CS 51: Abstraction and Design in Computation
First Midterm Examination
Spring, 2020

- *You have 90 minutes to complete this exam.*
- *This is a closed-book exam. However, you are free to use up to 10 letter-size pages of notes or other printed materials in preparing your solutions for this exam. No electronic devices of any kind may be used.*
- *The exam is in three sections comprised of 13 questions. Numbers in brackets like this* ***[nnn points]*** *are the points (out of 75 total) allocated to the problem and may provide a very approximate recommendation for allocating time.*
- *Write the answers to all problems in the boxes provided. Write with a pen (or a very dark pencil) as we will be scanning your exams for grading. Write clearly, as we can and will only grade what we can unambiguously read. The exam packet is intentionally stapled in the lower left corner to facilitate the scanning process. Do not remove the staple or remove any pages from the exam packet. Anything written outside of the provided boxes will not be graded. If additional room is needed for an answer, make a note in the box provided and write the remainder of your answer in one of the boxes at the back of the examination.*
- *Many of the problems ask you to define something or write code to do something. Throughout the exam, when we ask you to define a value or function or type or module, we mean that you should provide a top-level OCaml definition written in well-formed, idiomatic OCaml using the appropriate OCaml definitional construct (*`let`*,* `type`*,* `module`*, etc.). Your answers will be graded firstly on the well-formedness and correctness of the code, but in keeping with the course's goals, we may also secondarily consider the many other dimensions of code quality – including design and style issues – in evaluating your answers.*
- *To allow for anonymous grading of the exam, please write your name and ID number in the boxes below on this page* ***[0.5 points]****, and your ID number (but not your name) in the box provided at the top of all subsequent odd-numbered pages* ***[0.5 points]****.*
- ***By filling in your name and ID number below, you affirm your awareness of the standards of the Harvard College Honor Code.***

Solution:

SOLUTION

1. FUNCTIONAL FUNDAMENTALS

Problem 1. ***[20 points]*** *For each of the expressions below, write in the first box the type, if any, that OCaml would infer for the expression, or write "NO TYPE" if the expression is not typable.*

Then, for each of the typable expressions only, *write in the second box the expression's value if any, or write "NO VALUE" if the expression has no value. (For function values give the value simply as* `<fun>` *and for abstract values use* `<abstr>`*, mimicking the OCaml* REPL.*)*

We've done the first one for you as an example.

(1)
```
2 * 3 * 7 ;;
```

(2)
```
let rec f = fun x -> if x = 1 then pred x
                     else f (pred x) in
f 1000 ;;
```

(3)
```
List.fold_left (||) false [true] ;;
```

(4) `Some None ;;`
(5) `let rec f g h = f g h in f ;;`
(6) `let rec f g h = h f g in f ;;`
(7) `let rec f g h = f h g in f ;;`

(8)
```
let x = 2.0 in
let x = "2.0" in
x ^ x ;;
```
(9)
```
let x = 5 in
if x > 10 then true else raise Exit ;;
```
(10)
```
if if true then false else true then false else true ;;
```

Solution:

(1)
```
# 2 * 3 * 7 ;;
- : int = 42
```
(2)
```
# let rec f = fun x -> if x = 1 then pred x
#                      else f (pred x) in
# f 1000 ;;
- : int = 0
```
(3)
```
# List.fold_left (||) false [true] ;;
- : bool = true
```
(4)
```
# Some None ;;
- : 'a option option = Some None
```
(5)
```
# let rec f g h = f g h in f ;;
- : 'a -> 'b -> 'c = <fun>
```
(6)
```
# let rec f g h = h f g in f ;;
Line 1, characters 18-19:
1 | let rec f g h = h f g in f ;;
                      ^
Error: This expression has type 'a -> ('b -> 'c -> 'd) -> 'e
       but an expression was expected of type 'b
       The type variable 'b occurs inside 'a -> ('b -> 'c ->
  'd) -> 'e
```

This fails to type as `f` takes `h` as an argument which can itself take `f` as an argument. Thus the type of `f` must contain itself as a subpart.

(7)
```
# let rec f g h = f h g in f ;;
- : 'a -> 'a -> 'b = <fun>
```
(8)
```
# let x = 2.0 in
# let x = "2.0" in
# x ^ x ;;
Line 1, characters 4-5:
1 | let x = 2.0 in
        ^
Warning 26 [unused-var]: unused variable x.
- : string = "2.02.0"
```

(9)
```
# let x = 5 in
# if x > 10 then true else raise Exit ;;
Exception: Stdlib.Exit.
```
This expression types properly as a `bool` (since the two branches of the conditional must be of the same type, and one is an overt `bool`). You can tell it types since it managed to run (and raise an exception). Another way you can determine that it types as a `bool` is to place it in an appropriate context where it avoids evaluation, for instance,
```
# let p () = let x = 5 in
#            if x > 10 then true else raise Exit  ;;
val p : unit -> bool = <fun>
```
Note that `p` is of type `unit -> bool`.
By raising an exception, it avoids returning a value, so "NO VALUE" is the correct answer to that part.

(10)
```
# if if true then false else true then false else true ;;
- : bool = true
```

Problem 2. ***[5 points]*** *Without using any functions from the* `List` *module, define a function* `copies : int -> string -> string` *such that* `copies n str` *returns a string composed of* `n` *copies of the given string* `str`*. If* `n` *is negative, the function should return the empty string.*

For instance,

```
# copies 4 "abc" ;;
- : string = "abcabcabcabc"

# copies (-2) "abc" ;;
- : string = ""

# copies 12 "o_O " (* a crowd *) ;;
- : string = "o_O o_O o_O o_O o_O o_O o_O o_O o_O o_O o_O o_O "
```

Solution:

```
let rec copies n str =
  if n <= 0 then ""
  else str ^ (copies (pred n) str) ;;
```

2. MULTISETS

A MULTISET is a mathematical object much like a set – that is, an unordered collection of elements – except that a multiset, unlike a set, can contain more than one instance of the same element. Natural operations on multisets include adding and dropping elements and determining the count of how many occurrences of an element (zero or more) exist in a multiset, as well as union and intersection of multisets. In this section, you'll work with a multiset module signature and its implementation. But first, a short digression.

2.1. **Comparing values.** Recall the definition of the `COMPARABLE` module signature from the textbook that packages together a type with an ordering function over elements of the type. (Note that the `compare` function here uses a different convention for its return value than the one from the previous section; it returns an `order`, not an `int`.)

We repeat the module signature here for your reference:

```
module type COMPARABLE =
  sig
    type t
    type order = Less | Equal | Greater
    val compare : t -> t -> order
  end ;;
```

Problem 3. ***[7 points]*** *Define a module called* `IntComparable` *that satisfies the* `COMPARABLE` *signature where the type* `IntComparable.t` *is* `int`*. Your definition should allow for behavior like*

```
# IntComparable.compare 3 4 ;;
- : IntComparable.order = IntComparable.Less
# IntComparable.compare 5 5 ;;
- : IntComparable.order = IntComparable.Equal
```

Make sure to apply an appropriate module signature to `IntComparable`*. This module will be useful in the later parts of this section.*

Solution:

```
# module IntComparable : (COMPARABLE with type t = int) =
#   struct
#     type t = int
#     type order = Less | Equal | Greater
#     let compare x y =
#       if x < y then Less
#       else if x = y then Equal
#       else Greater
#   end ;;
module IntComparable :
  sig
    type t = int
    type order = Less | Equal | Greater
    val compare : t -> t -> order
  end
```

2.2. **A multiset signature and its implementation.** A signature for a multiset abstract data type is the following:

```
module type MULTISET =
  sig
    type element  (* the type of elements of the multiset *)
    type t        (* the type of the multiset itself *)

    (* an empty multiset *)
    val empty_set : t

    (* empty_p mset -- Returns `true` if and only if `mset`
       is empty *)
    val empty_p : t -> bool

    (* add elt mset -- Returns a multiset like `mset` with
       one more `elt` *)
    val add : element -> t -> t

    (* drop elt mset -- Returns a multiset with one `elt`
       removed from `mset` *)
    val drop : element -> t -> t

    (* count elt mset -- Returns the number of `elt`s in
       `mset` *)
    val count : element -> t -> int

    (* union mset1 mset2 -- Returns a multiset containing
       the elements of both argument multisets *)
    val union : t -> t -> t

    (* intersection elt mset -- Returns a multiset containing
       the elements that are in both argument multisets *)
    val intersection : t -> t -> t
  end ;;
```

Figure 1 provides a partial definition for a functor `MakeMultiset` that generates modules implementing the `MULTISET` signature whose elements are taken from a `COMPARABLE` module. In this implementation, the multiset is internally represented as a list of pairs of an element and the count of how many times the element occurs in the multiset. It obeys the invariants that counts are always positive and the pairs are kept sorted by the element.

Problem 4. ***[5 points]*** *You'll notice that in the second line of the functor implementation in Figure 1, there's a box where the signature of the module that the functor generates should go. What ought to go in the box to specify the signature of modules generated by the* `MakeMultiset` *functor?*

Solution:

```
MULTISET with type element = Element.t
```

Problem 5. ***[5 points]*** *Using the* `MakeMultiset` *functor, define a module* `IntMultiset` *for multisets of integers.*

Solution: The following works, providing no explicit signature:

```
# module IntMultiset =
#   MakeMultiset (IntComparable) ;;
module IntMultiset :
  sig
    type element = IntComparable.t
    type t = MakeMultiset(IntComparable).t
    val empty_set : t
    val empty_p : t -> bool
    val add : element -> t -> t
    val drop : element -> t -> t
    val count : element -> t -> int
    val union : t -> t -> t
    val intersection : t -> t -> t
  end
```

If an explicit signature is provided, it will require a sharing constraint, viz.,

```
# module IntMultiset : (MULTISET with type element = int) =
#   MakeMultiset (IntComparable) ;;
module IntMultiset :
  sig
    type element = int
    type t
    val empty_set : t
    val empty_p : t -> bool
    val add : element -> t -> t
    val drop : element -> t -> t
    val count : element -> t -> int
    val union : t -> t -> t
    val intersection : t -> t -> t
  end
```

Problem 6. ***[2 points]*** *In a sentence, explain the advantage of using a* functor *to generate (monomorphic) implementations of the* `MULTISET` *signature, as in Figure 1, over providing a single (polymorphic)* module.

> **Solution:** The implementation as a functor allows packaging up a comparison function with the element type, allowing for custom comparison functions instead of a fixed comparison function (such as `Stdlib.compare`).

For the remaining problems in this section, you can assume that the `IntMultiset` module has been opened as by

```
# open IntMultiset ;;
```

Problem 7. ***[3 points]*** *Now define an integer multiset* `m` *that contains two 5s and a 1.*

Solution:

```
# let m = empty_set |> add 5 |> add 1 |> add 5 ;;
val m : IntMultiset.t = <abstr>
```

Problem 8. ***[3 points]*** *Give an expression of type* `bool` *that evaluates to* `true` *just in case the multiset* `m` *has more 5s than 1s.*

Solution:

```
# count 5 m > count 1 m ;;
- : bool = true
```

```
module MakeMultiset (Element : COMPARABLE)
                    : ( ... the module signature goes here... )
  =
  struct
    type element = Element.t

    (* multisets are implemented as an association list of
       elements and their count, sorted by element according
       to the comparison function *)
    type t = (element * int) list

    let empty_set = []

    let empty_p mset = mset = empty_set

    let rec adjust fn elt mset =
      match mset with
      | [] -> let newcount = fn 0 in
              if newcount = 0 then mset
              else (elt, newcount) :: mset
      | (current, curcount) :: rest ->
         match Element.compare elt current with
         | Less -> let newcount = fn 0 in
                   if newcount = 0 then mset
                   else (elt, newcount) :: mset
         | Equal -> let newcount = fn curcount in
                    if newcount = 0 then rest
                    else (elt, newcount) :: rest
         | Greater -> (current, curcount) :: adjust fn elt rest

    let rec add elt mset =
      adjust succ elt mset

    let rec drop elt mset =
      adjust (fun count -> if count = 0 then 0 else pred count)
             elt mset

    (* ...the rest of the implementation would go here... *)
  end ;;
```

FIGURE 1. Part of an implementation of multisets using sorted association lists.

3. THE ROYAL SUCCESSION

In this and the following sections, you should feel free to make idiomatic use of library functions such as `map`*,* `fold_left`*,* `fold_right`*, and* `filter` *and other functions from the the* `List` *module and the* `Stdlib` *module. For brevity, you can also assume that these modules have been opened already as by*

```
# open List ;;
```

The royal succession is the sequencing of members of the British royal family as to what order they will ascend to the throne. As of the passing of the Succession to the Crown Act 2013, the succession order is based on "absolute primogeniture", a traversal of the family tree of the monarch with the parent at the root of the tree coming before the children's families and with siblings ordered by age. (Sex and membership in the Catholic Church are no longer factors.) Thus, for instance, for the Windsor (partial) family tree depicted in Figure 2, the order of succession begins at the root of the tree with Elizabeth, then succeeding to the oldest child Charles and his family (in primogeniture order – William, George, Charlotte, etc.), then Anne and her family, and finally Andrew and Edward's families.

We can represent a royal family tree using the following type definition, a record type that contains the name and age of a royal, together with a list of children:

```
type royal = {name : string;
              age : int;
              children : royal list} ;;
```

The Windsor family (or at least a portion of it) is then as given in Figure 3. It defines a value named `windsors`, which is used below.

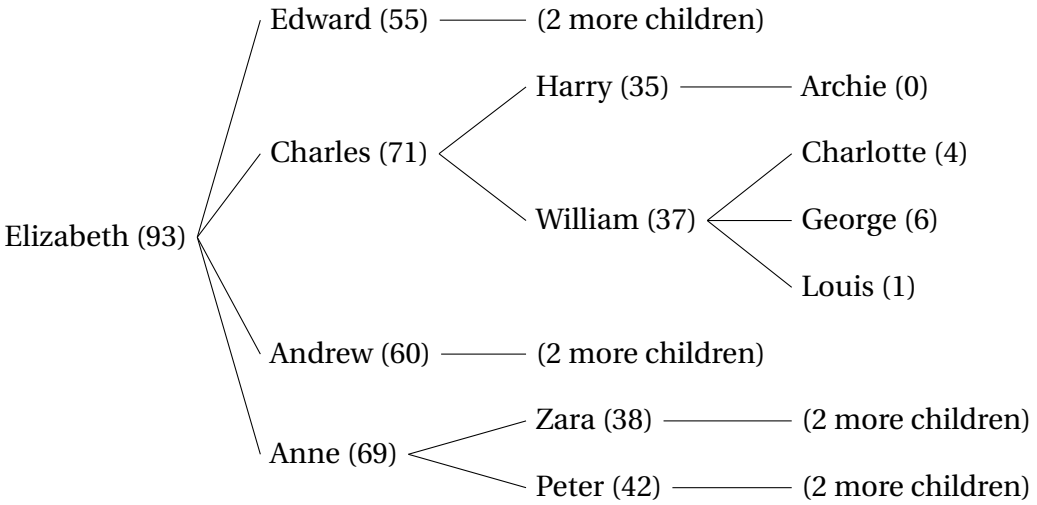

FIGURE 2. A partial family tree of Elizabeth II, along with the ages of the members. Note that the tree as depicted is not in age order.

Problem 9. ***[6 points]*** *Recall that the* `Stdlib` *function* `compare : int -> int -> int` *compares two integers* `x` *and* `y` *using the following convention: It returns* `0` *if* `x` *is equal to* `y`*, a negative integer if* `x` *is less than* `y`*, and a positive integer if* `x` *is greater than* `y`*. (This is just the convention expected by the* `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` *function.)*

Define a function `compare_age : royal -> royal -> int` *that uses the same convention to compare the ages of two royals. That is, it returns a negative integer if the first of the two royals is younger, zero if the same age, and a positive integer if the first of the two is older.*

Solution:

```
# let compare_age (person1 : royal) (person2 : royal) : int =
#   Stdlib.compare person1.age person2.age ;;
val compare_age : royal -> royal -> int = <fun>
```

or

```
# let compare_age (person1 : royal) (person2 : royal) : int =
#   person1.age - person2.age ;;
val compare_age : royal -> royal -> int = <fun>
```

Solutions with pattern matching to extract the ages are possible as well.

Problem 10. ***[1 points]*** *Is your definition of* `compare_age` *curried or uncurried?*

- ☐ *curried*
- ☐ *uncurried*
- ☐ *neither*

Solution: Curried of course. The requested type of the function specifies it as a curried function.

```
let windsors =
  {name = "Elizabeth";
   age = 93;
   children =
     [{name = "Anne";
       age = 69;
       children = [{name = "Peter";
                    age = 42;
                    children = [] (* eliding two children *) };
                   {name = "Zara";
                    age = 38;

                    children = [] (* eliding two children *) }]};
      {name = "Andrew";
       age = 60;
       children = [] (* eliding two children *)};
      {name = "Charles";
       age = 71;
       children = [{name = "William";
                    age = 37;
                    children = [{name = "Louis";
                                 age = 1;
                                 children = []};
                                {name = "George";
                                 age = 6;
                                 children = []};
                                {name = "Charlotte";
                                 age = 4;
                                 children = []}]};
                   {name = "Harry";
                    age = 35;
                    children = [{name = "Archie";
                                 age = 0;
                                 children = []}]}]};
      {name = "Edward";
       age = 55;
       children = [] (* eliding two children *)}]} ;;
```

FIGURE 3. The Windsor family tree represented as the `royal` data type

Problem 11. ***[7 points]*** *Define a function* `count_royals` *that returns the number of royals in its argument royal family tree. For instance,*

```
# count_royals windsors ;;
- : int = 13
```

Solution:

```
# let rec count_royals (family : royal) : int =
#   1 + fold_left (+) 0
#                 (map count_royals family.children) ;;
val count_royals : royal -> int = <fun>
```

The adding one (for the parent) can be brought into the fold as initial value.

```
# let rec count_royals (family : royal) : int =
#   fold_left (+) 1
#             (map count_royals family.children) ;;
val count_royals : royal -> int = <fun>
```

Some might pull out the summing function, as

```
# let sum = fold_left (+) 0 ;;
val sum : int list -> int = <fun>
#
#     let rec count_royals family =
#       1 + sum (map count_royals family.children) ;;
val count_royals : royal -> int = <fun>
```

or use backward application to good effect, as

```
# let rec count_royals family =
#   family.children      (* take all the children *)
#   |> map count_royals  (* and count their family members *)
#   |> fold_left (+) 0   (* add up all the counts *)
#   |> succ ;;           (* plus one for the parent *)
val count_royals : royal -> int = <fun>
```

Problem 12. ***[2 points]*** *What is the type of* `count_royals`*?*

Solution: `royal -> int`

Problem 13. ***[8 points]*** *Define a function* `primogeniture : royal -> string list`*, which returns a list of the names of the members of a royal family in primogeniture order (that is, according to the succession traversal derived above). For instance, the computation*

```
# primogeniture windsors ;;
- : string list =
["Elizabeth"; "Charles"; "William"; "George"; "Charlotte";
  "Louis"; "Harry";
 "Archie"; "Anne"; "Peter"; "Zara"; "Andrew"; "Edward"]
```

shows that 6-year-old George is the third in line to the throne after Charles and William.

Feel free to make use of functions you've implemented in previous problems as well as `List` *library functions. Keep in mind that the* `royal` *data structure might not have the children listed in age order (for instance, as in Figure 3).*

Solution: Here is a solution that makes good use of the `List` module:

```
#     let rec primogeniture ({name; children; _} : royal)
#                           : string list =
#       cons name (concat (map primogeniture
#
#                         (rev (sort compare_age children)))) ;;
val primogeniture : royal -> string list = <fun>
```

Taking advantage of backward application clarifies the code.

```
# let rec primogeniture ({name; children; _} : royal)
#                       : string list =
#   children
#   |> sort compare_age
#   |> rev
#   |> map primogeniture
#   |> concat
#   |> cons name ;;
val primogeniture : royal -> string list = <fun>
```

In the first example, the use of `cons` instead of `::` is incidental. Either works fine, though the REPL pretty-printer handles `cons` slightly nicer. In the second example, use of `cons` is needed for the partial application, since value constructors can't be partially applied.

End of exam.

Total points: 75

Do not write below this line. It will not be scanned.

EXTRA SPACE FOR ANSWERS