Foundations of Computer Science
Solution Notes for Example Sheet 1

In this document you will find some solution notes for the problems of Example Sheet 1 of Foundations of Computer Science. If you find any mistake or any typos, please do let me know. Also, I am happy to hear (and include them in the notes (with credit) if you want) about alternative solutions to the problems or variations of a problem that you came up with.
The first supervision is about the core principles of Computer Science. This includes programming in OCaml, understanding the types of OCaml, understanding core principles of type systems, understanding how to measure efficiency of algorithms and proving properties of algorithms using the principle of induction.

1 Preliminaries

Remember to test the code that you write with examples. To deepen your understanding (and also as practice for the exams), it is good to try to first run the examples on paper.

Exercise 1 Write an OCaml function that converts degrees to radians and a function that converts radians to degrees.

```
let degToRad d = Float.pi *. d /. 180.;;
let radToDeg r = 180. *. r /. Float.pi;;

radToDeg Float.pi;;
degToRad 30.;;
radToDeg (degToRad 33.);;
degToRad (radToDeg 0.4);;
```

We can also make it a bit more efficient by precomputing the divisions `Float.pi /. 180.`.

```
(* We need to be careful how to choose the exact decimal representation. *)
let degToRad d = d *. 0.01745329251;;
let radToDeg r = r *. 57.2957795131;;

radToDeg Float.pi;;
degToRad 30.;;
radToDeg (degToRad 33.);;
degToRad (radToDeg 0.4);; (* Less precision than the code above. *)
```

Discussion:

- If you hardcoded `pi`, how many digits did you pick and why?
- Did you add any tests? Do these tests include some known values e.g., `rad(pi)`, extreme values and testing invertibility e.g., `rad(deg(33))` being close to 33?
- Are there any considerations for values outside of the ranges $[0, 360)$ and $[-\pi, \pi)$?

Exercise 2 Write an OCaml function that takes 3 integers and returns the largest one.

```
let max3 x y z = max (max x  y) z;;

max3 1 2 3;;
max3 1 3 2;;
max3 2 1 3;;
max3 2 3 1;;
max3 3 1 2;;
max3 3 2 1;;
max3 1 1 2;;
```

Discussion:

- Did you define a helper function?
- Did you test thoroughly?
- What is the difference between `max max x y z` and `max (max x y) z`.
- What is the fewest number of function calls you could make to `max` in order to implement a max function for 4 integers? What about 8? What is the general solution?

Exercise 3 Write an OCaml function that takes 4 integers and outputs the pair of integers with the maximum sum.

The idea is to select the two largest integers out of the 4 given values.

```
let max_sum4 w x y z =
  if w >= max3 x y z then (w, max3 x y z)
  else if x >= max3 x y z then (x, max3 w y z)
  else if y >= max3 x y z then (y, max3 x w z)
  else (z, max3 x y w);;

max_sum4 1 2 3 3;; (* (3, 3) *)
max_sum4 1 3 2 4;; (* (4, 3) *)
max_sum4 4 8 2 3;; (* (8, 4) *)
max_sum4 (-3) (-1) (-2) (-4);; (* (-1, -2) *)
```

Discussion:

- Does your algorithm work for negative integers?
- Does your algorithm work when the input has duplicate values?
- (Discussion) Sometimes it is easier to solve the more general version. What do you think?
- What do you think about the approach of sorting and adding the two largest values?

Exercise 4 Write an OCaml function that takes 3 integers and outputs the pair of integers with the maximum product.

```
let max_prod3 x y z =
  let mx = max3 (x * y) (y * z) (z * x) in
    if mx = x * y then (x, y)
    else if mx = y * z then (y, z)
    else (z, x);;

max_prod3 0 3 1;; (* (3, 1) *)
max_prod3 3 2 1;; (* (3, 2) *)
max_prod3 (-1) 0 3;; (* (-1, 0) *)
max_prod3 (-1) (-3) (-2);; (* (-3, -2) *)
max_prod3 (-3) (-3) 1;; (* (-3, -3) *)

(* To avoid duplicate computations *)
let max_prod3 x y z =
  let p1 = x * y in
  let p2 = y * z in,
  let p3 = z * x in
  let mx = max3 p1 p2 p3 in
    if mx = p1 then (x, y)
    else if mx = p2 then (y, z)
    else p3;;
```

Discussion:

- Does your algorithm work for negative integers?
- Does your algorithm work with duplicate values?
- (Discussion) Sometimes it is easier to solve the more general version.
- What do you think about the approach of sorting and multiplying the two largest values?
- Alternative solution: if zero/one negatives, pick largest two, else pick smallest two.
- (Minor) Why is the comparison approach slightly preferred to the multiplication approach? Multiplication is (constant times) slower than comparison. But this comes at the cost of making the code harder to read.

Exercise 5 [Sequences] Write an OCaml function to compute:

(a) The n-th term in an arithmetic sequence.

(b) The sum of the first n terms in an arithmetic sequence.

(c) The n-th term in a geometric sequence.

(d) The sum of the first n terms in a geometric sequence.

(e) The n-th term in a harmonic sequence.

(f) The sum of the first n terms in a harmonic sequence.

The question is underspecified and you could provide either integer or float implementations as long as you keep the types correct.

```
let a_nth a0 d n = a0 + d * n;;

a_nth 3 10 0;; (* 3 *)
a_nth 3 10 2;; (* 23 *)

(* See comment below for making n * (n-1) more precise *)
let sum_a_nth a0 d n = a0 + d * (n+1) * n / 2;;

sum_a_nth 3 10 2;; (* 39 = 3 + 13 + 23 *)
sum_a_nth 3 10 0;; (* 3 *)

let rec pow x n =
  if n = 0 then 1
  else if n mod 2 = 0 then pow (x * x) (n/2)
  else x * pow (x * x) (n/2);;

let g_nth a0 r n = a0 * pow r n;;

g_nth 3 2 4;; (* 48 = 3 * 2^4 *)
g_nth 4 3 2;; (* 36 = 4 * 3^2 *)
g_nth 4 3 0;; (* 4 *)

let sum_g_nth a0 r n =
  if r = 1 then a0 * (n + 1)
  else a0 * (pow r n - 1)/(r - 1);;

sum_g_nth 2 3 4;; (* 80 = 2 * (3^0 + 3^1 + 3^2 + 3^3) *)

let h_nth n = 1. /. float_of_int n;;

h_nth 2;; (* 0.5 *)
h_nth 3;; (* 0.333333333333 *)

let sum_h_nth n =
  let rec sum_h_nth_aux k acc =
    if k = 0 then acc
    else sum_h_nth_aux (k-1) (acc +. h_nth k)
```

```
  in
    sum_h_nth_aux n 0.0;;

sum_h_nth 2;; (* 1.5 *)
sum_h_nth 3;; (* 1.83333333 *)
```

Discussion:

- Check off-by-one errors in the arithmetic sequence and series.
- Use formula for the sums (except for the harmonic sum).
- Use the efficient power function for the geometric.
- Is the harmonic implementation tail recursive?
- Should you include checks for $n \geq 0$?

2 Lecture 1

Exercise 6 One solution to the year 2000 bug involves storing years as two digits, but interpreting them such that 50 means 1950 and 49 means 2049.

(a) Write an OCaml function that converts between this format and the 4-digit representation.

(b) Write an OCaml function to compare two years in this format.

(c) Write an OCaml function to add/subtract an integer amount of years from another.

[Exercises 1.1, 1.2 in Lecturer's handout]

(a)
```
let to_4digit x =
    if 0 <= x && x < 50 then x + 2000
    else if 50 <= x && x < 100 then x + 1900
    else raise (Invalid_argument "x should be in range [0, 99]");;

to_4digit 45;; (* 2045 *)
to_4digit 96;; (* 1996 *)
to_4digit 50;; (* 1950 *)
to_4digit 49;; (* 2049 *)
to_4digit 0;; (* 2000 *)
to_4digit 100;; (* Exception: Invalid_argument "x should be in range [0, 99]".
    *)

let to_2digit x =
    if 2000 <= x && x < 2050 then x - 2000
    else if 1950 <= x && x < 2000 then x - 1900
    else raise (Invalid_argument "x should be in range [1950, 2049]");;

to_2digit 1950;; (* 50 *)
to_2digit 2049;; (* 49 *)
to_2digit 1949;; (* Exception: Invalid_argument "x should be in range [1950,
    2049]". *)
to_2digit 2050;; (* Exception: Invalid_argument "x should be in range [1950,
    2049]". *)
to_2digit 2051;; (* Exception: Invalid_argument "x should be in range [1950,
    2049]". *)
to_2digit 2035;; (* 35 *)
to_2digit 2000;; (* 00 *)
to_2digit 1951;; (* 51 *)
```

Discussion:

- Check for extreme cases.

- Check for reporting an error.
- (Possibly) check invertibility.

(b) Either direct or indirect comparison is possible, with the second one being a bit less efficient but easier to implement.

```
let compare_dates x y = Int.compare (to_4digit x) (to_4digit y);;
```

Also, you don't need to look in the details of how OCaml defines comparison.

For the direct implementation, the reference is here, so if the dates are the same we return 0, if the first one is larger we return $+1$ otherwise -1.

```
(* Without check for validness of inputs. *)
let compare_dates x y =
   if x = y then 0
   else if x >= 50 then
      if y < 50 || y > x then -1
      else 1
   else
      if y >= 50 || x > y then 1
      else -1;;


compare_dates 10 17;; (* -1 *)
compare_dates 10 10;; (* 0 *)
compare_dates 97 42;; (* -1 *)
compare_dates 42 97;; (* 1 *)
compare_dates 22 50;; (* 1 *)
compare_dates 22 49;; (* -1 *)
```

(c)
```
let add_dates x d = to_2digit ( d + to_4digit x);;

add_dates 0 10;; (* 10 *)
add_dates 20 35;; (* Exception *)
add_dates 35 4;; (* 39 *)
add_dates 32 24;; (* Exception *)
```

Discussion:

- Check that it works for both positive and negative numbers.
- Check for reporting an error.

Extended Note 1 [Types and type inference] In OCaml, almost everything is an expression and all valid expressions must have a type. If you type an invalid expression, you get a compilation error. An expression is typically denoted by e (e_1, e_2 if there are many) and the type is denoted by T (T_1, T_2 if there are many). The type of the expression tells us the type of the result we get once we evaluate expression (if we get any). For example,

1. The expression $e =$ `10 + 20` has type *int* (sometimes we write this as $e : int$). When e is evaluated (or executed) it will produce the value 30 which has type *int*.
2. The expression $e =$ `let f x = x + 1` has type $int \rightarrow int$, meaning that e evaluates to a function takes integers as inputs and outputs integers.
3. The expression $e =$ `let f x = x > 3` has type $int \rightarrow bool$, meaning that e evaluates to a function that takes integers as inputs and outputs booleans.

When writing OCaml code, we can choose to explicitly state value types (e.g., `let add_int ((x:int), (y:int)) = x + y;;`), known as *type constraints* or allow the OCaml compiler to infer them.

Exercise 7

(a) Write two OCaml functions: one to compute the mean of three integers and one to compute the mean of three floats. What are the types of the two functions?

(b) Write two OCaml functions: one to compute the median of three integers and one to compute the median of three floats. What are the types of the two functions?

(c) (open-ended) Some programmers add type constraints to functions that don't need them (e.g., `let f(x:int) = x + 1;;` or `let g(x:int) = (x, x);;`) in OCaml? What are the benefits and what are the downsides?

[Source: [1999P1Q5]]

(a)
```
let mean_int x y z = float_of_int(x + y + z);;
val mean_int : int -> int -> int -> float = <fun>
```

(b)
```
let mean_float x y z = (x +. y +. z) /. 3;;
```

(c) In some cases, this makes the code easier to read and faster to review. Also, if you are writing code for a library, the type signatures of the functions you expose may be binding for future releases, and you may not want to provide support for the most general version that your code currently handles. (One more reason is that it removes the need for OCaml to infer the type of your program, which sometimes can be very slow (see here if you are interested).

The main downsides of doing this, is that the programmer spends time and effort on something that can be done automatically, the code becomes a bit harder to update, and the user of the function (potentially) does not get the full applicability of the function.

In languages like Haskell it is a best practice to prove the signature of the function.

Exercise 8 [Types] What are the types of `(+)` and `(+.)`? Why did the authors of OCaml choose to have two different operations? What alternatives did they have?

```
# (+);;
- : int -> int -> int = <fun>

# (+.);;
- : float -> float -> float = <fun>
```

This question is a bit open-ended. In SML (the predecessor of OCaml), it used to be that + could work for both integers and floats. For example, you could write:

```
fun f a = a + 3.0;;
> val f = fn: real -> real;
```

and also

```
fun g a = a + 3;;
> val g = fn: int -> int;
```

The compiler would start with giving + the default type `int` unless it noticed that one of its arguments is a float. So,

```
fun h a b = a + b;;
> val h = fn: int -> int -> int;
```

But,

```
fun k a b = a + (f b);;
> val k = fn: real -> real -> real;
```

The main advantage of this is that there are fewer operators. A disadvantage is that it makes the typing system of OCaml more complicated, as it needs to provide special treatment to the + operator.

Exercise 9 [Value declaration] What is the syntax for value declarations? Why are they useful?

Their syntax is `let <name> = <expression>;;` and `let <name> = e1 in e2;;`. The first is used for defining named expressions, such as functions. The second can be used to create a scoped value. This avoids recalculating some results.

Discussion:

- What is the output of the following code and why?

```
let v1 = 10 in
  let v2 = 20 in
    v1 * v2;;
```

- What is the output of the following code and why?

```
let v1 = 10 in
  let v2 = 20 in
    let v1 = 30 in
      v1 * v2;;
```

Exercise 10 [If-statement]

(a) What is the syntax for `if`-statements?

(b) What are the type restrictions that must hold for an `if`-statement? In your answer, consider the OCaml expression `if e1 then e2 else e3` with types $e_1 : T_1$, $e_2 : T_2$ and $e_3 : T_3$. What conditions must hold for the types T_1, T_2 and T_3?

(a) The syntax is `if e1 then e2 else e3;;` (and there is one special case `if e1 then e2`, if `e1` is of unit type).

(b) $T_1 : \mathit{bool}$, $T_2 = T_3$. For the special case, $e_1 : \mathit{bool}$ and $e_2 : \mathit{unit}$.

Exercise 11 [Function declaration]

(a) What is the syntax for function declarations?

(b) What are the type restrictions that must hold for the declaration of a function with one argument?

(a) The syntax is `let (rec) <function_name> p1 p2 ... = e;;`.

(b) If $p_1 : T_1$ and $e : T_2$, then $f : T_1 \to T_2$.

Exercise 12 [andalso and orelse]

(a) How would you implement andalso (`&&`) and orelse (`||`) using `if`-statements?

(b) How would you implement andalso (`&&`) and orelse (`||`) using functions? Why is it not equivalent to the native implementation?

(c) Why is the implementation defined to be equivalent to that of `if`-statements and not that of functions?

(*Hint:* If you get stuck, have a look at the official documentation.)

(a) We can replace `a && b` with `if a then b else false` and similarly, `a || b` with `if a then true else b`. This way `b` is evaluated only if it could change the resulting value of the expression.

(b) A function implementation would be something like

```
let andalso a b = a and b;;
let orelse a b = a or b;;
```

The problem with these is that the moment you call the function e.g., `andalso false (loop 1)`, the arguments are evaluated and so `loop 1` will be called first (even when it is not needed), so the function will terminate.

(c) In some cases, this is a matter of convenience, we want to check the bounds of some index and then test something with that index. This would not work with the function definition. Also, it is good for efficiency reasons to avoid checking `b` when it is not needed.

Discussion:

- (Slightly irrelevant) What does the following code do?

```
let f1 x = x + 3;;
let f2 x = x + 10;;
let a = 10;;

(if a > 4 then f2 else f1) 17;;
```

- Why should arguments be evaluated first? Can they be evaluated only when needed?

```
let print x = Printf.printf "Hello %d " x; x;;

(print 3; (+)) (print 2) (print 1);;

(* This is because the previous statement is equivalent to: *)
((print 3; (+)) (print 2)) (print 1);;

(* Actually the order of evaluation is not specified in the documentation
   both orderings are valid. *)
```

- Are these operators commonly used? Yes.

Exercise 13 [Recursive functions]
(a) What is a recursive function?
(b) What is the problem with the following function `let rec nsum n = n + nsum (n-1);;`?

(a) Recursive functions are functions that call themselves for some values of the inputs.

(b) The function does not have a base case. In a machine with unbounded memory, it would not terminate. In a typical machine (and without compiler optimisations) it will probably lead to stack overflow.

Discussion:

- What happens when we call the function `let rec nsum n = if n = 0 then 0 else n + nsum (n-1);;` for negative values of n? The function will loop until it reaches another positive integer. OR it will cause stack overflow. Depending on the system OCAML has 31-int or 63-bit int.

- What happens in the equivalent tail recursive version? It should not terminate (not cause a stack overflow)

Exercise 14 [Fibonacci sequence] The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$ (for $n > 1$) with $F_0 = 0$ and $F_1 = 1$.
(a) Compute the first 7 terms of the Fibonacci sequence.
(b) Write a recursive function that computes the n-th Fibonacci number. What is the largest value that it can compute in under a second? Can you make it more efficient?

(a) $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8$.

(b) The simplest recursive implementation is the following.

```
let rec fib n =
   if n = 0 then 0
   else if n = 1 then 1
   else fib (n-1) + fib (n-2);;

fib 6;;
(* Function has exponential running time. *)
```

One can speed it up using the following version:

```
let fib n =
   let rec fib_aux k a b =
      if k = 0 then b
      else fib_aux (k-1) b (a + b)
   in
      if n = 0 then 0
      else fib_aux (n-1) 0 1;;

fib 6;;
(* Function has exponential running time. *)
```

Discussion:

- Does your function always compute the correct answer? (Probably not if you are using ints)
- How efficient is your implementation?

Exercise 15 For each of the following expressions, indicate whether it compiles and if it does state its type.

(a) `let f x y = if x = y then x else 0`

(b) `let f x y = if x / y < 1 then x else y`

(c) `let f x y = if x /.  y < 1 then x else y`

(d) `let f x y = if x / y < 1.0 then x else y`

(e) `let f x y = if x < y then x else y`

(f) `let f x y = if x / y < 3 then x else 2.1`

(a) `val f :  int -> int -> int = <fun>`, because 0 is an integer so x must also be an integer.

(b) `val f :  int -> int -> int = <fun>`

(c) Does not compile because the types of `x /.  y` and `1` do not match.

(d) `x / y` makes the compiler infer that `x` and `y` are integers, so their result cannot be compared to `1.0`.

(e) `val f :  'a -> 'a -> 'a = <fun>`

(f) `x` is inferred to be an integer but then the `if` statement does not typecheck.

Discussion:

- Discuss how the OCaml compiler can infer the type of x and hence of the entire expression.
- For the third exception we get the following error message:

```
Error: This expression has type int but an expression was expected of type
         float
  Hint: Did you mean `1.'?
```

If we reverse the order of the arguments, we don't get the hint. Why do you think this is the case? Probably the compiler assumes the left type as correct and tries to match it to the right type. In this case, there is no simple fix for the right case, so it does not have any suggestions.

```
let f x y = if 1 < x /. y then x else y;;

# let f x y = if 1 < x /. y then x else y;;
Error: This expression has type float but an expression was expected of type
         int
```

Exercise 16 [Floating point precision] Because computer arithmetic is based on binary numbers, simple decimals such as 0.1 often cannot be represented exactly. Write a function `mul` that performs the

computation m

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

where x has type `float`. (It is essential to use repeated addition rather than multiplication!)
The value computed with `n = 10000` and `x = 0.1` may print as `1000.0`, which looks exact. If that happens, then evaluate the expression `mul 0.1 10000 - 1000.0`.
An error of this type has been blamed for the failure of an American Patriot Missile battery to intercept an incoming Iraqi missile during the first Gulf War. The missile hit an American Army barracks, killing 28 people.

[Exercise 1.5 in Lecturer's handout]

```
let rec mult n x acc =
  if n = 0 then acc
  else mult (n-1) x (acc +. x);;

mult 10000 0.1 0.0;;
 - : float = 1000.00000000015882
```

The reason why this happens is because 0.1 is stored imprecisely, so as we add up instances of 0.1 the errors accumulate.

Exercise 17 [Floating point precision] Consider the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$.
(a) Show that it satisfies $\phi = \frac{1}{\phi - 1}$.
(b) Write an OCaml function that computes a sequence of float numbers where the n-th term is given by $\gamma_n = \frac{1}{\gamma_{n-1} - 1}$ (for $n > 0$) and $\gamma_0 = \phi$. What do you notice? [*Hint:* in OCaml, $\sqrt{5}$ is expressed as `sqrt 5.0`.]

[Exercise 1.6 in Lecturer's handout]

(a)

$$\frac{1}{\phi - 1} = \frac{1}{\frac{1+\sqrt{5}}{2} - 1} = \frac{1}{\frac{-1+\sqrt{5}}{2}} = \frac{2}{-1+\sqrt{5}} = \frac{2(-1-\sqrt{5})}{(-1+\sqrt{5})(-1-\sqrt{5})} = \frac{2(-1-\sqrt{5})}{(-1)^2 - (\sqrt{5})^2} = \frac{2(-1-\sqrt{5})}{-4} = \phi.$$

(b)
```
let rec gamma n =
  if n = 0 then (1. +. sqrt 5.0)/. 2.
  else 1. /. ((gamma (n-1)) -. 1.);;

gamma 0;; (* 1.6180339887498949 *)
gamma 10;; (* 1.61803398875098181 *)
gamma 100;; (* -0.618033988749894792 *)
```

Discussion: What is the problem with the following code?

```
let rec gamma n =
  if n = 0 then (1. +. sqrt 5.0)/. 2.
  else 1. /. (gamma (n-1));;
```

Exercise 18 [Floating point] Explain the following outputs:
(a) `Int64.of_float(9007199254740993.0)` gives `9007199254740992L`.
(b) `3.0 +. 3E-90` gives `3.0`

(a) `9007199254740993 = 2^53 + 1` the numbers are 53-bits apart, so they cannot be represented in the mantissa of the floating point number and hence the most significant one is kept.

(b) The digits of `3` and `3E-90` are also very far apart and hence cannot be represented in the mantissa of the floating point number and hence the most significant one is kept.

Discussion:

- Discuss the importance of having a floating point standard like the IEEE 754 standard.
- Discuss the emergence of floating point standards for deep learning (e.g., Bfloat 16).

Exercise 19 [Operator precedence (+)]

(a) What is *operator precedence*?

(b) Which operator comes first *plus*, *minus* or *negation*? Explain the following results:

i. `3.0 +. -3.0 +. 3E-90` gives `3E-90`.

ii. `3E-90 +. 3.0 -. 3.0` gives `0.0`.

(c) Which operator comes first *function application* or *plus*? What should the results of the following be given that `let f x = x * 2`?

i. `f 2+3`

ii. `f f 2`

(a) All(?) operators in OCaml are binary. Operation precendence is the order in which operators are evaluated and this is a language choice. References in the OCaml standard

(b) The precedence is: `* , mod, +/-, @/^`
Associativity : right `(@, ^)`, left: `(+, -, >, <, function application)`
For `@` it matters because the computation time is less.

So `3.0 +. -3.0 +. 3E-90` is equivalent to `((3.0 +. -3.0) +. 3E-90)`, so the threes cancel out and `3E-90` remains. Similarly, `3E-90 +. 3.0 -. 3.0` is equivalent to `((3E-90 +. 3.0) -. 3.0)`, but the first addition gives `3`, which then results to `0`.

(c) This is why `f 2 2` does not compile, because it is converted to `(f 2) 2` instead of `f (f 2)`. On the other hand, `f 2+3` is equivalent to `((f 2)+3)`

Discussion:

- Do we need operators? (No, we could be using functions everywhere, it is mostly for readability)
- Why not use parentheses for everything?
- Brief discussion of abstract syntax trees.
- Do you know of any ternary operators in any programming language? Why do you think they are not used in OCaml?
- Is there a function `f` for which the expression `f f 2` type checks? Yes, e.g., `let f x = x;;`
- On the topic of floating point numbers, if you were given a list what would be a sensible way to add up the values? Add values that are similar in magnitude. (See also Kahan's summation algorithm.)

Exercise 20 [Infix operators (Optional)] Any function that takes two arguments can be registered as infix. Consider `let (+++) x y = x + y` and `let ( *** ) x y = x * y + 1`. The parentheses around the function names register the function as infix (only for function names with special characters). The precedence is determined by the first character of the function name. What do the following expressions return?

(a) `3 +++ 4 *** 5`

(b) `3 +++ 4 *** 5 *** 6`

(a) `3 +++ (4 *** 5) = 3 + (4 * 5 + 1) = 24`

(b) `3 +++ (4 *** (5 *** 6)) = 3 + (4 * (5 * 6 + 1) + 1) = 128`

Discussion:

- Why doesn't the following compile: `let (***) x y = x * y + 1`? Because OCaml parses it as a comment.

Exercise 21 [Environment] What do the following lines of code do? Why?

```
let a = 10;;
let f x = x + a;;
let a = 20;;
f 5;;
```

```
# let a = 10;;
val a : int = 10
# let f x = x + a;;
val f : int -> int = <fun>
# let a = 20;;
val a : int = 20
# f 5;;
- : int = 15
```

Extended Note 2 [Coding style] There are many valid coding styles (e.g.,, here and here) that one can follow. Following best principles and keeping the code simple is good for many reasons. Throughout the course you will see many coding practices that are better (in some way) than others. Here, I list two to get you into the spirit:

1. Avoid parentheses when they are not needed: `4 * r * r` evaluates to the same value as `((4) * r) * (r)`, but the second one adds unnecessary clutter.
2. Do not write `if ... then true else false` nor `if ... then false else true`. Why? What would you use instead?

(In general, coding style best practices are not carved in stone.)

3 Lecture 2

Extended Note 3 [Induction]
(You will see more of mathematical induction in the Discrete Maths course)
The principle of mathematical induction is a technique that can (among other things) be used to reason that a program is correct. If we want to prove that a statement holds over all positive numbers (in the Discrete Maths course this will usually be for all non-negative integers), then it suffices to show that:

1. it holds for $n = 1$ (base case), and
2. if it holds for $n = k$ then it also holds for $n = k + 1$

Example 1 Prove that

$$S(n) = \frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \ldots + \frac{1}{n \times (n+1)} = \frac{n}{n+1}.$$

Proof. We will start by proving the base case, for $n = 1$:

$$S(1) = \frac{1}{1 \times 2} = \frac{1}{1 \times (1+1)},$$

which holds trivially.
Assume that it holds for $n = k$, that is $\frac{1}{1\times2} + \frac{1}{2\times3} + \ldots + \frac{1}{k\times(k+1)} = \frac{k}{k+1}$, then we need to show that it holds for $n = k + 1$. For $n = k + 1$, the sum is written as

$$S(k+1) = \frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \ldots + \frac{1}{k \times (k+1)} + \frac{1}{(k+1) \times (k+2)}.$$

The first k terms are equal to the $S(k)$ term, so

$$S(k+1) = S(k) + \frac{1}{(k+1) \times (k+2)}.$$

Using the induction hypothesis,

$$S(k+1) = \frac{k}{k+1} + \frac{1}{(k+1) \times (k+2)}.$$

With simple algebraic manipulation we get,

$$\begin{aligned} S(k+1) &= \frac{k \cdot (k+2)}{(k+1) \times (k+2)} + \frac{1}{(k+1) \times (k+2)} = \frac{k \cdot (k+2) + 1}{(k+1) \times (k+2)} = \frac{k^2 + 2k + 1}{(k+1) \times (k+2)} \\ &= \frac{(k+1)^2}{(k+1) \times (k+2)} = \frac{k+1}{k+2} \end{aligned}$$

This shows that it also holds for $n = k+1$, so by the principle of mathematical induction it holds for all natural numbers. □

Example 2 Prove that the following program computes the sum of integers from 1 to n

```
let rec nsum n =
    if n = 1 then 1
    else n + nsum (n-1);;
```

Proof. Again, we start by proving the base case: For $n = 1$, the function gets into the first branch of the if-statement and returns 1, which is the correct answer.
Assume that it holds for $n = k$, that is that calling `sum_1_to_n k` returns $S(k) = 1 + \ldots + k$. Then we need to show that `sum_1_to_n (k + 1)` returns $S(k+1)$. Note that $S(k+1) = S(k) + (k+1)$ and $k+1$ is always greater than 1, so when the function is called it will enter the second branch. By the induction hypothesis, `sum_1_to_n k` returns $S(k)$. Therefore, second branch returns $k + 1 + S(k) = S(k+1)$. Hence, it holds for $S(k+1)$ and so it holds for all natural numbers. □

Note: The base case does not have to start with 0. Consider the following example:

Example 3 Prove that $2^n > 2n$ for every positive integer $n > 2$.

Proof. We begin the base case for $n = 3$. The LHS is equal to $2^3 = 8$ and the RHS is $2 \cdot 3 = 6$. So, the inequality holds.
Assume that it holds for $n = k$, that is $2^k > 2 \cdot k$ (induction hypothesis), then we need to show that $2^{k+1} > 2 \cdot (k+1)$. By multiplying the inequality obtained by the induction hypothesis, we have

$$2^k > 2 \cdot k \Rightarrow 2 \cdot 2^k > 2 \cdot 2 \cdot k \Rightarrow 2^{k+1} > 2k + 2k > 2k + 4 > 2 \cdot (k+1) \Rightarrow 2^{k+1} > 2 \cdot (k+1)$$

Therefore, it holds for $n = k+1$ and by the principle of mathematical induction it holds for all natural numbers greater than 2. □

Exercise 22 [Sum]

(a) Using the principle of mathematical induction, show that

$$\sum_{i=1}^{n} i = \frac{1}{2} n \cdot (n+1).$$

(b) Using the theorem above, write a more efficient version for computing the sum of the first n numbers.

(c) For which cases does your algorithm fail?

(d) When does integer overflow happen?

(e) (+) Is there a case where your function might overflow, even if the sum fits in an integer? How can you fix this? [*Hint:* Argue that n or $n+1$ must be even].

(a) We will prove this by induction on n.

Base case: For $n = 1$, $\sum_{i=1}^{1} i = 1 = n$.

Induction step: Assume that it is true for $n = k$, i.e., $\sum_{i=1}^{k} i = k \cdot (k+1)/2$. Then, for $n = k+1$,

$$\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1) = k \cdot (k+1)/2 + (k+1) = (k+1)(k/2+1)/2 = (k+1)(k+2)/2.$$

Hence, it is true for all n by the principle of mathematical induction.

(b)
```
let rec nsum n = n * (n+1) / 2;;
```

(c) The algorithm fails when $n < 0$ or when it is too large leading to overflow.

(d) Yes, there is. In a 64-bit system where the maximum integer representable is $2^{62} - 1$, for $n = 2^{31}$, the result is $2^{31} \cdot (2^{31}+1)/2 = (2^{62}+2^{31})/2 = 2^{61}+2^{30}$ which is $< 2^{62}-1$, so it is representable. However, the above implementation first multiplies $2^{31} \cdot (2^{31}+1) = 2^{62}+2^{31}$ which overflows to -4611686016279904256 and then the division by 2 just gives a negative number.

One way to fix this is to note that either n or $n+1$ is even, so we can divide the appropriate number by 2, before taking the product. Hence, the product will be equal to the result (and not larger).

```
let nsum n =
   if n mod 2 = 0 then (n/2) * (n+1)
   else ((n+1)/2) * n;;

nsum 2147483648;; (* 2305843010287435776 *)
```

You can double check the results on Wolfram.

Discussion:

- Be careful to use integers, as conversion to floats may lead to losing precision in a 64-bit int system.

Exercise 23 [Naive power]

(a) Using the principle of induction show that naive power computes x^n.

(b) In terms of n, how many multiplications does your function perform?

(c) For which cases does the function fail?

(d) Modify the function to compute the power of an integer.

(a) We will show by the principle of mathematical induction that npower $x\ n \Rightarrow x^n$ for every x and for every n.

Base case: For $n = 0$,

$$\begin{aligned} \text{npower } x\ 0 &\Rightarrow \textbf{if } 0 = 0 \textbf{ then } 1 \textbf{ else } x * \text{npower } (0-1) \\ &\Rightarrow 1 \end{aligned}$$

So, it is true for $n = 0$, that npower $x\ 0 \Rightarrow 1$.

Inductive step: Assume that it is true for $n = k$, i.e., that npower $x\ k \Rightarrow x^k$. Then, for $n = k+1$,

$$\begin{aligned} \text{npower } x\ n &\Rightarrow \textbf{if } \mathrm{k}+1 = 0 \textbf{ then } 1 \textbf{ else } \mathrm{x} * \text{npower } ((\mathrm{k}+1)-1) \\ &\Rightarrow x * \text{npower } ((k+1)-1) \\ &\Rightarrow x * \text{npower } k \\ &\Rightarrow x * x^k \Rightarrow x^{k+1}. \end{aligned}$$

Hence, it is also true for $n = k+1$ and so true for all $n \in \mathbb{N}$ by the principle of mathematical induction.

(b) To compute $\underbrace{x * x * \ldots * x}_{n \text{ terms}}$ we use $n-1$ multiplications.

(c) Fails when n is a negative integer (but we can fix this by calling `npower (1/. x) (-n)`). Fails in some sense when the provided values are too large causing overflows.

(d) Change the operations from `*.` to `*` (but be careful not to change `n`).

Exercise 24 [Efficient power (++)]

(a) Using the principle of induction (with the stronger hypothesis that the function returns correct values for all $n \leq k$ instead of just $n = k$), show that the `power` function in Section 1.6, correctly computes x^n.

(b) Approximately how many multiplications does this function perform in terms of n.

(a) As hinted in the statement we will prove that "for all n. for all $i \leq n$. for all x. power x $n = x^n$" using induction.

Base case: Consider $n = 0$, then we only need to check $i = 0$. Consider an arbitrary x, then

$$\begin{aligned} \text{power } x \; n &\Rightarrow \textbf{if } 0 = 0 \textbf{ then } 1 \textbf{ else if } \text{even } 0 \textbf{ then power } (x * x) \; (0/2) \textbf{ else } x * \text{power } (x * x) \; (0/2) \\ &\Rightarrow 1 \end{aligned}$$

Hence, power x $n \Rightarrow x^0 = 1$.

Inductive step: Assume true for $n = k$, i.e., "for all $i \leq k$. for all x. power x $n = x^n$". Now consider $n = k+1$ and notice that since $n/2 < n$ (as $n \geq 1$), we know from the induction hypothesis that for every y, power y $(n/2) = y^(n/2)$.

We we will now consider two subcases:

Case (i)**:** when n is even, i.e. $n = 2z$ for some $z \in \mathbb{N}$, so

$$\begin{aligned} \text{power } x \; (2z) &\Rightarrow \textbf{if } (2z) = 0 \textbf{ then } 1 \textbf{ else if } \text{even } (2z) \textbf{ then power } (x * x) \; ((2z)/2) \textbf{ else } x * \text{power } (x * x) \; ((2z)/2) \\ &\Rightarrow \textbf{if } \text{even } (2z) \textbf{ then power } (x * x) \; ((2z)/2) \textbf{ else } x * \text{power } (x * x) \; ((2z)/2) \\ &\Rightarrow \text{power } (x * x) \; ((2z)/2) \\ &\Rightarrow \text{power } x^2 \; z \\ &\Rightarrow (x^2)^z (\text{for } y = x^2) \\ &\Rightarrow x^n. \end{aligned}$$

Case (ii)**:** when n is odd, i.e. $n = 2z + 1$ for some $z \in \mathbb{N}$, so

$$\begin{aligned} \text{power } x \; (2z+1) &\Rightarrow \textbf{if } (2z+1) = 0 \textbf{ then } 1 \textbf{ else if } \text{even } (2z+1) \textbf{ then power } (x * x) \; ((2z+1)/2) \textbf{ else } x * \text{power } (x * x) \\ &\Rightarrow \textbf{if } \text{even } (2z+1) \textbf{ then power } (x * x) \; ((2z+1)/2) \textbf{ else } x * \text{power } (x * x) \; ((2z+1)/2) \\ &\Rightarrow x * \text{power } (x * x) \; ((2z+1)/2) \\ &\Rightarrow x * \text{power } x^2 \; z \\ &\Rightarrow x * (x^2)^z (\text{for } y = x^2) \\ &\Rightarrow x^{2z+1} \\ &\Rightarrow x^n. \end{aligned}$$

Hence, by the principle of natural induction we are done.

(b) It performs $\mathcal{O}(\log_2 n)$ multiplications. The recurrence relation for the number of multiplications is given (and solved) in Exercise **??**, with the difference that $T(0) = 0$ (since we don't make any multiplications when $n = 0$).

Discussion:

- There are other ways to argue without using the reductions. In particular, one can do these arguments in text, i.e., "when n is even, then the function makes a call to ...". However, this is a bit more systematic.

Exercise 25 [Is it a power]

(a) Write an OCaml function that takes a positive integer v and determines if v is a power of 2.

(b) Write an OCaml function that takes positive integers v and b and determines if v is a power of b.

(a)
```
let rec is_pow_of_2 x =
  if x = 1 then true
  else if x mod 2 = 0 then is_pow_of_2 (x/2)
  else false;;

(* Or more concisely. *)
let rec is_pow_of_2 x =
  if x mod 2 = 0 then is_pow_of_2 (x/2)
  else x = 1;;
```

(b)
```
let rec is_pow_of b x =
  if b = 1 then x = 1
  else if x mod b = 0 then is_pow_of b (x/b)
  else x = 1;;

# is_pow_of 4 64;;
- : bool = true
# is_pow_of 4 65;;
- : bool = false
# is_pow_of 1 3;;
- : bool = false
# is_pow_of 1 1;;
- : bool = true
```

Discussion:

- What if we allow any non-negative integer? Fails for 0.
- What if we allow all integers?
- What is the time complexity of this approach?
- Sometimes solutions fail for a base of $d = 1$.
- Is there a faster way to check binary powers in other programming languages?

```
int powerOfTwo(int n) {
   return n && (!(n & (n-1)));
}
```

Extended Note 4 Page 14 does not contain the full derivation. The following contains more details:

$$
\begin{aligned}
\text{nsum } 3 &\Rightarrow \textbf{if } 3 = 0 \textbf{ then } 0 \textbf{ else } 3 + \text{nsum } 2 \\
&\Rightarrow \textbf{if } \text{false} \textbf{ then } 0 \textbf{ else } 3 + \text{nsum } 2 \\
&\Rightarrow 3 + \text{nsum } 2 \\
&\Rightarrow 3 + \textbf{if } 2 = 0 \textbf{ then } 0 \textbf{ else } 2 + \text{nsum } 1 \\
&\Rightarrow 3 + \textbf{if } \text{false} \textbf{ then } 0 \textbf{ else } 2 + \text{nsum } 1 \\
&\Rightarrow 3 + (1 + \text{nsum } 1) \\
&\Rightarrow 3 + (2 + \textbf{if } 1 = 0 \textbf{ then } 0 \textbf{ else } 1 + \text{nsum } 0) \\
&\Rightarrow 3 + (2 + \textbf{if } \text{false} \textbf{ then } 0 \textbf{ else } 1 + \text{nsum } 0) \\
&\Rightarrow 3 + (2 + (1 + \text{nsum } 0)) \\
&\Rightarrow 3 + (2 + (1 + \textbf{if } 0 = 0 \textbf{ then } 0 \textbf{ else } 0 + \text{nsum } -1)) \\
&\Rightarrow 3 + (2 + (1 + \textbf{if } \text{true} \textbf{ then } 0 \textbf{ else } 0 + \text{nsum } -1)) \\
&\Rightarrow 3 + (2 + (1 + 0)) \\
&\Rightarrow 3 + (2 + 1) \\
&\Rightarrow 3 + 3 \\
&\Rightarrow 6
\end{aligned}
$$

A glimpse of formal semantics. We can define formally the operation of the if-statement as follows. Let $e_1 \Rightarrow v_1$, then if $v_1 = \text{true}$, then

$$\textbf{if } \text{e}_1 \textbf{ then } \text{e}_2 \textbf{ else } \text{e}_3 \Rightarrow e_2$$

Otherwise if $v_2 = \text{false}$,

$$\textbf{if } \text{e}_1 \textbf{ then } \text{e}_2 \textbf{ else } \text{e}_3 \Rightarrow e_3$$

Exercise 26 [Expression evaluation] Show some of the derivation steps for the expression `power(3, 12)`.

Extended Note 5 [Recursive vs Tail recursive]
To investigate further how a tail-recursive function is evaluated differently from a non-tail-recursive function, we will use the debugging capabilities of OCaml.
Step 1: Define the recursive and iterative versions to sum integers from 1 to n.

```
let rec nsum n =
  if n = 0 then 0
  else n + nsum (n-1);;

let rec nsumIter (n, ans) =
  if n = 0 then ans
  else nsumIter(n-1, (n + ans));;
```

Step 2: Enable tracing, which shows the input arguments for each function call and the its return value.

```
#trace nsum;;
#trace nsumIter;;
```

Step 3: Execute `nsum 5` and `nsumIter 5`. Note that all `nsumIter` function calls return the same value. For the OCaml implementer, this means that there is no reason to keep in memory the inputs (nor intermediate values) of the previous function calls, only of the last one. Then, the implementation should chain the returned value of the last function call to the return value of the first function call.

```
nsum 5
  nsum 4
   nsum 3
    nsum 2
     nsum 1
```

```
      nsum 0
      nsum () = 0
     nsum () = 1
    nsum () = 3
   nsum () = 6
  nsum () = 10
 nsum () = 15

nsumIter (5, 0)
 nsumIter (4, 5)
  nsumIter (3, 9)
   nsumIter (2, 12)
    nsumIter (1, 14)
     nsumIter (0, 15)
     nsumIter () = 15
    nsumIter () = 15
   nsumIter () = 15
  nsumIter () = 15
 nsumIter () = 15
nsumIter () = 15
```

(You will learn more about how the compiler achieves this in the Part IB Compiler Construction course)

Exercise 27 [Factorial function] The factorial of a positive integer n is given by $n! = n \cdot (n-1) \cdot \ldots \cdot 1$.

(a) Write a recursive function to compute the factorial.

(b) Write an iterative function to compute the factorial. Verify that it is iterative by inspecting the values returned by the functions.

(a)
```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1);;
```

(b)
```
let fact n =
    let rec facti n acc =
       if n = 0 then acc
       else facti (n-1) (n * acc)
    in
      facti n 1;;

(* For tracing to work, facti needs to be defined outside
   the function. *)
# #trace facti;;
facti is now traced.
# facti 3 1;;
facti <-- 3
facti --> <fun>
facti* <-- 1
facti <-- 2
facti --> <fun>
facti* <-- 3
facti <-- 1
facti --> <fun>
facti* <-- 6
facti <-- 0
facti --> <fun>
facti* <-- 6
facti* --> 6
facti* --> 6
facti* --> 6
facti* --> 6
- : int = 6
```

The function always returns the same answer, so there is no need to store the previous calls on the callstack (and so some compilers will keep only one call on the stack, saving memory).

Discussion:

- Did you read (and understand) the note before the question?
- What should we do if $n < 0$?

Exercise 28 [Iterative power functions]
(a) Write an iterative version of the power function.
(b) Write an iterative version of the efficient power function.

(a)
```
let rec npoweri x n acc =
    if n = 0 then acc
    else npoweri x (n - 1) (x *. acc);;

let npower x n = npoweri x n 1.0;;

npower 2.0 5;; (* 32 *)
npower 3.0 4;; (* 81 *)
```

(b)
```
let rec poweri x n acc =
    if n = 1 then acc *. x
    else if n mod 2 = 0 then poweri (x *. x) (n / 2) acc
    else poweri (x *. x) (n / 2) (x *. acc);;

let power x n = poweri x n 1.0;;

poweri 2.0 5;; (* 32 *)
poweri 3.0 4;; (* 81 *)
```

Exercise 29 [Tail-recursion]
(a) What is the benefit of tail-recursion?
(b) Why would someone not use tail-recursion?

(a) Tail-recursion can reduce the amount of memory used, by reducing the size of the call stack.

(b) It makes the code more complicated and sometimes the benefits are not big (e.g., could be asymptotically the same or the bottleneck might be in a different function).

Discussion:

- In which cases does it make sense to trade off code readability with efficiency? (In most cases it is hard to decide. In functins used by many applications such as the core library functions, these should probably be made as efficient as possible)

Extended Note 6 [Big-O notation] *You will learn more about Big-O notation in the Part IA algorithms course*
The formal definition for $\mathcal{O}(\cdot)$ notation is the following:
$f(n)$ is in $\mathcal{O}(g(n))$ iff there exist $c > 0$ and n_0 such that for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$.
Example 1 Let $f(n) = n$ and $g(n) = n^2$. Show that $f(n) = \mathcal{O}(g(n))$.

Proof. Consider $c = 1$, then we need to show that there is an n_0 such that for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$, or equivalently that $0 \leq n \leq n^2$. The first part of the inequality ($n \geq 0$ is true for all $n \geq 0$). Now we will investigate when is the second part true:

$$n \leq n^2 \Leftrightarrow 0 \leq n^2 - n \Leftrightarrow 0 \leq n \cdot (n-1).$$

For $n \geq 0$ and $n \geq 1$, the product $n \cdot (n-1)$ is non-negative, hence the inequality holds. Therefore, we must choose $c = 1$ and $n_0 = 1$. □

Proving that $f(n) = \mathcal{O}(g(n))$ is generally harder directly using the definition. Instead we rely on using the following properties:

1. $\mathcal{O}(1)$ is contained in $\mathcal{O}(n^a)$ for $a > 0$.
2. $\mathcal{O}(n^a)$ is contained in $\mathcal{O}(n^b)$ if $a \leq b$.
3. $\mathcal{O}(a^n)$ is contained in $\mathcal{O}(b^n)$ if $a \leq b$.
4. $\mathcal{O}(\log n)$ is contained in $\mathcal{O}(n^a)$ for all $a > 0$.
5. If $\mathcal{O}(g_1(n))$ is contained in $\mathcal{O}(f_1(n))$ and $\mathcal{O}(g_2(n))$ is contained in $\mathcal{O}(f_2(n))$, then $\mathcal{O}(g_1(n) + g_2(n))$ is contained in $\mathcal{O}(f_1(n) + f_2(n))$ (for positive functions g_1 and g_2).
6. $\mathcal{O}(\alpha \cdot f(n))$ is equivalent to $\mathcal{O}(f(n))$.

Example 2 Let $f(n) = 4 \cdot n^2 + 3n + 8$. Show that $f(n) = \mathcal{O}(n^2)$.

Proof. Split $f(n) = f_1(n) + f_2(n) + f_3(n)$, where $f_1(n) = 4 \cdot n^2$, $f_2(n) = 3n$ and $f_3(n) = 8$. Using Property 6, $f_1(n) = \mathcal{O}(n^2)$, $f_2(n) = \mathcal{O}(n)$ and $f_3(n) = \mathcal{O}(1)$. Using Property 4, $f_1(n) = \mathcal{O}(n^2)$, $f_2(n) = \mathcal{O}(n^2)$ and $f_3(n) = \mathcal{O}(n^2)$. Using Property 5, $f(n) = \mathcal{O}(n^2 + n^2 + n^2) = \mathcal{O}(3 \cdot n^2)$. Using Property 6, $f(n) = \mathcal{O}(n^2)$. □

Exercise 30 [Big-O notation]

(a) (Open-ended) Why is the big-O notation used? Can we not just do empirical computations of the running time?

(b) The following method can be used to measure the evaluation time of some expression.

```
let measure () =
    let t = Sys.time() in
        (<expression you want to time>; Printf.printf "%fs\n" (Sys.time()
            -. t));;
```

Measure the time it takes to execute `sillySum` (section 2.5) for values $n = 20$ to $n = 30$. Plot the graph vs input size and compare with the theoretical running time.

(c) What assumptions do we usually make when computing the runtime efficiency of an algorithm?

(d) What is the time complexity of the naive and efficient power function?

Exercise 31 [Big-O notation properties (++)]

(a) Show Big-O notation Property 2.

(b) Plot a graph on any plotting software (or Google) to convince yourself that $\log n$ is contained in $\mathcal{O}(0.1 \cdot \sqrt{n})$.

Exercise 32 [Big-Omega and Big-Theta notations]

(a) What is the formal definition for big-Omega notation?

(b) What is the formal definition for big-Theta notation?

(c) What does it mean for an algorithm to run in *polynomial time*?

(d) (Open-ended) A friend of yours says that all algorithms that run in polynomial time are extremely efficient. They also suggest that non-polynomial time algorithms are of no use. What would you respond?

Exercise 33 [(+)] The runtime $T(n)$ of an algorithm satisfies the following recurrence relation $T(1) = 1$ and for $n > 0$, $T(n + 1) = T(n) + 1$. Prove that the runtime is linear in big-O notation.

We start by evaluating $T(n)$ for some small values of n, i.e. $T(1) = 1$, $T(2) = T(1) + 1 = 1 + 1$, $T(3) = T(2) + 1 = 2 + 1 = 3$. A reasonable conjecture is that $T(n) = n$. Now we will try to prove this by induction.
Base case: For $n = 1$, the base case is true by the assumption $T(1) = 1$.
Inductive step: Assume true for $n = k$, i.e., $T(k) = k$. The, for $n = k + 1$, by the recursive formula,

$$T(k + 1) = T(k) + 1 = k + 1.$$

Hence, by the principle of natural induction it is true for all $n \in \mathbb{N}$.

Exercise 34 [(+)] The runtime $T(n)$ of an algorithm satisfies the following recurrence relation $T(1) = 1$ and for $n > 0$, $T(n + 1) = T(n) + n + 1$. Prove that the runtime is quadratic in big-O notation.

Again, we start by evaluating $T(n)$ for some small values of n, i.e. $T(1) = 1$, $T(2) = T(1) + 2 = 1 + 2$, $T(3) = T(2) + 1 = 1 + 2 + 3$. A reasonable conjecture is that $T(n) = 1 + 2 + \ldots + n$ (which we know to be $n \cdot (n + 1)/2$). Now we will try to prove this by induction.
Base case: For $n = 1$, the base case is true by the assumption $T(1) = 1 = 1 \cdot (1 + 1)/2$.
Inductive step: Assume true for $n = k$, i.e., $T(k) = k(k+1)/2$. Then, for $n = k+1$, by the recursive formula,

$$T(k + 1) = k \cdot (k + 1)/2 + (k + 1) = (k + 1) \cdot (k/2 + 1) = (k + 1)(k + 2)/2.$$

Hence, by the principle of natural induction it is true for all $n \in \mathbb{N}$.

Exercise 35 [(+)] The runtime $T(n)$ of an algorithm satisfies the following recurrence relation $T(1) = 1$ and for $n > 0$, $T(n) = T(n/2) + 1$. Prove that the runtime is logarithmic in big-O notation. Assume that $n = 2^k$.

Again, we start by evaluating $T(n)$ for some small values of k, i.e. $T(2^0) = 1$, $T(2^1) = T(2^0) + 1 = 2$, $T(2^2) = T(2^1) + 1 = 3$, $T(2^3) = T(2^2) + 1 = 4$. A reasonable conjecture is that $T(2^k) = k + 1$. Now we will try to prove this by induction.
Base case: For $k = 0$, the base case is true by the assumption $T(2^0) = T(1) = 1$.
Inductive step: Assume true for $n = k$, i.e., $T(2^k) = k + 1$. The, for $n = k + 1$, by the recursive formula,

$$T(2^{k+1}) = T(2^k) + 1 = (k + 1) + 1 = k + 2.$$

Hence, by the principle of natural induction it is true for all $k \in \mathbb{N}$, so for all n being a power of 2, we have $T(n) = T(2^k) = k + 1 = \log_2 n + 1$.

Exercise 36 [(++)] The runtime $T(n)$ of an algorithm satisfies the following recurrence relation $T(1) = 1$ and for $n > 0$, $T(n) = T(n/2) + n$. Prove that the runtime is linear in big-O notation. Assume that $n = 2^k$.

For $T(2^0) = 1$, $T(2^1) = T(1) + 2 = 1 + 2$, $T(2^2) = T(2) + 4 = 1 + 2 + 4$. So our conjecture is that $T(2^k) = 1 + 2 + \ldots 2^k = 2^{k+1} - 1$.
Base case: For $k = 0$, this is true by the assumption $T(2^0) = T(1) = 1 = 2^{0+1} - 1$.
Inductive step: Assume true for $n = k$, i.e., $T(2^k) = 2^{k+1} - 1$. The, for $n = k + 1$, by the recursive formula,

$$T(2^{k+1}) = T(2^k) + 2^{k+1} = (2^{k+1} - 1) + 2^{k+1} = 2 \cdot 2^{k+1} - 1 = 2^{k+2} - 1.$$

Hence, by the principle of natural induction it is true for all $k \in \mathbb{N}$, so for all n being a power of 2, we have that $T(n) = T(2^k) = 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2 \cdot n - 1$.

Exercise 37 [(++)] 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 $\mathcal{O}(n \log n)$.

[Exercise 2.4 in Lecturer's handout]

Again, we start by evaluating $T(n)$ for some small values of k, i.e. $T(2^0) = 1$, $T(2^1) = 2 \cdot T(2^0) + 1 = 2 + 1$, $T(2^2) = 2 \cdot T(2^1) + 1 = 2 \cdot (2 + 1) + 1 = 4 + 2 + 1$, $T(2^3) = 2 \cdot T(2^2) + 1 = 2 \cdot (4 + 2 + 1) + 1 = 8 + 4 + 2 + 1$. A reasonable conjecture is that $T(2^k) = 1 + 2 + \ldots + 2^k = 2^{k+1} - 1$. Now we will try to prove this by induction.
Base case: For $k = 0$, the base case is true by the assumption $T(2^0) = T(1) = 1 = 2^{0+1} - 1$.

Inductive step: Assume true for $n = k$, i.e., $T(2^k) = 2^{k+1} - 1$. The, for $n = k + 1$, by the recursive formula,

$$T(2^{k+1}) = 2 \cdot T(2^k) + 1 = 2 \cdot (2^{k+1} - 1) + 1 = 2^{k+2} - 1.$$

Hence, by the principle of natural induction it is true for all $k \in \mathbb{N}$, so for all n being a power of 2, we have that $T(n) = T(2^k) = 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2 \cdot n - 1$.

Exercise 38 [Matrix exponentiation (+++)] (Only attempt this if you know about matrices). Consider two 2×2 matrices A and B.

(a) Implement an OCaml function that takes the elements of A and B and returns the matrix product of these two.

$$A \cdot B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

(b) Modify the `power` function to compute the power of a matrix.

(c) What is the time complexity of your algorithm?

(d) Implement an OCaml function that takes a 2×2 matrix A and a 2-element vector v and computes $A \cdot v$.

(e) The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$ (for $n > 1$) with $F_0 = 0$ and $F_1 = 1$. Show that for $n > 0$

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

(f) Using the functions you developed above, show how to compute the n-th Fibonacci number in $\mathcal{O}(\log n)$ time.

(a) One possible way to represent a matrix is using a pair of pairs. Here, we could have also used a single 4-tuple or a list.

```
let mat_mult ((a11,a12),(a21,a22)) ((b11,b12),(b21,b22)) =
   ((a11 * b11 + a12 * b21, a11 * b12 + a12 * b22),
    (a21 * b11 + a22 * b21, a21 * b12 + a22 * b22));;
```

(b)
```
let rec fast_mat_pow m n =
  if n = 1 then m
  else if n mod 2 = 1 then mat_mult m (fast_mat_pow (mat_mult m m) (n / 2))
  else fast_mat_pow (mat_mult m m) (n / 2);;
```

(Optional) It is relatively easy to see why the naive power function works. For the fast exponentiation method, we need to prove that it works for matrix multiplication. Note that implicitly we are using the identities

$$m^{2k} = \underbrace{m \cdot (m \cdot (m \cdot (\ldots \cdot m \ldots)))}_{2k \text{ terms}} = \underbrace{(m \cdot m) \cdot ((m \cdot m) \cdot \ldots \cdot (m \cdot m) \ldots)}_{k \text{ terms}} = \underbrace{(m \cdot m) \cdot (m^2 \cdot \ldots \cdot m^2 \ldots)}_{k \text{ terms}} = (m^2)^k,$$

and $m^{2k+1} = (m^2)^k \cdot m$, which follows directly from the first one. The property that allows changing the order of the parentheses is *associativity*, i.e. for any 2×2 matrices A, B and C, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$. Using this property, we can gradually transform the LHS to the RHS. In particular start with

$$m \cdot (m \cdot (\ldots)) = (m \cdot m) \cdot (\ldots)$$

and then recursively perform the same transformation for $\ldots$. (If you are interested, you can try and show that no matter how you place parentheses for an associative operator the result will be the same.)

(c) We are doing a constant number of operations (8 multiplications and 4 additions), so the time complexity for matrix multiplication of 2×2 matrices is $\mathcal{O}(1)$.

(d) Using the pair of pairs representation, we have

```
let rec fast_mat_pow m n =
   if n = 1 then m
   else if n mod 2 = 1 then mat_mult m (fast_mat_pow (mat_mult m m) (n / 2))
   else fast_mat_pow (mat_mult m m) (n / 2);;
```

(e) Finally we can find the n-th Fibonacci number as follows:

```
let fib n =
   if n = 0 then 0
   else let _, ans = mat_vec_mult (fast_mat_pow ((1,1), (1, 0)) n) (1, 0) in
      ans;;
```

4 Lecture 3

Exercise 39 [Tuples]

(a) What is the syntax for creating tuples?

(b) What is the syntax for accessing the elements of a tuple?

(c) How are the types of tuples represented? What are the types of the expressions you mentioned above?

(d) (+) Would it be possible to have a function that takes a tuple and an integer n and returns the n-th component of the tuple?

(a) The syntax for creating tuples is `(e1, e2, ... , en)` where $e_1, \ldots, e_n$ are expressions.

(b) One can access the i-th element of a tuple by `let (x1, .... , xi, ... , xn) = tuple in e` or if you are going to use only the i-th element use `let (_, ... , xi, ... , _) = tuple in e`.

(c) The type of a tuple has the form $T_1 * T_2 * \ldots * T_n$. The creation `(e1, e2, ... , en)` has type $(T_1, \ldots, T_n)$ where T_i is the type of e_i. Furthermore, if we write `let (x1, ... , xi, ... , xn) = e` and $e : T_1 * \ldots * T_n$ then $e_i : T_i$.

(d) No, it is not possible because this function would not type check. Actually, it is not possible to do it even for the pair function. Assume we had such a function `let f (x, y) n = ...`. So the input to `f` would be of type `'a * 'b` and the return value would be of type `'a` or `'b` depending on the value of n, so there cannot be a type for the return value (unless `'a` is equal to `'b`).

Exercise 40 [Tuples - types] State the types for the following tuples.

(a) `(1, 2, 3)`

(b) `(("A", 2), 3.0)`

(c) `(2.0 +. 3.0)`

(d) `(nsum, 2)`

(e) `(1, "AA", 2.0)`

(f) `([(1, 2.0); (3, 4.0)], "AA")`

(g) `([(1, 2); (3.0, 4)], "AA")`

(a) `int * int * int`

(b) `(string * int) * float`

(c) `float`

(d) `(int -> int) * int`

(e) `int * string * float`

(f) `(int * float) list * string`

(g) This expression does not compile because the list elements do not have the same type.

Exercise 41 [Polymorphism]

(a) What is *polymorphism*? What are type variables? Explain the term by giving references to `let identify x = x` and `let swap (x, y) = (y, x)`.

(b) Why is polymorphism useful?

(a) Polymorphism is the feature of programming languages that allows one function to be used for inputs of various types. Type variables `'a`, `'b`,... are variables used to indicate that any type can be used in their place. For example, `let identify x = x` is a function with type `'a -> 'a`, meaning that it accepts any type and it returns a value of that type.

The type of `let swap (x, y) = (y, x)` is `'a->'b->('b * 'a)`, meaning that it takes two arguments of any types T_1 and T_2 and will return a pair where the first element will have type T_2 and the second T_1.

(b) Polymorphism is useful because it allows us to write a function once and handle many types (instead of writing one function for each type).

Exercise 42 [Lists]

(a) What is the syntax for creating a list and for appending an item to a list?

(b) What is the syntax for accessing an element from an list?

(c) What is the type of a list? Can a list have items of different types?

(d) What is the type of the empty list? Why?

(e) How are lists represented internally?

(a) We can create a list using `[a1; a2; a3;...]` or using cons `a1::a2::a3:: ... :: []`.

(b) We can access the head of a list and its tail using `let x::xs = ls`.

(c) The type of a list is `'a list`. All elements of a list must be of the same type.

(d) The type of the empty list is `'a list`, so that it can be used with any type of elements (otherwise we would need a special empty list for each type).

(e)

Exercise 43 [Lists - types] What are the results of the following expressions and what are their types?

(a) `[]`

(b) `[1; 2; 3]`

(c) `[[1; 2], [1]]`

(d) `[1, [2; 3]]`

(e) `[1; 2] = [1; 2]`

(f) `[1] = [2]`

(g) `[] = []`

(a) `'a list = []`

(b) `int list = [1; 2; 3]`

(c) `(int list * int list) list = [([1; 2], [1])]` (because `[ , ]` creates a tuple)

(d) `(int * int list) list = [(1, [2; 3])]`

(e) `bool = true`

(f) `bool = false`

(g) `bool = true`

Exercise 44 [Function pattern matching]

(a) What is the syntax for defining a function with pattern matching?

(b) Define the function nsum using pattern matching.

(c) What is the problem with the following function declaration?

```
let rec nsum = function
    n -> n + nsum (n-1)
  | 0 -> 0;;
```

(d) When can you get a "This pattern-matching is not exhaustive." warning? What happens if the warning is realised?

(a) Here are two ways of doing it,

```
let f = function
  p1 -> ...
| p2 -> ...
| p3 -> ...;;

let f x = match x with
  p1 -> ...
| p2 -> ...
| p3 -> ...;;
```

The first method is a bit more useful when we want to pattern match on the last argument.

(b)
```
let rec nsum = function
  0 -> 0
| n -> n + nsum (n-1);;
```

(c) The problem is that the first pattern matches all values of n including $n = 0$. Hence, the code will never enter the second case. Hence, the function will not terminate (or will cause stack overflow).

(d) This warning means that there exist values of the type for which there is no pattern that matches it. For example, the following function converts integers to booleans. Any integer values other than 0 or 1 do not have a pattern that will match them.

```
let int_to_bool = function
  0 -> false
| 1 -> true;;
```

When we run e.g., `int_to_bool 2` we get `Exception: Match_failure (...).`.

Exercise 45 [Head Tail]

(a) Implement the `head` and `tail` functions.

(b) What are their types?

(a)
```
let head (x::_) = x;;

let head = function
  [] -> raise (Invalid_argument "The given list needs to be non-empty.")
| x::_ -> x;;

let tail (_::xs) = xs;;

let tail = function
  [] -> raise (Invalid_argument "The given list needs to be non-empty.")
| _::xs -> xs;;
```

(b) `'a list -> 'a` (meaning that it takes a list with elements of any type T and returns an element of type T)

`'a list -> 'a list` (meaning that it takes a list with elements of any type T and also returns a list with elements of type T)

Exercise 46 [Null]

(a) What does the `null` function do?

(b) What is its type?

(a) The `null` function checks if the list is empty.

(b) Its type is `'a list -> bool`, taking a list of any type and returning a boolean indicating if it is empty.

Exercise 47 [Append]

(a) What does `append` do? What is its type?

(b) Let ℓ_1 and ℓ_2 be the lengths of the two lists. What is the efficiency of append in big-O notation?

(c) Write an efficient OCaml function for appending one list to another.

(a) Append takes two lists L_1 and L_2 and returns the concatenation of the two, i.e. a list containing the elements of L_1 followed by the elements of L_2.

(b) Notice that the append operation terminates when it finds the end of list L_1. Hence, it only requires time proportional to the size of the list, which is ℓ_1.

Exercise 48 [Reverse]

(a) What does reverse do?

(b) What is the time complexity of `nrev` in section 3.6?

(c) What is the time complexity of `rev` in section 3.7?

(a) Given a list (of any type) it returns a list containing the elements of the initial list but in reverse order, i.e., given $[x_1; x_2; \ldots; x_n]$ it returns $[x_n; x_{n-1}; \ldots; x_2; x_1]$.

(b) As explained in the lecture notes `nrev` takes quadratic time to the length of the list (i.e. $\Theta(n^2)$, where n is the number of elements in the list).

(c) As explained in the lecture notes `rev` takes linear time to the length of the list (i.e. $\mathcal{O}(n)$, where n is the number of elements in the list).

Discussion:

- *Technically, why is not enough to say in (b) that `nrev` takes $\mathcal{O}(n^2)$ time?* Because `rev` in (c) also takes $\mathcal{O}(n^2)$ since $\mathcal{O}(n)$ is contained in $\mathcal{O}(n^2)$.

Exercise 49 [List length]

(a) Write an OCaml recursive function to compute the length of a list.

(b) Write an OCaml iterative function to compute the length of a list.

(c) Are your functions polymorphic? Would there be any disadvantage if these were not polymorphic?

(a)
```
let rec len = function
  [] -> 0
| _::xs -> 1 + len xs;;

len [];; (* 0 *)
len [1;2;3];; (* 3 *)
```

(b)
```
let rec lenit acc = function
  [] -> acc
| _::xs -> lenit (acc + 1) xs;;

let len ls = lenit 0 ls;;

len [];; (* 0 *)
len [1;2;3];; (* 3 *)
```

(c) Yes, they are since their type is `'a list -> int`. We would need to have a different length function for each possible type of lists.

Exercise 50 [Last element]

(a) Write an OCaml iterative function to return the last element of a list.

(b) What happens if we call the function with the empty list?

(c) What is the time complexity of your function? Can we do better?

[Exercise 3.2 in Lecturer's handout]

(a)
```
let rec get_last = function
    [x] -> x
  | _::xs -> get_last xs;;

  get_last [1; 2; 3];; (* 3 *)
  get_last [1];; (* 1 *)
  get_last [];; (* Exception: match failure *)

  let rec get_last = function
    [] -> raise (Invalid_argument "The given list needs to be non-empty.")
  | [x] -> x
  | _::xs -> get_last xs;;
```

(b) In the first version there is a match failure, while in the second case an "invalid argument" exception is raised.

We should avoid returning `0` or `-1` or any other value, as this would limit the type of the function (from polymorphic to one that works e.g., for integer lists).

As we will see in the coming lectures, we can also use "options".

(c) The implementation requires to iterate over all elements so it is $\mathcal{O}(n)$, where n is the length of the list. We cannot do anything more efficient with the list representation of OCaml.

Exercise 51 [Odd elements] Write an OCaml function that returns the elements at odd positions in the list. Indexing starts at 0. For example, given $[a; b; c; d]$ it should return $[b; d]$.

[Exercise 3.3 in Lecturer's handout]

```
(* Recursive implementation *)
let rec get_odd = function
|[] -> []
|_ :: [] -> []
|_ :: y :: t ->y::get_odd t;;

(* Iterative implementation *)
let rec get_odd_rec acc = function
|[] -> acc
|_ :: [] -> acc
|_ :: y :: t ->get_odd_rec (y::acc) t;;
let get_odd l = rev (get_odd_rec [] l);;

# get_odd [1;2;3];;
- : int list = [2]
# get_odd [1;2;3;4;5;6];;
- : int list = [2; 4; 6]
# get_odd [1;2;3];;
- : int list = [2]
# get_odd [];;
- : 'a list = []
# get_odd [1];;
- : int list = []
```

```
(* Mutually recursive implementation *)
let rec get_odd = function
|[] -> []
|_::xs -> get_even xs
and get_even = function
|[] -> []
|y::ys -> y::get_odd ys;;
```

Exercise 52 [Strange functions (++)]
(a) What is the type of `let rec loop x = loop x`? Why?

(b) Why does `let rec trunc x = trunc` produce an error?

[Exercise 3.4 in Lecturer's handout]

(a) The `loop` function has type `'a -> 'b`. The compiler starts the type inference of the expression for the function body, assuming that `x` has the most general type `'a` and that `loop` has type `'a->'b`. These types are not constrained further in the expression of the function body, so it remains the same.

(b) Again we start with the most general type of `trunc` which is `'a -> 'b`. From the expression of the function body we know that the return value has type `'a -> 'b` (since it returns `trunc`). Hence, `'b` must be the same as `'a->'b`, This essentially means that `trunc` has type `'a->('a->'b)`. But then repeating the same argument we have that the type of trunc must be `'a->('a->('a->'b))`. Repeating again, `trunc` would need to have an infinite type `'a->'a->'a->...` which is not allowed in OCaml. This is like trying to solve a "type equation" for `'a` and `'b` satisfying `'a = 'a -> 'b`.

This relates to the concept of unification that you will revisit in the Part IB Prolog course.

Exercise 53 [All tails] Implement an OCaml function `tails` to return the list of the tails of its argument. For example, given `[1; 2; 3]` it should return `[[1; 2; 3]; [2; 3]; [3]; []]` (or perhaps in a different order).

[Exercise 3.5 in Lecturer's handout]

```
let rec all_tails = function
  ([], ans) -> []::ans
| (x::xs, ans) -> all_tails(xs, (x::xs)::ans);;
```

Discussion:

- *What is the time complexity of your solution?* It is $\mathcal{O}(n)$, because a copy is not made for each entry (the tails share elements).

Exercise 54 [All pairs] Implement an OCaml function `all_pairs` that takes a list and returns all possible ordered pairs in any order. For example, given `[1; 2; 3]` it could return `[(1, 2); (1, 3); (2, 3)]`.

We break the problem in two subproblems. First generate all pairs for a given element `x` and a list `ys`, and do this for every element of the list.

```
(* Solution using append *)
let rec single_pairs = function
  (x, []) -> []
| (x, y::ys) -> (x, y)::single_pairs(x, ys);;
let rec all_pairs = function
  [] -> []
| (x::ys) -> single_pairs(x, ys) @ all_pairs(ys);;
```

The time complexity of the above algorithm is $\mathcal{O}\left(n^2\right)$, but there is a bit of duplicate work with the append `@`. By using an accumulator instead of the append we can remove this duplicate work, but the time complexity will still be $\mathcal{O}\left(n^2\right)$.

```
(* Iterative version *)
let rec single_pairs_iter = function
  (x, [], ans) -> ans
| (x, y::ys, ans) -> single_pairs_iter(x, ys, (x, y)::ans);;
let rec all_pairs = function
  ([], ans) -> ans
| (x::ys, ans) -> all_pairs(ys, single_pairs_iter(x, ys, ans));;
```

Discussion:

- Can there be a faster algorithm? The number of pairs in the output is $n \cdot (n-1)/2$ (why?), hence we cannot aim for an algorithm with better complexity. However, we can generate a lazy list with all possible pairs, as we will see in future supervisions.