CMSC330 Spring 2011 Midterm #2

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.

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

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

   $$E \rightarrow x \mid \text{fun } x = E \mid E\ E \mid (E)$$

   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.

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

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 =**
`map (fun x -> x+1) []`	**TYPE =**
`let f x y = x::y in` `  f (x,y)`	**TYPE =**
`let f x y = fun y -> y in` `  f 0 0 0`	**TYPE =**
`let f g x = x g in` `  f`	**TYPE =**
`fold (fun a h -> a + !h)`	**TYPE =**

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

   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"))`

3. (20 pts) **OCaml functions**

a. (5 pts) Write an OCaml expression having the following type:

```
('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

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.

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.

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

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`?

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

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)`?

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)`?

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?

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.

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]`.

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!

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!