1. Impure Programming

1. In pure functional OCaml, the only way to have a function with an unconstrained return type is to loop forever, i.e.

   ```
   # let rec f x = f x;;
   ```

   But non-pure OCaml adds another way to do this, what is it?

   [small]

2. Doubly-linked lists are lists which have pointers to the previous and next elements, give a type for doubly-linked lists, and the insertion, fold-left and fold-right functions on them. Make both fold-left and fold-right tail-recursive.

   Note, you will need to look at == operator in OCaml.

   [medium]

3. Discuss how you can detect cycles, and the time and space complexity of this.

   [small/medium]

4. It is possible to emulate cyclic lists in OCaml using pure programming too, explain how and contrast it with the impure way.

   [small/medium]

5. Why does OCaml not allow one to write `exception Poly of 'a`?

   [small]

2. Lazy lists

1. Write a datatype for lazy lists/streams, and a function to create an infinite stream of integers.

   [small]

2. Write a function to filter the stream.

   [small]

3. You can define the prime number sequence as taking an integer, and removing all multiples of that integer from the rest of the sequence. (Starting at two, of course.) Code the lazy list of primes.

   [medium/big]

3. Big question

This question is driven at making a standalone program, for which I'll help write part of it. First, some type definitions for the rest of the question:

```ocaml
type var_id = int
type term =
  | Var of var_id
  | List of term list
  | Int of int
  | Bool of bool
  | Str of string;;
```

```ocaml

type subst = (var_id * term) list
let rec lookup (term: term) (subst: subst) = match term, subst with
  | Var v, (w, wval)::wvs ->
    if v=w then wval else lookup (Var v) wvs
  | _ -> term
let extend (varid: var_id) (value: term) (subst: subst) =
  (varid, value)::subst

type state = subst * var_id;;
type stream = state list;;
type goal = state -> stream;;

(* Some helper functions to abstract away using lists for the stream *)
let fail: goal = fun (s, c) -> [];;
let success: goal = fun (s, c) -> [(s, c)];;
let map = List.map;;
let join = String.concat;;
```

1. Some of the past exam questions give you some code, that has been obfuscated, and ask you to explain it, or make modifications to it. Consider the following program, in which the names have been rewritten by a hungry supervisor:

```ocaml
exception Out_of_ingredients
let rec cook pantry pizza pasta =
  let deep_dish = lookup pizza pantry
  in let ravioli = lookup pasta pantry
  in match deep_dish, ravioli with
  | Var pepperoni, Var cheese when pepperoni=cheese -> pantry
  | Var pepperoni, _ -> extend pepperoni ravioli pantry
  | _, Var cheese -> extend cheese deep_dish pantry
  | List(flour::base), List(tomato::sauce) ->
      cook (cook pantry flour tomato) (List base) (List sauce)
  | x, y when x=y -> pantry
  | _ -> raise Out_of_ingredients;;
```

   Describe what it does. You might want to rewrite it using better names. Also, a good place to start is putting type annotations in places.

   [small/medium]

2. Using that code, we have the following function:

```ocaml
let eq a b =
  fun (s, c) ->
    try
      let s' = cook s a b
      in success (s', c)
    with Out_of_ingredients -> fail (s, c);;
```

Can you rewrite `cook` (or your better named version), and `eq` to not use exceptions?

[small]

3. For the last part of the question, we have some more definitions, and the `var_id` in the state becomes obvious – it represents the first free variable (hence why we are using integers for variable identifiers, rather than say strings.

```ocaml
let fresh (f: term -> goal) : goal =
  fun (s, c) -> f (Var c) (s, c + 1);;
let mplus (a: stream) (b: stream) : stream = a @ b;;
let rec bind (s: stream) (g: goal) : stream =
  match s with
  | [] -> []
  | sub::subs -> mplus (g sub) (bind subs g);;

let disj (g1: goal) (g2: goal) : goal =
  fun state -> mplus (g1 state) (g2 state);;

let conj (g1: goal) (g2: goal) : goal =
  fun state -> bind (g1 state) g2;;
```

This logic programming language returns a list of all the possible results at once, discuss how you could make it not do so, and how you can then change the search strategies.

To help with this question, I have included some examples of use. For a simple example, we have the code:

```ocaml
let test =
  let query =
    fresh @@ fun p -> fresh @@ fun q ->
      let l1 = List [p; Int 7]
      in let l2 = List [Int 9; Int 7]
      in disj
          (conj
            (eq q (Int 5))
            (eq l1 l2))
          (eq q (Int 1))
  in query ([], 0);;
```

which returns

```ocaml
val test : stream = [([(0, Int 9); (1, Int 5)], 2);
                     ([(1, Int 1)], 2)];;
```

and this means that there are two solutions to the logic expression

$$(q = 5 \land [p, 7] = [9, 7]) \lor q = 1$$

either `p` is 9 and `q` is 5, or `q` is 1 and it doesn't matter what `p` is.

To give another example, which will help for the search strategy part, consider representing a kind of binary tree where only the leaves hold values, using list terms, for example:

```ocaml
let t1 = (List [List [List [Int 1; Int 2]; List [Int 3; Int 4]];
                List [List [Int 5; Int 6]; List [Int 7; Int 8]]]);;
```

The expression to walk all nodes (leaves and branches) of the tree can be written as follows.

```ocaml
(* Walk a nested list *)
let rec walk term out =
  disj
    (fresh @@ fun l -> fresh @@ fun r ->
      conj
        (eq term (List[l; r]))
        (disj (walk l out) (walk r out)))
    (eq term out);;
```

For convenience, I will also define some pretty printing functions, you don’t necessarily need to understand them though they are not complicated, just a bit tedious.

```ocaml
(* Walks substitutions recursively until no more variables can be
   resolved -- assumes no recursive substitutions *)
let rec make_ground subst term : term = match term with
  | Var v -> (match (List.assoc_opt v subst) with
    None -> term | Some t -> make_ground subst t)
  | List l -> List (List.map (make_ground subst) l)
  | t -> t;;

(* Formats a term *)
let rec string_of_term = function
  | Var v -> Printf.sprintf "V%d" v
  | Int i -> Printf.sprintf "%d" i
  | Str s -> String.escaped s
  | Bool b -> if b then "true" else "false"
  | List l -> "[" ^ String.concat "; " (List.map string_of_term l) ^ "]";;

let print_all_mappings v streams =
  streams
  |> map (fun (subst, _) -> string_of_term @@ make_ground subst v)
  |> join "\n"
  |> print_endline;;
```

Finally we can walk the tree and print the variable binding to the second argument of `walk`.

```ocaml
walk t1 (Var 0) ([], 1)
|> print_all_mappings (Var 0);;
```

```
1
2
[1; 2]
3
4
[3; 4]
```

```
[[1; 2]; [3; 4]]
5
6
[5; 6]
7
8
[7; 8]
[[5; 6]; [7; 8]]
[[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]]
```

[medium]

3.1. Bonus

This uses code from the "Big question" section, optionally along with the lazy list version, to make a type checker for the simply typed lambda calculus. First, we will need some convenience functions.

```ocaml
let any (g: goal list) : goal =
  List.fold_left disj fail g;;

let all (g: goal list) : goal =
  List.fold_left conj success g;;
```

The simply typed lambda-calculus has the typing rules

$$\frac{\Gamma, \text{arg} : \alpha \vdash \text{body} : \beta}{\Gamma \vdash \text{fn arg} \to \text{body} : \alpha \to \beta}\,(\text{fun})$$

$$\frac{\Gamma \vdash \text{func} : \alpha \to \beta \qquad \Gamma \vdash \text{arg} : \alpha}{\Gamma \vdash \text{func arg} : \beta}\,(\text{app})$$

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha}\,(\text{var. found})$$

$$x \neq y\,\frac{\Gamma \vdash x : \beta}{\Gamma, y : \alpha \vdash x : \beta}\,(\text{var. not found})$$

Which we can easily express as a series of logic rules in our simple logic programming system.

```ocaml
let rec typecheck gamma term typ =
  any [
    (* fun *)
    (fresh @@ fun var -> fresh @@ fun body ->
      fresh @@ fun a -> fresh @@ fun b ->
      all [
        eq term @@ List[Str "fun"; var; body];
        eq typ @@ List[a; Str "->"; b];
        typecheck ((var, a)::gamma) body b]);
    (* app *)
    (fresh @@ fun fn -> fresh @@ fun arg ->
      fresh @@ fun a -> fresh @@ fun b ->
      all [
        eq term @@ List[fn; arg];
        eq typ b;
        typecheck gamma fn (List[a; Str "->"; b]);
        typecheck gamma arg a]);
    match gamma with
    | [] -> fail
    | (x, a)::rest ->
      any [
        (* var found *)
        all [eq x term; eq a typ];
        (* var not found *)
        typecheck rest term typ]
  ];;
```

We will also write a couple of functions to pretty-print the resulting type, to make it easier to read.

```ocaml
(* Creates a module for mapping variables (integers) *)
module VarMap = Map.Make(struct include Int end);;

(* Maps a number to a type, e.g. 0 to 'a, 25 to 'z, 26 to 'aa, 27 to 'ab etc *)
let format_var n =
  let char_of_var n = Char.chr (Char.code 'a' + n)
  in let rec inner n acc =
    if n < 26
    then char_of_var n :: acc
    else inner (n / 26 - 1) @@ char_of_var (n mod 26) :: acc
  in "'" ^ (String.of_seq @@ List.to_seq @@ inner n []);;

(* Map each variable to 'a, 'b, etc *)
let map_var vars v = match VarMap.find_opt v vars with
  | None ->
    let mapping = format_var (VarMap.cardinal vars)
    in (VarMap.add v mapping vars, mapping)
  | Some s -> (vars, s);;

(* Formats a type (which is assumed to be only `List[type; Str"->" type]` or
   `Var v`) as a readable string *)
let rec format_type vars term = match term with
  | List[List l; Str "->"; r] ->
    let (vars', l') = format_type vars (List l)
    in let (vars'', r') = format_type vars' r
    in (vars'', Printf.sprintf "(%s) -> %s" l' r')
  | List[l; Str "->"; r] ->
    let (vars', l') = format_type vars l
    in let (vars'', r') = format_type vars' r
    in (vars'', Printf.sprintf "%s -> %s" l' r')
  | Str s -> (vars, s)
  | Var v -> map_var vars v
  | _ -> failwith "format_type";;

let format_results query results =
  map (fun ((subst, _): state) ->
    let (_, ty) = format_type VarMap.empty (make_ground subst query)
    in ty) results
  |> join "\n" |> Printf.printf "Results:\n%s\n";;
```

Now we can try some examples! This fails because it has an unbound variable

```ocaml
let t0 = typecheck [] (Str"x") (Var 0) ([], 1);;
format_results (Var 0) t0;;
```

```
Results:
```

If we bind the variable, it works:

```
let t1 = typecheck [(Str"x", (Var 0))] (Str"x") (Var 0) ([], 1);;
format_results (Var 0) t1;;
```

```
Results:
'a
```

For a different expression that has no unbound variables:

```
let t2 = typecheck [] (List[Str"fun"; Str"x"; Str"x"]) (Var 0) ([], 1);;
format_results (Var 0) t2;;
```

```
Results:
'a -> 'a
```

And for a more complicated expression:

```
let t3 = typecheck
  []
  (* Encoding the term `fun x -> fun y -> fun z -> x y (z y)` *)
  (List[Str"fun"; Str"x";
    List[Str"fun"; Str"y";
      List[Str"fun"; Str"z";
        List[
          List[Str"x"; Str"y"];
          List[Str"z"; Str"y"]]]]])
  (Var 0)
  ([], 1);;
format_results (Var 0) t3;;
```

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

Note, because we do not do occurs checks, the following also works (not pretty-printing it because the pretty-printer would try to expand it forever):

```
let t4 = typecheck
  []
  (* Encoding the term `fun x -> x x` *)
  (List[Str"fun"; Str"x"; List[Str"x"; Str"x"]])
  (Var 0)
  ([], 1);;
```

```
val t4 : stream =
  Cons
  (([(7, List [Var 7; Str "->"; Var 8]); (3, List [Var 7; Str "->"; Var 8]);
      (4, Var 8); (6, Str "x"); (5, Str "x");
      (0, List [Var 3; Str "->"; Var 4]); (2, List [Str "x"; Str "x"]);
      (1, Str "x")],
    9),
  <fun>)
```

Which does represent the infinite type `'a = 'a -> 'b` as expected. To fix this, you would have to look at adding an “occurs check” into the `extend` function.

If you want to learn more about this kind of logic programming, you could read more about “microkanren”, or read the book “Reasoned Schemer” – it is an excellent book, I recommend it. So are the rest of that series.