Foundations of Computer Science

Supervision 1

Dima Szamozvancev and Michael Lee

1 Introduction to programming

1.1 Representations

Prerequisites: Lecture 1

Representation is one of the key parts of programming. In my experience, choosing (or, for novel problems, *inventing*) the right representation makes the “correct” / “best” solution obvious.

For this (and the next) section, I have created a `representations` Dune project, for you to work in. Please format your answers in (for example) a PDF document before submission, rather than submitting the modified project.

If you get stuck with at any point in the next two sections, or you don’t understand my instructions please email me ASAP.

1. One solution to the year 2000 bug mentioned in Lecture 1 involves storing years as two digits, but interpreting them such that 50 means 1950, 0 means 2000 and 49 means 2049.

   a) Consider `representations/lib/twoDigit.ml`. The `lib` directory of a dune project contains library code (think helper code that you want to write and use repeatedly). Execute these commands:

   ```
   cd representations
   dune build
   ```

   You should see an error.

   ```
   File "lib/twoDigit.ml", line 1:
   Error: The implementation lib/twoDigit.ml
         does not match the interface lib/.representations.objs/
     byte/representations__TwoDigit.cmi:
         The type `year' is required but not provided
         File "lib/twoDigit.mli", line 1, characters 0-9: Expected
     declaration
         The value `make' is required but not provided
         File "lib/twoDigit.mli", line 2, characters 0-22: Expected
      declaration
         The value `get' is required but not provided
         File "lib/twoDigit.mli", line 3, characters 0-20: Expected
      declaration
         The value `is_lteq' is required but not provided
   ```

```
    File "lib/twoDigit.mli", line 4, characters 0-33: Expected
declaration
    The value `add' is required but not provided
    File "lib/twoDigit.mli", line 5, characters 0-28: Expected
declaration
```

The complaint is that some types and values are not provided in `twoDigit.ml`: we'll fix that by answering the following questions.

b) To fix the first warning, choose a representation for the "year" type that allows years to be stored as two digits. You can do this by writing

```
type year = ?
```

For example, if I wanted to choose an integer representation, I would write

```
type year = int
```

c) To fix the next two warnings, code OCaml functions `make` and `get`. `make` should take an integer and return a `year`, throwing an Exception if the integer is `< 1950` or `> 2049`. `get` should take a `year` and return an integer.
You can define an Exception in OCaml as follows:

```
exception YearOutOfBoundsException
```

and raise it as follows

```
raise YearOutOfBoundsException
```

d) To fix warning 4, code an OCaml function `is_lteq` to compare two years (just like the `<=` operator compares integers).

e) To fix warning 5, code an OCaml function, `add: year -> int -> year`, to add/subtract some given number of years from another year. You should throw an exception if the result is `< 1950` or `> 2049`.

f) To test your answers, you should write some programs!
Write your answers in `representations/bin/dates.ml`. You can execute `dates.ml` by running `dune exec dates` in the terminal.
Alternatively, you can navigate to the `representations/bin` directory, and execute `dune utop` in the terminal. This command opens a REPL (read-eval-print loop), an interactive environment for executing OCaml. Once you've opened the REPL, run `open Representations;;` and `open TwoDigit;;`. This imports your code. Remember to end every statement with `;;`. Use `##quit;;` to quit the REPL once you're done.

g) Comment on the merits and demerits of the two digit representation. In what situations might the two digit representation be useful?

1.2 Abstraction Barriers

Prerequisites: Lecture 1

The choice of representation heavily influences how you think, and the code you write. An abstraction barrier *abstracts*, or *parameterises* over the choice of representation, hiding the representation. In this section, we'll see a concrete example of an abstraction barrier in OCaml, and reflect on why they are important.

In this section, we will continue to use the `representations` project.

1. In OCaml, abstraction barriers are enforced by *signatures*.
   a) There should be a `representations/lib/twoDigit.mli` file, that contains the following code:

```
type year
val make : int -> year
val get: year -> int
val is_lteq: year -> year -> bool
val add: year -> int -> year
```

   This is known as a *signature*. The signature of a module exposes certain types and values for other modules to access (In Part IA Object Oriented Programming, you'll learn about *access modifiers*: signatures play a similar role). This particular signature for the `TwoDigit` module tells other modules that they can rely on a type `year`, and four functions, `make`, `get`, `is_lteq`, and `add`. Your answers to question 1 provided an implementation, or *structure*, for this signature.

   b) Extend your earlier tests with a test that tries to manipulate the underlying representation of `year`s. For example, if `year`s are represented as integers, you could try to manipulate the integer by writing

```
(* ... *)
open TwoDigit;;
(make 1950) + 1;;
```

   Does OCaml let you write this expression? What is being abstracted over?

   c) In `TwoDigit.mli`, replace

```
type year
```

   with

```
type year = int (*or whatever type you chose to use for the
   representation*)
```

Now repeat part (c) and report on your results. What is the difference between `type year` and `type year = int`?

2. a) Repeat Questions 1 of Representations, but using a four digit representation of years. Your code should throw an error for years `< 1950` and `> 2049`.
   Create a `FourDigit.ml` and `FourDigit.mli` files in `representations/bin`. `FourDigit.mli` should be exactly the same as `TwoDigit.mli`. `FourDigit.ml` should implement the `year` type, and the `make`, `get`, `is_lteq` and `add` functions, but using a 4 digit representation rather than a 2 digit representation.

   b) Modify your test code to use `FourDigit` rather than `TwoDigit`. That is, replace `open TwoDigit` with `open FourDigit`. Run the tests, as before. Do you expect to have to make any changes? Why, or why not?

3. Using your experience from Questions 1 and 2, what is the main idea behind *abstraction barriers*? Why are they useful?

1.3 Floating Point Imprecision

Prerequisites: Lecture 1

If programming boils down to picking the right representation, what representations can you pick? A nice starting point might be "the representations I know from A level mathematics", like the integers, the naturals, the reals, etcetera. Sadly, that's not quite correct. Mathematics lets you deal with **infinite** structures. Computers, however are **finite**. Hence, in practice, we are restricted to finite representations. In this section, we'll look at the infinite real numbers: $\mathbb{R}$, which can only be **approximated** using floating point numbers.

1. Because computer arithmetic is based on binary numbers, simple decimals such as 0.1 often cannot be represented exactly. Write a function that performs the computation

$$\underbrace{x + x + \cdots + x}_{n}$$

   where x has type `float`. (It is essential to use repeated addition rather than multiplication!) See what happens when you call the function with $n = 1000000$ and $x = 0.1$.

2. Another example of the inaccuracy of floating-point arithmetic takes the golden ratio $\varphi = 1.618\ldots$ as its starting point:

$$\gamma_0 = \frac{1+\sqrt{5}}{2} \qquad \text{and} \qquad \gamma_{n+1} = \frac{1}{\gamma_n - 1}$$

   In theory, it is easy to prove that $\gamma_n = \cdots = \gamma_1 = \gamma_0$ for all $n > 0$. Code this computation in OCaml and report the value of γ_{50}. *Hint*: in OCaml, $\sqrt{5}$ is expressed as `sqrt 5.0`.

1.4 Functional Programming

Prerequisites: Lecture 1

If you've used Python, or C, before, you'll be used to writing code that performs some sort of side effect, but doesn't actually return anything useful. Here's one example:

```
int x;
x = 3;
```

In OCaml, this sort of programming style is not very supported: you should always be thinking about what an expression returns!

1. Why is it silly to write an expression of the form `if b then true else false`? What about expressions of the form `if b then false else true`? How about `if b then 5 else 5`?
2. Briefly discuss the meaning of the the terms *expression*, *value*, *command* and *effect* using the following examples:
   - `true`
   - `57 + 9`
   - `print_string "Hello world!"`
   - `print_float (8.32 *. 3.3)`
3. Which of these is a valid OCaml expression and why? Assume that you have a variable `x` declared, e.g. with `let x = 1`.
   - `if x < 6 then x + 3 else x + 8`
   - `if x < 6 then x + 3`
   - `if x < 6 then x + 3 else "A"`
   - `x + (if x < 6 then 3 else 8)`
4. Comment on the similarities and differences between:
   a) The `if` conditional in OCaml
   b) The `if` conditional in Python
   c) The `if` conditional in Java (if you have learnt about it)
   d) The `a ? b : c` ternary operator in Java (if you have learnt about it)

2 Recursion and Iteration

Prerequisites: Lecture 2

A recursive function is one that calls itself. Recursion is one of the key concepts in programming, especially in functional programming. Online, you will find many people comparing recursion to iteration. You will also hear from people that while recursion is more elegant, iteration is more efficient. In my opinion, these people are half-right. My hope is that by doing these exercises, you will understand *why* recursion is elegant, but also understand how recursion and iteration relate to each other.

1. The triangular numbers refer to the following sequence:

$$1, 3, 6, 10, 15, \ldots$$

Implement the `triangle` function in OCaml, such that `triangle n` calculates the n^{th} triangular number:

```
triangle 0 = 1
triangle 1 = 3
triangle 2 = 6
...
```

2. Code an iterative version of the efficient `power` function from Section 1.6.
3. Write an iterative version of your answer to the previous question in Python, using a loop. Try to make your answer look similar to your answer for Q2
4. Using your answer to the previous question, devise a procedure for converting a tail recursive algorithm into an iterative (as in, using a loop) algorithm. You can assume that we can identify the base cases and recursive cases of the tail recursive algorithm. Your procedure should explain how to treat:
   a) The arguments to the function (in the previous question, `x`, `n`, and `acc`)
   b) The base cases (in the previous question, when `n == 0`. In general, there might be more than one base case)
   c) The recursive cases (in the previous question, when `n > 0`. In general, there might be more than one recursive case).
5. Looking at the tail-recursive functions you've seen or written so far, think about why they are called *tail*-recursive: what is the common feature of their evaluation that would explain this terminology? If you have previous understanding of how functions are evaluated in a computer (stack frames), can you explain why tail-recursive functions are often more space-efficient than recursive ones?
6. Are there recursive functions that cannot be written in a tail recursive fashion? Justify your answer a little.
   Note: this is an extremely difficult question, and I don't expect an answer! Just have a think, give it your best guess. Try not to look it up, or ask an LLM: I want to see what *you* think.

3 Efficiency

Prerequisites: Lecture 2
A large section of computer science (not least, coding interviews) concentrate on *efficiency*: specifically, how algorithms scale with the size of the input data. In this course, we will understand the tools for reasoning about efficiency: big O notation, and recurrence relations.

1. Add a column to the table shown in Slide 206 with the heading *60 hours*.
2. Use a *recurrence relation* to find an upper bound for the recurrence given by $T(1) = 1$ and $T(n) = 2T(n/2) + 1$. You should be able to find a tighter bound than $O(n \log n)$. Prove that your solution is an upper bound for all n using mathematical induction.

3. Let $g_1, \ldots, g_k$ be functions such that $g_i(n) \geq 0$ for $i = 1, \ldots, k$ and all sufficiently large n. Show that if $f(n) = O(a_1 g_1(n) + \cdots + a_k g_k(n))$ then $f(n) = O(g_1(n) + \cdots + g_k(n))$.

4 Lists

Prerequisites: Lecture 3

Lists are one of the fundamental data structures in functional programming.

A list of type T is one of two possible things:

1. An empty list, `[]`
2. An element of type T (`x`) followed by some arbitrary list of type T (`xs`), `x::xs`

In this section, we will familiarise ourselves with the syntax of lists in OCaml, and grow comfortable with some basic operations on lists.

1. Explain why seeing the following expressions in OCaml code should be a cause for concern:
   - `1 :: [2, 3, 4]`
   - `"hello " @ "world"`
   - `xs @ [x]`
   - `[x] @ xs`
   - `hd xs + length (tl xs)`
2. We've seen how tail-recursion can make some list-processing operations more efficient. Does that mean that we should write all functions on lists in tail-recursive style?
3. Code a recursive and an iterative function to compute the sum of a list's elements. Compare their relative efficiency.
4. Code a function to return the last element of a non-empty list. How efficiently can this be done? See if you can come up with two different solutions.
5. Code a function to return the list consisting of the even-numbered elements of the list given as its argument. For example, given `[a,b,c,d]` it should return `[b,d]`. *Hint*: pattern-matching is a very flexible concept.
6. Code a function `tails` to return the list of the tails of its argument. For example, given the input list `[1, 2, 3]` it should return `[[1, 2, 3], [2, 3], [3], []]`.
7. a) Code a function `sum` that returns the sum of the elements of an integer list
   b) Code a function `mul` that returns the product of the elements of an integer list
   c) What can you say about your answers to parts a and b? Is there a way of sharing the majority of their implementations? Why might sharing be good?

5 Optional question

1. Consider the polymorphic types in these two function declarations:

```
let id x = x
val id : 'a -> 'a = <fun>
let rec loop x = loop x
val loop : 'a -> 'b = <fun>
```

Explain why these types make logical sense, preventing runtime type errors, even for expressions like `id [id [id 0]]` or `loop true / loop 3`.