std b (i-1) t);;
```

c. (5 pts) Vectors in general, and bit vectors in particular, are often stored in a so-called *sparse* representation. In the case of bit vectors, this sparse form records two pieces of information: the length of the vector, and the positions (in *sorted order*) of the bits that are true. So, for example, the sparse representation of `[true;false;false]` would be the tuple `(3,[0])`, since the bit vector has three components and only the bit at position 0 is true. Likewise, `[false;true;false;true]` would be represented as `(4,[1;3])`, since the vector has length four, with the bits at positions 1 and 3 being true.

Write a function, `mkSparse`, that takes a bit vector as input and returns a sparse representation as output. Your function must have type `bitVector -> (int * (int list)`. Note that the integer list that is part of your output must be sorted in increasing order. You may *not* use any OCaml library functions in your code!

```
let rec fold f a = function
      [] -> a
  |  (h::t) -> fold f (f a h) t

let rev = fold (fun a -> fun h -> h::a) [];;

let mkSparseHelper = fold (fun (i, l) -> fun h -> (i+1, if h then i::l else l)) (0,[]);;

let mkSparse v =
   let (i, l) = mkSparseHelper v in (i, rev l);;
```

d. (5 pts) Using the sparse representation in part c, write a function `sparseLookup` that, given a bit vector in sparse representation and an index `i`, returns the value of the bit at position `i` in the bit vector. Your function must have the following type:

```
(int * (int list)) -> int -> bool
```

For example, `sparseLookup (4,[1;3]) 2` should evaluate to `false`, while `sparseLookup (4,[1;3]) 3` should evaluate to `true`.

Your function does not need to do any error checking. You may assume that list of integers passed in as part of the first argument is sorted. You may *not* use any OCaml library functions in your code!

```
let rec sparseLookup (i, l) j =
  match l with
     [] -> false
   | (h::t) -> if j < h then false
               else if j = h then true
               else sparseLookup (i, t) j
```