Instructions: read these first!

Do not open the exam, turn it over, or look inside until you are told to begin.

Switch off cell phones and other potentially noisy devices.

Write your full name on the line at the top of this page. Do not separate pages.

You may refer to a double-sided “cheat sheet”, but no computational devices (such as laptops, calculators, phones, iPads, friends, enemies, pets, lovers).

Read questions carefully. Show all work you can in the space provided.

Where limits are given, write no more than the amount specified. The rest will be ignored.

Avoid seeing anyone else’s work or allowing yours to be seen.

Do not communicate with anyone but an exam proctor.

If you have a question, raise your hand.

When time is up, stop writing.

The points for each part are rough indication of the time that part should take.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Total:</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>
1. [35 points] For each of the following OCaml programs, write down the **type** or **value** of the given variable, or circle “Error” if you think there is a type or runtime error.

(a) [5 points]

```ocaml
let ans =
    let x = 10 in
    let f y z = x + y + z in
    let x = 100 in
    let h = f 5 in
    h x
```

**Value** `ans =` ________________

(b) [6 points]

```ocaml
let rec chain fs = match fs with
    | [] -> fun x -> x
    | f::fs' -> fun x -> f (chain fs' x)
```

**Type** `chain:` ________________

(c) [5 points]

```ocaml
let ans = chain [ (fun x -> x * x)
               ; (fun x -> 16 * x)
               ; (fun x -> x + 1)
               ] 1
```

**Value** `ans =` ________________

(d) [3 points]

```ocaml
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

let ans0 = Node (2, Node (1, Leaf, Leaf)
                 , Node (3, Leaf, Leaf))
```

**Type** `ans0:` ________________

(e) [5 points]

```ocaml
let rec flerb xs = match xs with
    | [] -> Leaf
    | x::xs' -> Node (x, Leaf, flerb xs')
```

**Type** `flerb:` ________________
(f) [3 points]
  let ans = flerb [0;1;2]

  Error
  Value ans = ____________________________

(g) [5 points]
  let rec glub f t = match t with
    | Leaf         -> Leaf
    | Node (x,l,r) -> Node (f x, glub f l, glub f r)

  Error
  Type glub: ____________________________

(h) [3 points]
  let ans = glub (fun x -> 2 * x) ans0

  Error
  Value ans = ____________________________
2. [20 points] Consider the two functions `sum` and `fac` shown below:

```ocaml
let rec sum n = match n with
    | 0 -> 0
    | n -> n + sum (n-1)

let rec fac n = match n with
    | 0 -> 1
    | n -> n * fac (n-1)
```

(a) [5 points] Write a tail recursive version of `sum` by filling in the blanks below:

```ocaml
let sumTR n =
    let rec helper acc n = ____________________________
        | __________ -> ________________________________
        | __________ -> ________________________________
    in
    in helper ____________________________
```

(b) [5 points] Write a tail recursive version of `fac` by filling in the blanks below:

```ocaml
let facTR n =
    let rec helper acc n = ____________________________
        | __________ -> ________________________________
        | __________ -> ________________________________
    in
    in helper ____________________________
```
(c) [6 points] *Spot that pattern!* Now write a higher-order function

```ocaml
val foldn : ('a -> int -> 'a) -> 'a -> int -> 'a
```

that generalizes the tail-recursion in `sumTR` and `facTR`, by filling in the blanks below:

```ocaml
let foldn f b n =

let rec helper acc n = ________________________
  | ___________________ -> ________________________
  | ___________________ -> ________________________
  in

  in helper ________________________
```

(d) [4 points] Your solution for `foldn` should be such that you can now implement `sum` and `fac` *without recursion* simply by passing in appropriate parameters to `foldn`. What are those parameters?

```ocaml
let sum = foldn ________________________ ____________
let fac = foldn ________________________ ____________
```
3. [25 points]
   In NanoML, we used exceptions at various places, for example, when looking up a variable that did not exist in the environment. A better approach is to use the `a option type, defined thus:

   ```
   type 'a option = None | Some of 'a
   ```

   (a) [4 points] Now, instead of throwing an exception (who knows where or how or even if it will get caught!) if a function can possibly fail, we can have it return an option value. For example:

   ```
   let safeDiv num den = match den with
   | 0  -> None
   | _  -> Some (num / den)
   ```

   Since division is undefined (and throws a nasty failure), we instead write a `safeDiv` that gracefully returns a `None` if the result is undefined, and `Just i` when the denominator is non-zero and hence the division is safe. What is the type of `safeDiv`?

   **Error**

   **Type** safeDiv: ________________

   (b) [5 points] Fill in the blanks to write a version of `lookup` that returns an option, that is:

   ```
   val lookup: 'a -> ('a * 'b) list -> 'b option
   ```

   When you are done, you should get the following behavior:

   ```
   # lookup "a" [("a", 1); ("b", 2), ("a", 10)];;
   - : int option = Some 1
   # lookup "z" [("a", 1); ("b", 2), ("a", 10)];;
   - : int option = None
   ```

   ```
   let rec lookup k kvs = ____________________________
   | ____________________ -> ____________________________
   | ____________________ -> ____________________________
   ```
(c) [4 points] Fill in the blanks to write a function

\[
\text{val lift1 : ('a -> 'b) -> 'a option -> 'b option}
\]

such that when you are done, you get the following behavior

```ocaml
# lift1 string_of_int (Some 1);;
- : string option = Some "1"

# lift1 string_of_int None;;
- : string option = None
```

```ocaml
let lift1 f xo = ____________________________
    | ___________________ -> ___________________
    | ___________________ -> ___________________
```

(d) [5 points] Fill in the blanks to write a function

\[
\text{val lift2 : ('a -> 'b -> 'c) -> 'a option -> 'b option -> 'c option}
\]

such that when you are done, you get the following behavior

```ocaml
# lift2 (+) (Some 1) (Some 10);;
- : int option = Some 11

# lift2 (+) (None) (Some 10);;
- : int option = None

# lift2 (+) (Some 1) (None);;
- : int option = None

# lift2 (+) (None) (None);;
- : int option = None
```

```ocaml
let lift2 f xo yo = ____________________________
    | ___________________ -> ___________________
    | ___________________ -> ___________________
```
(e) [7 points] Consider the subset of NanoML given by the type:

```plaintext
type expr = Var of string (* variable *)
    | Con of int (* constant *)
    | Neg of expr (* negation of an expression *)
    | Plus of expr * expr (* sum of two expressions *)
```

Write an interpreter function

```plaintext
val eval : (string * int) list -> expr -> int option
```

such that when you are done you get the following behavior:

```plaintext
# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Var "y"));;
- int option = Some 3

# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Con 20));;
- int option = Some 21

# eval [("x", 1); ("y", 2); ("x", 100)] (Plus (Var "x", Var "z"));;
- int option = None

# eval [("x", 1); ("y", 2); ("x", 100)] (Neg (Var "y"));;
- int option = Some (-2)

# eval [("x", 1); ("y", 2); ("x", 100)] (Neg (Var "z"));;
- int option = None
```

**Note:** Your implementation of `eval` must not use any `match-with` expressions other than the one given. Instead, use `lift1` and `lift2`. You may also use any other functions you have implemented during this exam.

```plaintext
let rec eval env e = match e with
    | Var x       -> _______________________________
    | Con i       -> _______________________________
    | Neg e'      -> _______________________________
    | Plus (e1, e2) -> _______________________________
```