



CS 51: Abstraction and Design in Computation  
First Midterm Examination  
Spring, 2019

- *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 four sections comprised of 10 questions. Numbers in brackets like this **[nnn points]** are the points (out of 80 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.***

YOUR NAME:

YOUR HARVARD ID NUMBER:

## 1. TYPES AND TYPE INFERENCE

**Problem 1. [12 points]** For each of the following OCaml function types, define a function (with no explicit typing annotations, that is, no uses of the `:` operator) for which OCaml would infer that type. The functions need not be practical or do anything useful; they need only have the requested type. For this problem, do not make use of anything from any library modules other than `Pervasives`. Provide your answers in the boxes provided below. (The first problem is done for you as an example.)

(1) `bool -> unit`

```
let f b = if b then () else () ;;
```

(2) `int -> int option`

(3) `'a * bool -> 'a`

(4) 'a -> 'b list option

(5) ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)

**Problem 2.** [12 points] For each of the following top-level `let` definitions, give a typing for the variable being defined that provides its most general type (as would be inferred by OCaml) or explain briefly why the expression is not well typed. (The first problem is done for you as an example.)

(1) `let f x =  
    x +. 42. ;;`

`f : float -> float`

(2) `let x = let y x = x in y ;;`

(3) `let x = 3, 3 * 3, 3 ;;`

(4) `let x = List.map ((+) 3.) [4., 5., 6.] ;;`

(5) `let x = let open List in  
 fun y -> filter ((=) y) ;;`

## 2. DEFINING SOME FUNCTIONS

The `Pervasives.string_of_bool` function returns a string representation of a boolean. Here it is in operation:

```
# string_of_bool (3 = 3) ;;  
- : string = "true"  
# string_of_bool (0 = 1) ;;  
- : string = "false"
```

**Problem 3.** [2 points] What is the type of `string_of_bool`?

**Problem 4.** [6 points] Provide your own definition of the function `string_of_bool` (without using `Pervasives.string_of_bool` of course).

Recall that the `Pervasives.compare` function compares two values, returning an `int` based on their relative magnitude: `compare x y` returns `0` if `x` is equal to `y`, `-1` if `x` is less than `y`, and `+1` if `x` is greater than `y`.

A function `compare_lengths : 'a list -> 'b list -> int` that compares the lengths of two lists can be implemented using `compare` by taking advantage of the `length` function<sup>1</sup> from the `List` module:

```
let compare_lengths xs ys =
  compare (List.length xs) (List.length ys) ;;
```

For instance,

```
# compare_lengths [1] [2; 3; 4] ;;
- : int = -1
# compare_lengths [1; 2; 3] [4] ;;
- : int = 1
# compare_lengths [1; 2] [3; 4] ;;
- : int = 0
```

However, this implementation of `compare_lengths` does a little extra work than it needs to. Its complexity is  $O(n)$  where  $n$  is the length of the *longer* of the two lists.

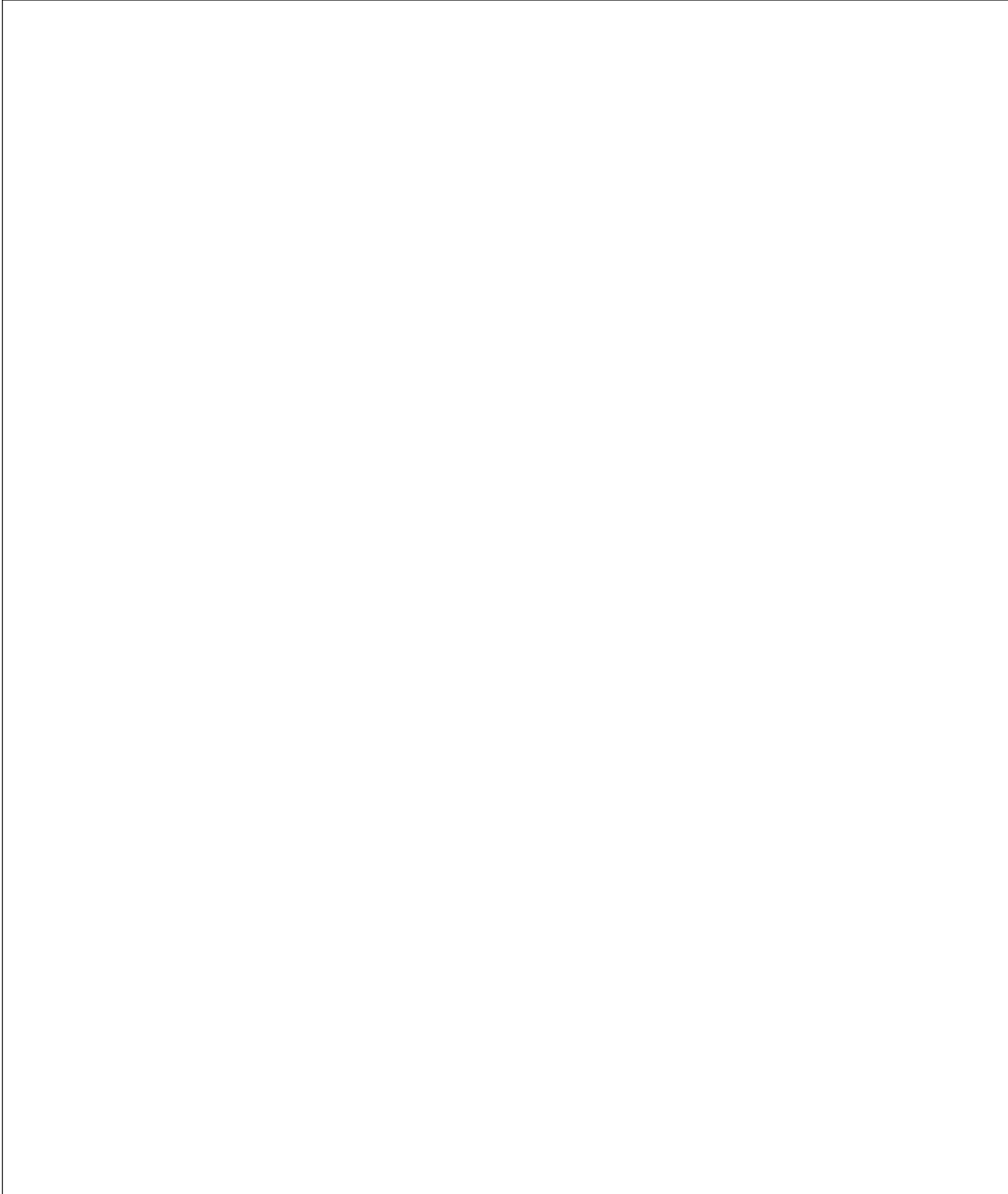
---

<sup>1</sup>For reference, this built-in `length` function is, unsurprisingly, linear in the length of its argument.

**Problem 5.** [6 points] *Why does `compare_lengths` have this big-O complexity? In particular, why does the length of the shorter list not play a part in the complexity? We're looking for a brief informal argument here, not a full derivation of its complexity.*



**Problem 6. [8 points]** Provide an alternative implementation of `compare_lengths` whose complexity is  $O(n)$  where  $n$  is the length of the shorter of the two lists, not the longer.



## 3. SUBSTITUTION SEMANTICS

**Problem 7. [8 points]** Provide a derivation demonstrating that the expression

$(\text{fun } x \rightarrow x + 2) 3$

evaluates to 5 according to the substitution semantics rules provided in Figure 1. For your reference, we've also provided the rules for substitution in Figure 2.



$$\bar{n} \Downarrow \bar{n} \quad (R_{int})$$

$$\begin{array}{l}
 P + Q \Downarrow \\
 \left| \begin{array}{l} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \end{array} \right. \\
 \Downarrow \overline{m+n}
 \end{array} \quad (R_+)$$

$$\begin{array}{l}
 P / Q \Downarrow \\
 \left| \begin{array}{l} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \end{array} \right. \\
 \Downarrow \overline{m/n}
 \end{array} \quad (R_/)$$

(and similarly for other binary operators)

$$\begin{array}{l}
 \text{let } x = D \text{ in } B \Downarrow \\
 \left| \begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array} \right. \\
 \Downarrow v_B
 \end{array} \quad (R_{let})$$

$$\text{fun } x \rightarrow B \Downarrow \text{fun } x \rightarrow B \quad (R_{fun})$$

$$\begin{array}{l}
 P \ Q \Downarrow \\
 \left| \begin{array}{l} P \Downarrow \text{fun } x \rightarrow B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array} \right. \\
 \Downarrow v_B
 \end{array} \quad (R_{app})$$

FIGURE 1. Definitions of rules for evaluating expressions, for a functional language with naming and arithmetic.

$$\overline{m}[x \mapsto P] = \overline{m} \quad (1)$$

$$x[x \mapsto P] = P \quad (2)$$

$$y[x \mapsto P] = y \quad \text{where } x \neq y \quad (3)$$

$$(Q + R)[x \mapsto P] = Q[x \mapsto P] + R[x \mapsto P] \quad (4)$$

and similarly for other binary operators

$$QR[x \mapsto P] = Q[x \mapsto P]R[x \mapsto P] \quad (5)$$

$$(\text{fun } x \rightarrow Q)[x \mapsto P] = \text{fun } x \rightarrow Q \quad (6)$$

$$(\text{fun } y \rightarrow Q)[x \mapsto P] = \text{fun } y \rightarrow Q[x \mapsto P] \quad (7)$$

where  $x \neq y$  and  $y \notin FV(P)$

$$(\text{fun } y \rightarrow Q)[x \mapsto P] = \text{fun } z \rightarrow Q[y \mapsto z][x \mapsto P] \quad (8)$$

where  $x \neq y$  and  $y \in FV(P)$  and  $z$  is a fresh variable

$$(\text{let } x = Q \text{ in } R)[x \mapsto P] = \text{let } x = Q[x \mapsto P] \text{ in } R \quad (9)$$

$$(\text{let } y = Q \text{ in } R)[x \mapsto P] = \text{let } y = Q[x \mapsto P] \text{ in } R[x \mapsto P] \quad (10)$$

where  $x \neq y$  and  $y \notin FV(P)$

$$(\text{let } y = Q \text{ in } R)[x \mapsto P] = \text{let } z = Q[x \mapsto P] \text{ in } R[y \mapsto z][x \mapsto P] \quad (11)$$

where  $x \neq y$  and  $y \in FV(P)$  and  $z$  is a fresh variable

FIGURE 2. Definition of substitution of expressions for variables in expressions for a functional language with naming and arithmetic.

## 4. BINARY SEARCH TREES AND GORN ADDRESSES

In this section, you'll work with binary search trees, and learn about a method for specifying particular nodes in trees due to computer pioneer Saul Gorn (Figure 3(a)).

Binary search trees (for the purpose of this section) are binary trees in which (non-empty) nodes store a value, and further satisfy an invariant that for a node with value  $x$  all of the values stored in its left subtree compare (via the  $<$  operator) as less than  $x$  and all in the right subtree compare greater than or equal to  $x$ . This allows efficient insertion into a tree and searching in the tree for a given item.

Here's an algebraic data type for polymorphic binary search trees, where the values stored at the nodes are of type `'a`.

```
type 'a bintree =  
  | Empty  
  | Node of 'a bintree * 'a * 'a bintree ;;
```

We've provided in the box below an attempt at a function `insert : 'a -> 'a bintree -> 'a bintree` to insert a new item into a binary search tree. Unfortunately, there are several mistakes in the code that lead to errors, warnings, or incorrect functioning.

```
1 let insert (item : 'a) (tree : bintree) : bintree =  
2   match bintree with  
3   | Empty -> Node (Empty, item, Empty)  
4   | Node (left, old, right) ->  
5     if old < item then  
6       Node (insert item left, right)  
7     else  
8       Node (left, insert item right) ;;
```

**Problem 8. [9 points]** *Identify as many of these bugs as there are (but no more), giving line numbers for each and explaining what each problem is as succinctly but specifically as you can and describing how it might be fixed.*



Using the insert function (as corrected), we can, for instance, build the tree depicted in Figure 3(b) as follows:

```
# let tr = Empty
#     |> insert 10
#     |> insert 5
#     |> insert 15
#     |> insert 7
#     |> insert 9 ;;
val tr : int bintree =
  Node (Node (Empty, 5, Node (Empty, 7, Node (Empty, 9,
    Empty))), 10,
    Node (Empty, 15, Empty))
```

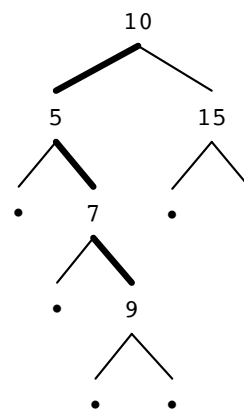
The *Gorn address* of a node in a tree (named after the early computer pioneer Saul Gorn of University of Pennsylvania (Figure 3(a)), who invented the technique) is a description of the path to take from the root of the tree to the node in question. For a binary tree, the elements of the path specify whether to go left or right at each node starting from the root of the tree. We'll define an enumerated type for the purpose of recording the left/right moves.

```
type direction = Left | Right ;;
```

Thus, for the tree `tr` defined above (depicted in Figure 3(b)), the Gorn address of the root is `[]` and the Gorn address of the node containing the item 9 is `[Left, Right, Right]`.



(a)

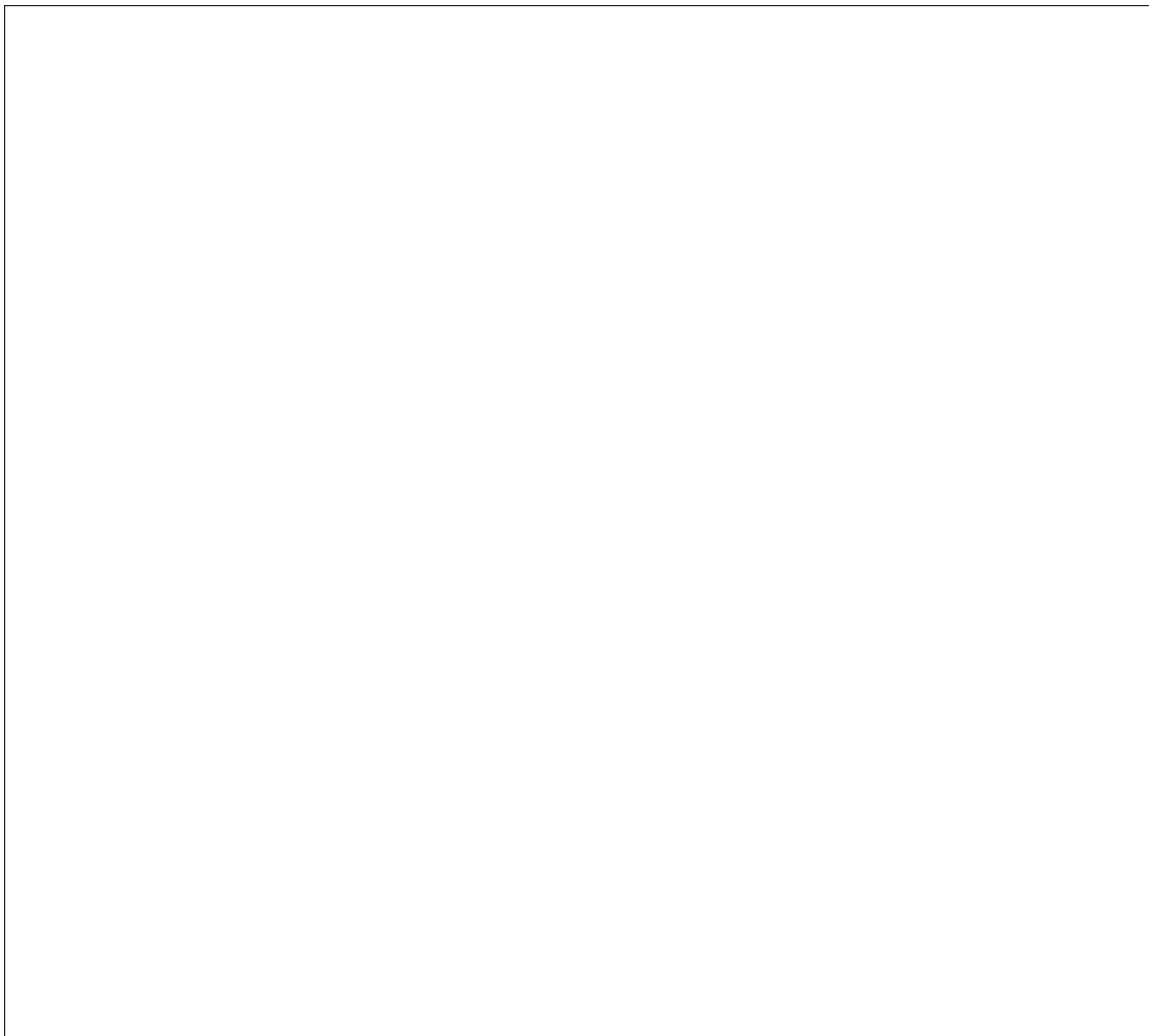


(b)

FIGURE 3. (a) Saul Gorn with the UNIVAC-1 computer at the University of Pennsylvania, 1950s. (b) A binary search tree, generated by inserting into an empty tree the integers 10, 5, 15, 7, and 9 in that order. The small circles indicate empty nodes. The Gorn address of the 9 node is `[Left, Right, Right]`, as indicated by the highlighted path.

**Problem 9. [10 points]** Define a function `gorn : 'a -> 'a bintree -> direction list` that given an item and a binary search tree returns the Gorn address of the item in the tree. It should raise a `Failure` exception if the item doesn't occur in the tree. For instance,

```
# gorn 9 tr ;;  
- : direction list = [Left; Right; Right]  
# gorn 10 tr ;;  
- : direction list = []  
# gorn 100 tr ;;  
Exception: Failure "gorn: item not found".
```





**Problem 10. [6 points]** Define a function `find` : 'a -> 'a bintree -> bool that given an item and a binary search tree returns `true` if the item is in the tree and `false` otherwise. You may want to use the `gorn` function in defining `find`. Examples of the `find` function in use include:

```
# find 9 tr ;;  
- : bool = true  
# find 100 tr ;;  
- : bool = false
```



End of exam.

EXTRA SPACE FOR ANSWERS

Reference this area with "See box 1"

Reference this area with "See box 2"

Reference this area with “See box 3”

Reference this area with “See box 4”

Reference this area with "See box 5"

Reference this area with "See box 6"