Comp 302 Final

Concepts

```
let curry f = (fun x y -> f (x, y))
let curry2 f x y = f (x, y)
let curry3 = fun f -> fun x -> fun y -> f (x, y)
let uncurry f = (fun (x, y) -> f x y)

(* Functions are right associative *)
(* Functions are not evaluated until they need to be *)
let test a b = a * a + b
test 3 = fun y -> 3 * 3 + y (* Not 9 + y *)
```

Syntax

Do not forget about 'rec', 'let ... in', brackets, constructors or tuples

```
match x with
  | a -> (* return *)
  | b -> (* Nested matching *)
    begin match ... with
      | ... ->
    end
  | _ -> (* wildcard return *)

let name arg1 arg2 =
  let inner' arg1' arg2' = out' in
  inner' arg1 arg2

exception Failure of string
raise (Failure "what_a_terrible_failure")

(* ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun> *)
let cur = fun f -> fun x -> fun y -> f (x,y)

(* 'a list list -> 'a list = <fun> *)
let first lst = match lst with
  | [] -> []
  | x::xs -> x

(* An anonymous 'function' has only one argument,
and can be matched directly without match ... with
val is_zero : int -> string = <fun> *)
let is_zero = function | 0 -> "zero" | _ -> "not_zero"

(* Variable bindings are overshadowed;
bindings are valid in their respective scopes *)
let m = 2;; let m = m * m in m (* is 4 *);;
m (* is 2 *);; let f () = m;; let m = 3;; f () (* is 2 *);;
```

List Ops

```
elem :: list          list1 @ list2
val length : 'a list -> int
val filter : ('a -> bool) -> 'a list -> 'a option
val map : ('a -> 'b) -> 'a list -> 'b list
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val rev : 'a list -> 'a list
val init : int -> (int -> 'a) -> 'a list (* by index *)
```

Types & Option

```
(* Base types: bool, int, char, float, 'a list, option *)
(* 'x denotes a polymorphic type (Java Generics) *)
type 'a option = None | Some of 'a
(* Constructors can be used to match within types
match cases are sufficient once all constructors are matched *)
type rational = Integer of int
  | Fraction of rational * rational
type 'param int_pair = int * 'param
let x = (3, 3.14) (* val x : int * float = 3, 3.14 *)
(* Valid specified type *)
let (x : int_pair) = (3, 3.14) (* val x : int_pair = 3, 3.14 *)
```

Higher Order Functions

```
(* sum : (int -> int) -> int * int -> int *)
let rec sum f (a, b) =
  if a > b then 0
  else f a + sum f (a + 1, b)
(* sumCubes : int * int -> int = <fun> *)
let sumCubes (a, b) = sum (fun x -> x * x * x) (a, b)
```

Induction

$e \Downarrow v$	multi step evaluation from e to v
$e \Rightarrow e'$	single step evaluation from e to e'
$e \Rightarrow^* e'$	multiple small step evaluations from e to e'

State theory and IH; do base case

```
let rec even_parity = function
  | [] -> false
  | true::xs -> not (even_parity xs)
  | false::xs -> even_parity xs

let even_parity_tr l = let rec parity p = function
  | [] -> p | p'::xs -> parity (p<>p') xs in
  parity false l

(* IH: For all l, even_parity l = even_parity_tr l *)
(* Case for true: *)
even_parity_tr true::xs
= parity false true::xs          (* Def of even_parity_tr *)
= parity (false <> true) xs      (* Def of parity *)
= parity true xs                 (* Def of <> *)
= not (parity false xs)          (* Prove? *)
= not (even_parity_tr xs)        (* Def of even_parity_tr *)
= not (even_parity xs)           (* IH *)
= even_parity true::xs           (* Def of even_parity *)
```

```
module type STACK =                                              hi
sig
    type stack
    type t
    val empty : unit -> stack
    val push : t -> stack -> stack
    val size: stack -> int
    val pop : stack -> stack option
    val peek : stack -> t option
end

module IntStack : (STACK with type t = int) =
struct
    type stack = int list
    type t = int
    let empty () = []
    let push i s = i :: s
    let size = List.length
    let pop = function | [] -> None | _ :: t -> Some t
    let peek = function | [] -> None | h :: _ -> Some h
end

(* val double : ('a -> 'a) -> 'a -> 'a = <fun> *)
let double = fun f -> fun x -> f(f(x))

(* Susp *)

type 'a susp = Susp of (unit - 'a)
let delay f = Susp(f)      (* (unit -> 'a) -> 'a susp *)
let force (Susp f) = f()   (* 'a susp -> 'a *)

(* ('a -> 'b -> 'c) -> 'a str -> 'b str -> 'c str *)
let rec zip f s1 s2 = {hd = f s1.hd s2.hd ;
     tl = delay (fun () -> zip f (force s1.tl) (force s2.tl)) }

(* Coin *)

exception BackTrack

(* val change : int list -> int -> int list = <fun> *)
let rec change coins amt = if amt = 0 then []
    else (match coins with
        | [] -> raise BackTrack
        | coin :: cs ->
            if coin > amt then change cs amt
            else try coin :: (change coins (amt - coin))
                with BackTrack -> change cs amt)

(* val change : int list -> int ->
     (int list -> 'a) -> (unit -> 'a) -> 'a = <fun> *)
let rec change coins amt success failure =
     if amt = 0 then success []
    else match coins with
        | [] -> failure ()
        | coin :: cs ->
            if coin > amt then change cs amt success failure
            else change coins (amt - coin)
                (fun list -> success (coin :: list ))
                (fun () -> change cs amt success failure)
```