0/27 Questions Answered

Midterm Exam 1

Student Name

Search students by name or email...

Q1 Instructions and affirmation

0 Points

Before beginning this exam, make sure you have read and understood the exam instructions that were distributed ahead of the exam and are available at https://url.cs51.io/exam-instructions, and check the boxes below.

☐ I have read and understood the exam instructions that were distributed ahead of the exam.

☐ I affirm my awareness of the standards of the Harvard College Honor Code.

Save Answer

Q2 Further details

0 Points

Throughout this exam, for the purpose of writing your answers, you can assume that the `Stdlib` and `List` modules have already been opened for you.

Save Answer

Q3 Finger exercises

7 Points

Each of the expressions below contains a single blank (shown as an underline "____"). Your job is to fill in the blank with a single expression that is well-formed and well-typed in context, such that the expression as a whole evaluates to `42`. (For the purpose of this problem, you can ignore any warnings generated.)

If such an expression exists, select "This expression" and in the text box provide an appropriate expression to fill in the blank. If no such expression exists, select the option "No such expression" and leave the text block blank. (If you select "No such expression" but also fill in the text block, we'll ignore the latter.)

For instance, in the example

```
let x = ________ in
x + 40 ;;
```

you would check the "This expression" option and provide the expression

```
2
```

or

```
3 - 1
```

or any of a wide variety of other expressions that would work.

Q3.1
1 Point

```
6 * ________ ;;
```

No such expression

This expression:

Save Answer

Q3.2
1 Point

```
map _________ [1; 42; 1] ;;
```

No such expression

This expression:

Save Answer

Q3.3
1 Point

```
let x = 42 in
let y = 7 in
let _________ = x in
y ;;
```

No such expression

This expression:

Save Answer

Q3.4
1 Point

```
let x : bool -> int = _________ in 42 ;;
```

No such expression

This expression:

Save Answer

Q3.5
1 Point

```
let f x = x * x - x in f _________ ;;
```

No such expression

This expression:

Save Answer

Q3.6
1 Point

```
42 |> _________ |> ((+) 2) ;;
```

No such expression

This expression:

Save Answer

Q3.7
1 Point

```
let _________ = Some 42 in x ;;
```

No such expression

This expression:

Save Answer

Q4 Finding elements in a list
12 Points

The problems in this section concern a polymorphic function `findafter` that takes a condition and a list and returns the remainder of the list starting with the first element for which the condition returns `true`. For example,

```
# findafter (fun x -> x > 7) [1; 3; 5; 7; 9; 11] ;;
- : int list = [9; 11]

# let is_even n = n mod 2 = 0 in
  findafter is_even [1; 3; 7; 2; 5; 4; 4] ;;
- : int list = [2; 5; 4; 4]

# findafter (fun x -> x) [false; false; true; false; true; false] ;;
- : bool list = [true; false; true; false]
```

If no element satisfies the condition, `findafter` should raise an *appropriate* exception.

You may code the function directly or use higher-order functions in the map/fold/filter style.

Q4.1
2 Points

What is the type of `findafter`?

Save Answer

Q4.2

6 Points

Define the `findafter` function.

Save Answer

Q4.3

2 Points

What is the length of the shortest list that the `findafter` function can return?

0

1

2

None of the above

Save Answer

Q4.4
2 Points

We define a new function called `mystery` as follows:

```
let mystery x =
  match x with
  | [] -> []
  | _ -> findafter (fun x -> true) x
```

Describe the behavior of the `mystery` function in as simple a form as possible.

Save Answer

Q5 Zipping and unzipping

20 Points

In this section, you'll implement some functions related to the `zip` function from lab 4, repeated here for your reference:

```
let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
  match x, y with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
```

This version of `zip` has the problem that it doesn't handle unequal length lists well: It generates an inexhaustive match warning at compile time and it raises a match failure exception at run time when applied to lists of unequal lengths.

Q5.1

4 Points

Define a new version of `zip` that

1. generates no inexhaustive match warnings, and
2. raises a more appropriate run-time exception when applied to lists of unequal lengths.

Save Answer

Q5.2
6 Points

Another way of handling unequal length lists is to provide for a default value that is used to pad the end of the shorter list to make it the length of the longer list. Define a function `zip_default : 'a -> 'a list -> 'a list -> ('a * 'a) list` that works in this way. Here are some examples of its use:

```
# zip_default 0 [1; 3; 5] [2; 4; 6; 8; 10] ;;
- : (int * int) list = [(1, 2); (3, 4); (5, 6); (0, 8); (0, 10)]
# zip_default true [false; true] [] ;;
- : (bool * bool) list = [(false, true); (true, true)]
```

Save Answer

Q5.3
2 Points

Notice that the types of the two list arguments for `zip` are (respectively) `'a list` and `'b list`, and the types of the two list arguments for `zip_default` are (respectively) `'a list` and `'a list`. Why this difference? Provide a succinct explanation; a sentence or two should suffice. *Overlong, disjunctive, or hedged answers will be discounted.*

Save Answer

Q5.4
2 Points

Now consider a polymorphic function `unzip` that is the opposite of `zip`. It takes a list of pairs and returns a pair of lists that when zipped give the original list. Here are some examples of its use:

```
# unzip [(3,true); (4, false); (5, false)] ;;
- : int list * bool list = ([3; 4; 5], [true; false; false])

# unzip (zip [false; true; false; false] [1; 2; 3; 4]) ;;
- : bool list * int list = ([false; true; false; false], [1; 2; 3; 4])
```

What is the type of `unzip`?

Save Answer

Q5.5
6 Points

Now provide a definition for `unzip`.

Save Answer

Q6 Recipe measurements

12 Points

In the next few problems, you'll develop a way to specify recipes and perform calculations over them. Here are some type definitions related to recipes. (We've left off the definition of the `measure` type.)

```
(* unit_ -- The type of measurement units *)
type unit_ = Cup | Ounce | Tablespoon | Teaspoon ;;

(* ingredient -- Names for ingredients *)
type ingredient = string ;;

(* measure -- The type for quantities of an ingredient, either as a
   count (2 apples, say) or a measurement (1.5 cups of water). *)
type measure =
  ___________________________
```

Q6.1

2 Points

Why do you think we chose the name `unit_` (with a trailing underline) for the type for measurement units, rather than the simpler type name `unit`?

Save Answer

Q6.2
4 Points

The following list of ingredients is based on a guacamole recipe from Spruce Eats
(https://www.thespruceeats.com/best-simple-guacamole-recipe-7507237). *)

```
let guacamole = [
  Count (2, "ripe avocados");
  Measure (0.25, Cup, "diced onions");
  Count (1, "garlic clove");
  Count (1, "small serrano chile, minced");
  Measure (1., Tablespoon, "lime juice");
  Measure (1., Teaspoon, "salt");
  Measure (0.125, Teaspoon, "ground cumin");
  Measure (0.25, Cup, "packed coarsely chopped fresh cilantro")
] ;;
```

Based on this example, you should have enough information to provide a definition of the `measure` type. Provide that definition here:

Save Answer

Q6.3
6 Points

Define a function `ounces : measure list -> float` that takes a list of ingredients and returns the total number of ounces of all of the measured ingredients. (The "counted" ingredients should be ignored, that is, treated as contributing 0 ounces.) For your reference, conversions with ounces are as follows:

- 6 teaspoons per ounce
- 2 tablespoons per ounce
- 8 ounces per cup

For instance,

```
# ounces guacamole ;;
- : float = 4.6875
```

Save Answer

Q7 Bit sequences

23 Points

In the next few problems, you'll develop an abstract data type for bit sequences. Bit sequences can be manipulated by performing bit operations on them, operations like logical "and", "or", "xor", and "not". These bit operations act on two bit sequences of the same length by applying the logical operations to corresponding bits. For instance, consider the two bit sequences 0 1 1 0 1 and 1 0 1 1 1. The "and" of these two bit sequences is the bit sequence 0 0 1 0 1. The "xor" of these two bit sequences is 1 1 0 1 0.

```
     0 1 1 0 1           0 1 1 0 1
and  1 0 1 1 1      xor  1 0 1 1 1
--------------      --------------
     0 0 1 0 1           1 1 0 1 0
```

There is a natural interpretation of bit sequences as integers as well: we just consider them as bits in the binary representation of the integer. For the purposes of these problems, we'll think of the bits as being ordered from least to most significant bits. (That's the opposite order from how we usually write down binary numbers, but it makes the code easier.) So the representation of the number 4 as a bit sequence would be 0 0 1 (and not 1 0 0). Note that we can always add some zeroes at the end of the bit sequence without changing the number represented, so 4 can also be represented by the bit sequence 0 0 1 0 or 0 0 1 0 0 0, etc. This comes in handy when we need to make two bit sequences the same length so as to perform bit operations on them.

We've given you a major start on such a module. In this implementation, bit sequences are represented by lists of integers obeying some invariants:

- Each integer is either 0 or 1.
- The bits are provided from least to most significant, so the integer 4

(100 in binary) would be represented by the list `[0; 0; 1]` (or `[0; 0; 1; 0]`, etc.).

- Bit sequence representations may be of any length; The empty list `[]` represents the number 0.

Here is (most of) a module for bit sequences:

```
module Bits =
  struct
    (* The type of bit sequence representations *)
    type t = int list
    (* bits_of_int n -- Returns a bit representation of the int `n` *)
    let rec bits_of_int x =
      if x = 0 then []
      else x mod 2 :: bits_of_int (x / 2)
    (* int_of_bits bits -- Returns the integer represented by bit
       sequence `bits` *)
    let int_of_bits bits =
      fold_right (fun bit acc -> acc * 2 + bit) bits 0
    (* lengthen_bits bits1 bits2 -- Auxiliary function. Returns a pair of
       bit strings like `bits1` and `bits2` but padded at the end with 0s so
       they are the same length. *)
    let lengthen_bits bits1 bits2 =

      _________________________
    (* xor_ bits1 bits2 -- Returns the bitwise exclusive or of the two bit
       sequences `bits1` and `bits2` *)
    let xor_ bits1 bits2 =
      let bits1, bits2 = lengthen_bits bits1 bits2 in
      map2 (fun x y -> (x + y) mod 2) bits1 bits2
    (* and_ bits1 bits2 -- Returns the bitwise and of the two bit
       sequences `bits1` and `bits2` *)
    let and_ bits1 bits2 =
      let bits1, bits2 = lengthen_bits bits1 bits2 in
      map2 (fun x y -> x * y) bits1 bits2
    (* or_ bits1 bits2 -- Returns the bitwise or of the two bit
       sequences `bits1` and `bits2` *)
    let or_ bits1 bits2 =
      let bits1, bits2 = lengthen_bits bits1 bits2 in
      map2 (fun x y -> x + y - x * y) bits1 bits2
    (* not_ b -- Returns the bitwise negation of the bit
       sequence `b` *)
    let not_ =

      ________________________
    (* serialize b -> Returns a string representation of the bit
       sequence `b`, e.g., "100" for the representation of 4 *)
    let rec serialize bits =
      match bits with
      | [] -> ""
      | head :: tail -> (serialize tail) ^ (string_of_int head)
  end ;;
```

Q7.1
2 Points

Provide a specific example of an `int list` that violates one or more of the bit sequence invariants, and is thus not a valid representation of a bit sequence.

Save Answer

Q7.2
6 Points

Provide a definition of the `lengthen_bits` function. This is an auxiliary function that isn't of use outside the module. It is used to pad bit sequences with 0s at the end so they are the same length, so that they can then be operated on by functions like `xor_`. Examples:

```
# lengthen_bits [1; 1] [0; 0; 1] ;;
- : int list * int list = ([1; 1; 0], [0; 0; 1])

# lengthen_bits [0; 0; 1; 1] [1] ;;
- : int list * int list = ([0; 0; 1; 1], [1; 0; 0; 0])
```

Hint: Recall you can use functions that you have previously implemented on this exam.

Save Answer

Q7.3

4 Points

Provide a definition of the `not_` function. Example:

```
# not_ [0; 1; 1; 0; 0] ;;
- : int list = [1; 0; 0; 1; 1]
```

Save Answer

Q7.4

6 Points

We'd like to constrain the `Bits` module to an appropriate signature. Provide a definition for an appropriate OCaml signature called `BITS`. We've provided a start. You should fill in what goes in the blank to define an appropriate signature for the `Bits` module.

```
module type BITS =
  sig
    ______________________
  end ;;
```

Save Answer

Q7.5
2 Points

Recall that the first line of the definition of the `Bits` module above was `module Bits =`. What should it have been so that it would be constrained by the signature you just defined?

Save Answer

Q7.6
3 Points

The `Bits` module, constrained by the signature you defined, should now constitute an abstract data type. Can you use the `Bits` ADT to generate a bit sequence (like the one you provided above) that violates the representational invariants? If so, provide an example of such a case. If not, explain why not. A sentence or two should suffice.

Save Answer

Save All Answers

Submit & View Submission >