Foundations of Computer Science
Example Sheet 1

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.

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

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

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

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.

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]

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 $int \to int$, 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 $int \to bool$, 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]]

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?

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

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?

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?

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.)

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);;`?

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?

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`

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]

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]

Exercise 18 [Floating point] Explain the following outputs:

(a) `Int64.of_float(9007199254740993.0)` gives `9007199254740992L`.

(b) `3.0 +.  3E-90` gives `3.0`

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`

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`

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;;
```

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].

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.

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.

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.

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.

Exercise 28 [Iterative power functions]

(a) Write an iterative version of the power function.

(b) Write an iterative version of the efficient power function.

Exercise 29 [Tail-recursion]

(a) What is the benefit of tail-recursion?

(b) Why would someone not use tail-recursion?

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.

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.

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 $n = 2^k$.

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 $n = 2^k$.

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]

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.

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?

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")`

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?

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?

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) `[] = []`

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?

Exercise 45 [Head Tail]

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

(b) What are their types?

Exercise 46 [Null]

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

(b) What is its type?

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.

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?

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?

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]

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]

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]

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]

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)]`.