STUDENT LAST NAME: ______________________________

STUDENT FIRST NAME: ______________________________

STUDENT NUMBER: ____________________

Faculty of Science
Midterm Examination

COMPUTER SCIENCE COMP 302
Programming Languages and Paradigms

Examiner: Prof. Brigitte Pientka 4 October 2018

6:00pm to 7:30pm

Instructions:
This exam has 5 questions. Please answer all questions. This is a **closed book exam**: You may use a **cheat sheet of one page (written back and front)**. You may **not** use calculators, computers, or electronic aids of any kind. Please answer all questions **on the question paper itself** and return it at the end.

This exam has 7 pages, including the cover page.

Q 1	**Q 2**	**Q 3**	**Q 4**	**Q 5**	**Total**
6	**2**	**7**	**4**	**6**	**25**

Question 1: (6 points) There are different types.

OCaml is a statically typed programming language and it forces programmers to think about whether their programs type check. Consider the following examples and choose from each column the correct answer. For example, to choose option a), write as answer: a.

i. Expression `fun (x, y) -> if x then 2.5 else y`

has the most general type

			Answer
a) `float`	b) `bool -> float -> float`	c) `bool * float -> float`	**c**

ii. Expression `fun (x,y) -> x :: y`

has the most general type

			Answer
a) `'a * 'a list -> 'a list`	b) `'a -> 'a list -> 'a list`	c) `'a list`	**a**

iii. Expression `let combine f g = function x -> f (g x)`

has the most general type

			Answer
a) `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`	b) ill typed	c) `('a -> 'a) -> ('a -> 'a) -> 'a -> 'a`	**a**

iv. Expression `let rec f x = f (f x) in f true`

has the most general type

			Answer
a) `bool`	b) ill typed	c) stack overflow looping recursion?	**a**

v. Expression `let test f x = if f 3 > f 2 then f x else f`

has the most general type

			Answer
a) `int`	b) ill typed	c) `('a -> 'a) -> 'a -> 'a`	**b**

.vi Expression

```
let inc_head = fun (h::t) -> h+1 in inc_head [];;
```

has most general type

			Answer
a) `int`	b) `Error -- Nonexhaustive Match`	c) `int list -> int`	**a**

Question 2: (2 points) It's all about evaluation.

i. Consider the following OCaml program.

```ocaml
let point = (3, 2)
let component_x (x,y) = x
let component_y (x,y) = y

let shift_point n = (component_x (point) + n , component_y (point) + n)

let point = shift_point 1
```

After loading the above program, what will happen when we evaluate `shift_point 2`?

			Answer
a) it returns `(5, 4)`	b) it returns `(6,5)`	c) it returns `(6,5)` and updates the variable `point` with the new value `(6,5)`	**a**

ii. Consider the following OCaml program:

```ocaml
let initial_list = []
let rec snoc l x = match l with
 | []   -> [x]
 | h::t -> snoc t x
```

After loading the above program, what will happen when we evaluate

```ocaml
let l = snoc initial_list 3 in (initial_list, snoc l 4)
```

			Answer
a) it returns `([] , [3;4])`	b) it returns`([3] , [3;4])` and updates the variable `initial_list` with the new value `[3]`	c) it returns `([] , [4])`	**c**

Question 3: (7 points) Higher-Order Functions

In the homework, we have represented numbers using higher-order functions. In this question, we will represent booleans as Alonzo Church did using higher-order functions. We call them "Church booleans".

Examples of Church booleans:

`ttrue`	is represented as	`(fun x y -> x)`
`ffalse`	is represented as	`(fun x y -> y)`

We now consider some simple boolean operations on Church booleans.

NOTE When defining the boolean operations on Church booleans you should only use variables and function calls. **Do not first convert a Church Boolean to a boolean value in OCaml and then use the built-in boolean operations in OCaml.**

3 points Write a function `not` which when given `ttrue` (i.e. `fun x y -> x`) will return `ffalse` and when given `ffalse` (i.e. `(fun x y -> y)`) will return `ttrue`.

There are several good answers, e.g.

```
let not c = fun x -> fun y -> c y x

let not c = c ffalse ttrue
```

4 points Write a function `conj` that takes in a tuple of Church booleans and implements the truth table below.

Inputs		Output
`ttrue`	b	b
`ffalse`	b	`ffalse`

There are several good answers, e.g.

```
let conj (c1, c2) = fun x -> fun y -> c1 (c2 x y) y

let conj (c1, c2) = c1 c2 ffalse
```

Question 4: (4 points) Get Out The `map`

The *exterior product* of two vectors is defined as the matrix formed by all pairwise products of elements of the two vectors. In mathematical notation: if a is an n-dimensional vector and b is an m-dimensional vector then the exterior product of a and b is written $a \otimes b$ and is defined by

$$(a \otimes b)[i, j] = a[i] * b[j].$$

Clearly $a \otimes b$ is an n by m matrix.

We will represent vectors with lists, and we assume that the entries are integers (for simplicity). Write an OCaml function `eprod` that computes the exterior product of two vectors and produces a list of lists of integers as result. Assume that the lists are never empty.

```
# eprod [1;2;2;1] [3;5;1];;
- : int list list =
[[3; 6; 6; 3]; [5; 10; 10; 5]; [1; 2; 2; 1]]
# eprod  [3;5;1] [1;2;2;1];;
- : int list list = [ [3; 5; 1]; [6; 10; 2]; [6; 10; 2]; [3; 5; 1] ]
# eprod [1] [1;2;2;1];;
- : int list list = [[1]; [2]; [2]; [1] ]
```

Fill in the blanks, using the higher-order function `List.map` as many times as needed; do not use other helper functions. Recall that the type of `List.map` is

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

```
let rec eprod a b =

  List.map

    (fun x -> List.map (fun y -> y * x) a)

  b
```

If you do not understand how to use the template code, feel free to write down your solution in the scratch space (page 6) with a note below.

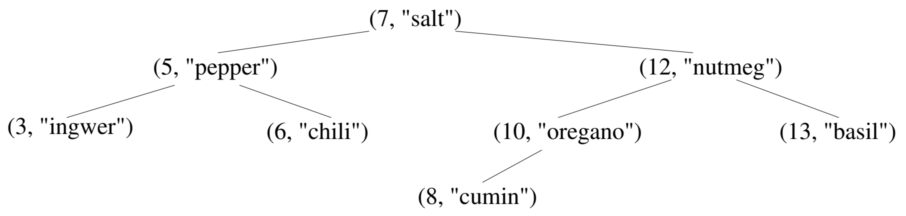
Question 5 [6 points]: Spicing things up

We consider here binary search trees which are defined as follows:

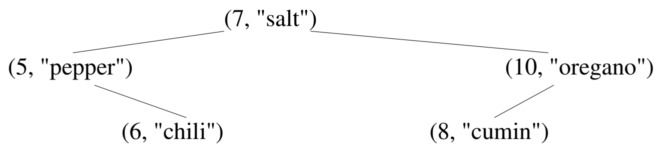
```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

Write a function `subtree: (int * 'a) tree -> (int * int) -> (int * 'a) tree` which takes as input a binary search tree `t`, where we label each node with a key and data, together with an interval `(lo, hi)` and removes from `t` all nodes whose keys fall outside the interval. Recall that a binary search tree is sorted.

Here is an example:



Keeping the tree with nodes with keys between 5 and 10 we obtain:



To compare whether a key is within an interval, use the following function `compare_interval`:

```
type interval_order =  Within | Less | Greater

let compare_interval x (lo, hi) =
  if x < lo then Less  else (if x > hi then Greater else Within)
```

There are two different approaches. The first uses direct recursion:

```
  let rec subtree t (lo, hi) = match t with
  | Empty -> Empty
  | Node ((k, v), l, r) ->
    match compare_interval k (lo, hi) with
    | Within -> Node ((k, v), subtree l (lo, hi), subtree r (lo, hi))
    | Less -> subtree r (lo, hi)
    | Greater -> subtree l interval
```

The second, using `tree_fold`, is on the following page.

```
let subtree t interval =
  tree_fold
    (fun ((k, v), l, r) ->
      match compare_interval k interval with
      | Within -> Node (x, l, r)
      | Less -> r
      | Greater -> l)
    t
    Empty
```