CS51 **Spring 2024**

Code Review 1: Intro to CS51

1 About OCaml

- `OCaml` is a **statically-typed** language: the types of expressions are determined at *compile time*—the time window during in which a program's source code are converted into binary instructions—rather than a program's *runtime* when the program is executing.
- `OCaml` is also an **implicitly-typed** language, so the compiler/interpreter can infer the types of some[1] expressions at compile time without explicit type declarations from the programmer.

1.1 Running OCaml Code

There are several ways to run an `OCaml` code, notably:

1. Compile and run code from an `OCaml` *source file*, which contains OCaml syntax (see Figure 1).
2. Define functions and evaluate expressions in the `OCaml REPL` (read-eval-print-loop), an environment that allows a user to define `OCaml` expressions and execute them line-by-line from an interpreter (see Figure 2).

```
// example.ml
let hello () =
    Printf.printf
        "Hello, world!\n" ;;
hello() ;;
```

```
$ ocamlbuild example.byte
$ ./example.byte
$ Hello, world!
```

Figure 1: Running `OCaml` Code from a source file. We compile a source file containing `OCaml` code (`example.ml`) into an executable byte file (`example.byte`) by running `ocamlbuild example.byte`, and we run the executable by executing `./example.byte`.

[1] Robust type inference algorithms (procedures that detect types of values) are often difficult to implement, so no compiler in practice can perfectly detect all types.

```
// example.ml
let add x y =
    x + y ;;
```

```
$ ocaml
# let plus_5 x = x + 5 ;;
val plus_5 : int -> int = <fun>
# plus_5 7 ;;
- : int = 12
# 3 * 2 + 5 ;;
- : int = 11
# #use "example.ml";;
val add : int -> int -> int = <fun>
# add 5 2 ;;
- : int = 7
```

Figure 2: Running `OCaml` Code via the `REPL`. We start the OCaml `REPL` by running `ocaml`. Another option is using `utop`, which provides some additional features over the traditional `REPL`. We can directly define functions (e.g. `plus_5` and evaluation expressions). We can also load `OCaml` defined code from a source file (e.g. `#use "example.ml"`) and evaluate user-defined expressions in the `REPL`.

1.2 Values and Types

`OCaml`'s expressions have *values* and *types*. **Atomic** types (e.g. `int`, `float`, `bool`, `strings`) are the simplest types defined in `OCaml`. **Composite** types (e.g. lists, tuples, records) are data structures composed of one or more values.

Data Type	Example Value
`tuple`	`(1, "hello", true)`
`list`	`[1; 2; 3]`
`record`	`{name="Alice"; age=25}`

Table 1: `OCaml` Composite Types. Note that lists can only contain elements of the same type, but records and tuples can contain elements of any type.

1.2.1 Operators

- **Operators** are symbols in `OCaml` that support operations over values. Operators can also be used as functions by wrapping parenthesis around the operator (e.g. `(+), ( * )`).
- **Infix** operators are those that can appear in an *infix* position (between two expressions) such as the arithmetic expressions `+`, `*`, `-`, `/`, etc. **Prefix** operators are those that can appear before an expression such as `-`.

Definition 1.1 (Operator Associativity)**. Operator Associativity** refers to the order in which operators of the same *precedence* are grouped in the absence of parentheses.

- Operators can be **left-associative** (operations are evaluated from left-to-right) or **right-associative** (operations are evaluated from right to left).

In `OCaml`, the `+` operator is left-associative, so an expression `5 + 2 + 7` evaluates to `(5 + 2) + 7` mathematically. The boolean-or operator `||` is right-associative, so `true || false || 5 = 2` is equivalent to `true || (false || 5 = 2)`.

Definition 1.2 (Operator Precedence)**. Operator Precedence** refers to the order in which different operators in expressions are grouped.

For instance, we say that the `*` operator has higher precedence than the `+` or `-` operator, so an expression `2 + 5 * 6` is equivalent to `2 + (5 * 6)`.

- The OCaml Manual defines the precedence and associativity of operators in the `OCaml` language.
- OCaml is a **strongly-typed** language, so compilers and interpreters cannot make implicit conversions between types [2].

1.2.2 Type Expressions

Type expressions describe the types of `OCaml` expressions. Examples of type expressions include `int`, `float`, and `bool` for atomic types, and `int * string`, `int list` and `bool -> unit` for tuples, integer lists, and functions.

1.2.3 Constructors

`OCaml` describes how to construct data structures via *constructors*.

Definition 1.3 (Constructors)**.**

- **Value Constructors** show us how to compose data structures from parts.
- **Type Constructors** show us how to construct type expressions for those data structures.

For instance, for tuples, the infix comma `,` is a value constructor while the "cross" symbol `*` is a type constructor.

	Tuples		*Records*	*Lists*
element types	differing		differing	uniform
selected by	order		label	order
type constructors	⟨⟩ * ⟨⟩	⟨⟩ * ⟨⟩ * ⟨⟩ ⋯	{a : ⟨⟩ ; b : ⟨⟩ ; c : ⟨⟩ ; ...}	⟨⟩ list
value constructors	⟨⟩ , ⟨⟩	⟨⟩ , ⟨⟩ , ⟨⟩ ⋯	{a = ⟨⟩ ; b = ⟨⟩ ; c = ⟨⟩ ; ...}	[] ⟨⟩ :: ⟨⟩

Table 7.1: Comparison of three structuring mechanisms: tuples, records, and lists.

Figure 3: Tuples, Records, and Lists. Table 7.1 in the Course Textbook

1.2.4 Let Expressions

The `let` keyword in OCaml binds an expression to a variable. For instance, `let x = 5` means to bind the expression `5` to the variable `x`.

[2] e.g. a `Python` interpreter allows you to add an int and a float or an int and a bool because it makes an implicit conversion of the int to a float and bool to an int for you. Expressions like `5 + 2.5` and `5 + True` are valid in a **weakly-typed** language like `Python`, but not in `OCaml`. In `OCaml`, the `+` operator can only be used to add two integers; it can't be used to add an int and a float.

- In OCaml, we can define expressions of the following form: `let <var> : <type> = e1 in e2`. We bind `e1` to `var`, which we can use in expression `e2`.

For instance, `let x : int = 5 in x + 2` evaluates to 7, since we bind the expression `5` to `x` and use it in the expression `x + 2`.

1.2.5 Functions

Functions map input values of one type to output values of another type. For example, a function of type `int -> bool` maps integers to booleans. In OCaml, functions are treated as values. Thus, expressions can evaluate to functions, functions can take in other functions as arguments, and functions can return functions.

- In OCaml, functions are **curried**, so each function takes in exactly one argument.
- We use the `fun` keyword to define functions. Consider the function `fun x -> fun y -> x + y`. This function takes in an argument `x` and returns a function that takes an argument `y` and adds `x + y`. We call this an **anonymous function**, a function with no name.
- We can assign a function a name by using the `let` construct. `let add : int -> (int -> int) = fun x -> fun y -> x + y` binds the function to the name `add`.
- We can use functions by *applying* them to an argument. For instance, we might evaluate `add 5 2` or `(fun x -> fun y -> x + y) 5 2`. Note that function application is *left-associative*, so `add 5 2` is equivalent to `(add 5) 2`.

Syntactic sugar, syntax that makes programs easier to read, allows us to more compactly define functions. For example, `let add x y = x + y` is syntactic sugar for the function defined above.

Remark 1.4 (Type Annotations)**.** It is good practice to **make our intentions clear** when programming. OCaml allows us to provide type annotations when defining functions. **In this class, you should always include type annotations for top-level functions**, as illustrated below.

Listing 1: Type Annotations

```
1 let add (x: int) (y : int): int =
2     x + y ;;
```

1.3 Abstract and Concrete Syntax

So far, the OCaml expressions we have seen have been **concrete syntax** expressions, expressions written in an unstructured linear form, such as the expression `3 + 4 * 5`. However, these concrete syntax expressions are potentially ambiguous: for example, do we interpret the expression `3 + 4 * 5` as `(3 + 4) * 5` or as `3 + (4 * 5)`?

To help us better understand how concrete syntax expressions are interpreted in OCaml, we can represent expressions as abstract syntax trees (ASTs).

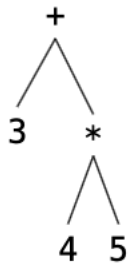

Figure 4: Abstract Syntax Tree of `3 + 4 * 5`

We read the abstract syntax tree shown in Figure 4 from the bottom-level. We first perform the operation `4 * 5` (by standard order of operations). Then, we add that result to 3. On the other hand, the abstract syntax tree for `(3 + 4) * 5` would look like:

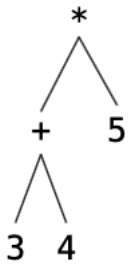

Figure 5: Abstract Syntax Tree of `(3 + 4) * 5`

Clearly, the AST for `(3 + 4) * 5` is different than that for `3 + 4 * 5`. It tells us that we first evaluate `3 + 4` and then multiply the result by `5`.

2 Recursion

In programming, we often define **recursive functions**, functions that call themselves.

2.1 Abstraction

One of the most common techniques in designing algorithms[3] are *reductions*: reducing a problem X to a problem Y by using an algorithm that solves problem Y as a "black box" or subroutine[4]. In terms of writing functions, this would mean using a function Y as a "helper function" in a function X.

Reductions leverage **abstraction**, the concept of generalizing concrete details away to focus on more important details. In terms of reductions, since we're trying to define a function X, we don't focus on the implementation details of the function Y, and we simply use it in the function X as a subroutine we assume to be correct.

[3]For the purposes of this class, we can use functions and algorithms, interchangeably. More precisely, you can think of algorithms as a recipe for how to perform a computation, while a function is the code that actually performs the computation.

[4]https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf

You have already seen many examples of the ideas of abstractions and reductions. For example, consider the `evens` function in Lab 2.

Listing 2: Abstraction Example

```
let evens : int list -> int list =
  List.filter (fun n -> n mod 2 = 0) ;;
```

- Here, we solve the problem of returning the even numbers in a list by reducing to the problem of filtering elements in a list that satisfy a certain predicate.
- We use the `List.filter` List Module function (code that the OCaml developers wrote) as a helper function.
- We don't care about the implementation details of `List.filter`; we only care about the function signature (its parameters and types), and we use it freely in the `evens` function, assuming that it works.

2.2 How to Think About Recursion

Recursion is a specific kind of reduction, where we reduce a problem X to a problem Y; in other words, we reduce a problem to **smaller instances of the same problem**. We can think of recursion in the following manner (taken from *Algorithms* by Jeff Erickson)[5]:

1. If we can solve an input of a problem directly, then solve it directly.
2. Otherwise, reduce the problem to smaller inputs of the same problem.

When defining recursive functions, we often define **base cases**—inputs which we can solve directly—and a **recursive step**—rules that break larger instances of a problem into one or more smaller instances that can be solved with **recursive calls** (where the function calls itself).

For example, when defining recursive functions that operate on lists, our base case is often the empty list (i.e. `[]`). In the recursive step we call our function on the subparts of the list (e.g. the *tail* of the list——the list excluding the first element).

Remark 2.1 (Recursion Tips)**.** For many students, recursion can be confusing at first, especially the act of self-referencing a function we're defining. Here are some tips for writing recursive functions.

1. Start by defining **base cases**. Remember that **base cases** are instances that should be solved directly, without making a recursive call to the function. In determining what the base cases should be, it helps to **write test cases before writing the function**, which also helps determine what smaller instances to reduce to in the recursive step.
2. In the recursive-step, **assume that a recursive call of the function will correctly solve subproblems**. Many students find using helper-functions intuitive. We suggest that **you think of the recursive function as a helper function** that **can solve smaller problems**![6]
3. Make sure you are using **types** correctly. There's a saying that in `OCaml`, your code is likely correct if you can get it to compile (due to the language's **strong-typing**). In the function body,

[5] https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf

[6] If your recursive call didn't correctly solve smaller subproblems, then the entire function wouldn't be correct! Therefore, after you've defined your base cases, you should assume that the recursive call will return the correct output just as you would assume a helper function is correct.

make sure that you are calling your recursive function on the correct types. If your recursive function takes in a list, make sure that it's being called on another list. This advice applies to programming beyond writing recursive functions.

2.3 OCaml Recursive Functions

Remark 2.2 (Recursive Definitions)**.** In `OCaml`, to recursively define an expression, you must use the `rec` keyword.

Listing 3: Recursion Example: Factorial

```
let rec factorial (x : int) : int =
  if x < 0 then raise (Invalid_argument "Input must be non-negative")
  else if x = 0 then 1
  else x * factorial (x - 1) ;;
```

2.4 Mutual Recursion

Definition 2.3 (Mutual Recursion)**.** Functions can be mutually recursive such that they refer to one another. Common use cases for mutual recursion are data parsing and formatting for data structures like arrays and dictionaries.

However, in OCaml, all values must be defined in advance, so typically functions can only refer to other values defined above them. To get around this issue, we define two functions as mutually recursive using the `and` keyword.

Example 2.4 (Even and Odd)**.** Consider the definition of these functions determining rather a number is an even or odd.

Listing 4: Even and Odd

```
let rec even (x: int) =
  if x = 0 then true
  else odd (x - 1)
and odd (x: int) =
  if x = 0 then false
  else even (x - 1)
```

3 Pattern Matching

The common mechanism for pattern matching in `OCaml` is the `match` statement. The general structure of a `match` statement is

```
match expr with
| pattern_1 -> expr_1
| pattern_2 -> expr_2
```

We can pattern match expressions that have atomic types or composite types.

- A **pattern** (e.g. `pattern_1`) is a `OCaml` language-defined template that allows us to conditionally match expressions to a value or bind parts of a data structure to variables, similarly to how we use the `let` keyword bind expressions to variable names.

- If the expression matches to `pattern_1`, we execute `expr_2`; if the expression matches to `pattern_2`, we execute `expr_2`.

Listing 5: Pattern Matching Example

```
let rec length (xs : int list) : int =
  match xs with
  | [] -> 0
  | _hd :: tl -> 1 + length tl ;;
```

In the example above, we use one pattern (i.e. `[]`) to check if the list is empty and return `0`. Otherwise, the pattern `_hd :: tl` binds the first element in the list to `hd` and the rest of the list to `tl`. **Please read Chapter 7.3** of the textbook to learn more about the details of pattern matching, and the OCaml Manual provides good supplementary material on what types of pattern matching we can do in `OCaml`.

3.1 Let Decomposition

We can also decompose composite types, particular tuples and records, using the `let` construct, as it allows us to bind expressions to variable names.

Listing 6: Let Decomposition

```
let addpair (pair : int * int) : int =
    let x, y = pair in x + y ;;
```

Remark 3.1 (Style and Pattern Matching)**.** Although pattern matching is a powerful technique, in some cases, it is stylistically better to avoid pattern matching. As noted in the *Design and Style Checklist* Section 7.2, we should avoid pattern matching when there is only a single pattern, as in the case with tuples. After all, this defeats the purposes of pattern matching, as there is only one pattern.

We **suggest** you take a look at the other points in the checklist for suggestions on how to effectively use pattern matching, in addition to the other sections.

4 Unit Testing

Once we have written functions, we want to verify the correctness of our functions.

- In CS51, you will mostly test the behavior of functions using **unit testing**, in which we test small pieces of code on a series of inputs and compare the function's output to the expected output.
- Note that **unit tests** are not a proof that a function is correct; of course, just testing a function on a subset of possible inputs does not provide a guarantee that the function is correct.
- However, **when you write unit tests**, you should **try to select cases that are most representative of all possible cases**.

In your problem set, you will be using the `unit_test` function provided by the `Absbook` module.

For concreteness, the `unit_test` function could be defined in the following manner (from the course textbook):

Listing 7: Unit Testing

```
let unit_test (test : bool) (msg : string) : unit =
    if test then
        Printf.printf "%s passed\n" msg
    else
        Printf.printf "%s FAILED\n" msg ;;
```

`unit_test` is a function of type `bool -> string -> unit`, where you provide a boolean expression to check the behavior of some code and a string to name the test.

For example, here is how we might write unit tests for the `fact` and `sum`, a function that computes the sum of the elements in a list.

Listing 8: Unit Testing: Factorial Example (from Section 6.7 of Textbook)

```
let fact_test () =
unit_test (fact 1 = 1) "fact 1";
unit_test (fact 2 = 2) "fact 2";
unit_test (fact 5 = 120) "fact 5";
unit_test (fact 10 = 3628800) "fact 10" ;;

let sum_test () =
unit_test (sum [] = 0) "sum empty list";
unit_test (sum [1;2;3] = 6) "sum lst of size 3" ;;

fact_test () ;;
sum_test () ;;

/*
fact 1 passed
fact 2 passed
fact 5 passed
fact 10 passed
sum empty list passed
sum lst of size 3 passed
*/
```

Here, we define unit tests inside `fact_test`. We separate our unit tests using the sequencing operator, `;`. We'll discuss this more later in the course, but just think of the `;` operator as allowing us to execute functions sequentially: for example, `func1; func2` evaluates function 1 then function 2.

After defining `fact_test`, we then make a function call `fact_test()` on line 13. How one structures unit tests depends on the individual, though we recommend defining separate test functions for each function.

5 Practice Problems

Complete the value and type questions without use of the `REPL`. Do not use the `OCaml` List Module for questions that ask you to define a function.

Problem 5.1 (Values and Types)**.** What is the type and value of each of the following OCaml expressions?

1. `( ** ) 3. 2.`
2. `let f x = if x then "true" else 42 in f (3 = 4)`
3. `fun x -> let term = 7.3 in x +. term`
4. `let f x y = (x + y, x * y) in f 3 7`

Problem 5.2 (Infix Operators)**.** Define an infix operator `+/.` that computes the average of two-floating point numbers. For example,

```
1.0 +/. 2.0 = 1.5
3.5 +/. 0.0 +/. 5.5 = 3.625
```

Problem 5.3 (Fibonacci)**.** Define a function `fib` such that `fib n` is the `n`th number in the Fibonacci sequence, which is 1, 1, 2, 3, 5, 8, 13...

We can write the Fibonacci as a **recurrence**, a mathematical equation that states how the `n`th term of a sequence is equal to some combination of the previous terms.

`fib 1 = 1`
`fib 2 = 1`
`fib n = fib (n-1) + fib (n-2)` for `n > 2`.

Problem 5.4 (Subset Sum)**.** Define a function `subset_sum` that checks if there exists a subset of a list with a given sum (called the *target*).

```
subset_sum 6 [1; 4; 3; 3; 2] = true
subset_sum 20 [4;5;2;7] = false
```

Problem 5.5 (Partition)**.** Define a function `partition` that divides a list into two lists, such that the first list contains elements that satisfy a predicate function `f` and the second list contains elements that don't satisfy the predicate. You may assume the input list is an integer list.

```
# partition ((>) 3) [4;8;2;-3;5;9]
- : int list * int list =
([2;-3], [[4;8;5;9])
```

Problem 5.6 (Pack Consecutive Duplicates)**.** Define a function `pack` that packs consecutive duplicates of list elements into sublists. You may assume that the input lists are string lists.

```
# pack ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "d"; "e"; "e"
   ; "e"; "e"];;
- : string list list =
[["a"; "a"; "a"; "a"]; ["b"]; ["c"; "c"]; ["a"; "a"]; ["d"; "d"];
 ["e"; "e"; "e"; "e"]]
```

Problem 5.7 (Unit Testing)**.** Write a suite of unit tests for `fib`, `subset_sum`, `pack`, and `partition`.

Problem 5.8 (Chess)**.** In chess, we move pieces across a 8x8 board such that the columns are labeled as letters and the rows are labeled as numbers.

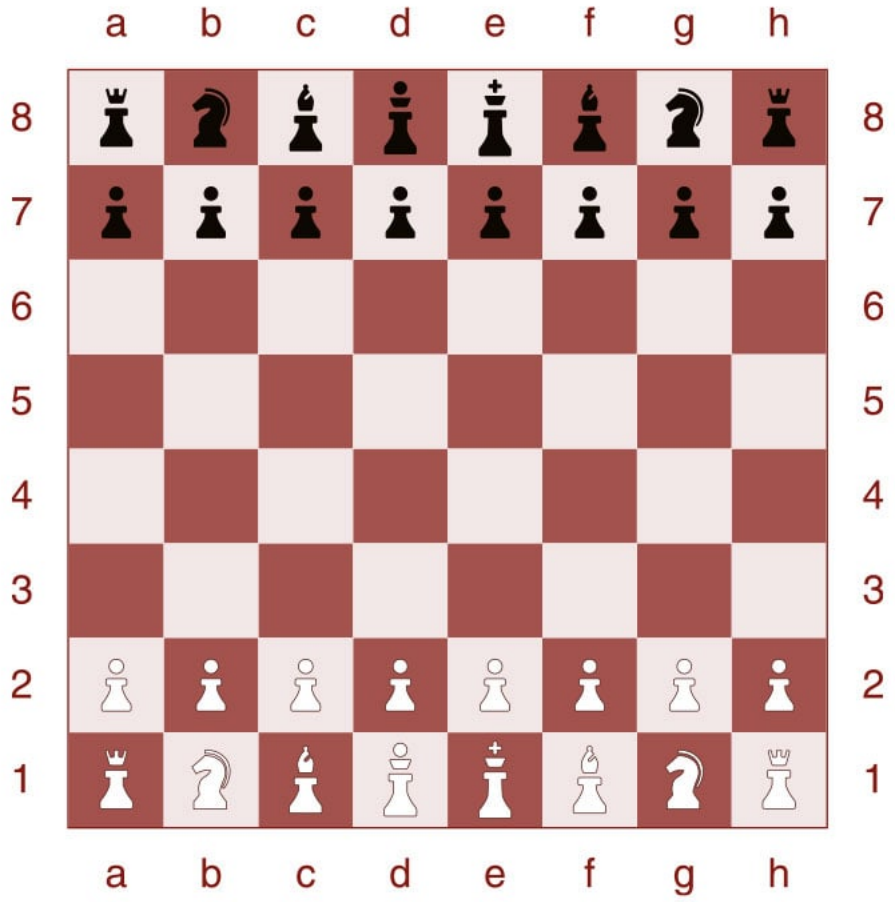

Figure 6: Diagram of chess board

Positions on a chess board are referred to as a combination of the column letter and row number. For instance, for the white pieces, we can say that the center pawn on the white square is at "e2".

1. Suppose, we want to define a type `position` that represents the position of a piece on the board. What type expression is appropriate to represent a piece's position?
2. After defining `position`, define a function `count_position` that takes in a `position` and a `position list`, and counts the number of times the `position` occurs in the `position list`.
3. A knight in chess can move two squares vertically and one square horizontally, or two squares horizontally and one square vertically (in an "L" shape). Define a function `legal_knight_moves` that takes in a `position` and returns a `position list` representing all the possible positions the knight can move to.
4. Define a function `min_knight_steps` that takes in a starting `position s` and a target `position t`, and returns the minimum number of moves it takes for a knight to go from `s` or

`t`. Note, that a knight can visit any position on the board from any starting position, so there's always a solution! See the Knight's tour.