Introduction to Functional Programming and the Structure of Programming Languages using OCaml

Gert Smolka
Saarland University

with additions and modifications by
Lena Becker, Sebastian Hack,
Holger Hermanns, Gregory Stock

Version of November 30, 2023

Copyright © 2021–2023 by Gert Smolka et al., all rights reserved

Contents

Preface iv

1 Getting Started 1
1.1 Programs and Declarations 1
1.2 Functions and Let Expressions 3
1.3 Conditionals, Comparisons, and Booleans 4
1.4 Recursive Power Function 5
1.5 Integer Division 6
1.6 Mathematical Level versus Coding Level 9
1.7 More about Mathematical Functions 11
1.8 A Higher-Order Function for Linear Search 13
1.9 Partial Applications 15
1.10 Inversion of Strictly Increasing Functions 17
1.11 Tail Recursion 18
1.12 Tuples 20
1.13 Exceptions and Spurious Arguments 21
1.14 Polymorphic Functions 23
1.15 Iteration 24
1.16 Iteration on Pairs 25
1.17 Computing Primes 27
1.18 Polymorphic Exception Raising and Equality Testing 29
1.19 Summary 30

2 Lists 31
2.1 Nil and Cons 31
2.2 Basic List Functions 32
2.3 List Functions in OCaml 35
2.4 Fine Points About Lists 37
2.5 Membership and List Quantification 38
2.6 Head and Tail 39
2.7 Position Lookup 40
2.8 Option Types 41
2.9 Generalized Match Expressions 42
2.10 Sublists 44
2.11 Folding Lists 45
2.12 Insertion Sort 47

2.13 Generalized Insertion Sort 49
2.14 Lexicographic Order 50
2.15 Prime Factorization 51
2.16 Key-Value Maps 53

3 Constructor Types and Trees **56**
3.1 Constructor Types 56
3.2 Rose Trees 60
3.2.1 Successor Trees 62
3.2.2 Shape of Arithmetic Expressions 63
3.2.3 Lexicographic Tree Ordering 63
3.3 Subtrees 64
3.4 Addresses 65
3.4.1 Successors and Predecessors 67
3.5 Size and Depth 68
3.6 Folding 69
3.7 Pre-Ordering and Post-Ordering 70
3.7.1 Subtree Access with Pre-Numbers 71
3.7.2 Subtree Access with Post-Numbers 72
3.7.3 Linearisations 73
3.8 Balancedness 74
3.9 Finitary Sets and Directed Trees 75
3.10 Labelled Trees 77
3.11 Projections 80

Preface

This text teaches functional programming and the structure of programming languages to beginning students. It is written for the Programming 1 course for computer science students at Saarland University. We assume that incoming students are familiar with mathematical thinking, but we do not assume programming experience. The course is designed to take about one third of the study time of the first semester.

We have been teaching a course like this at Saarland University since 1998. Students perceive the course as challenging and exciting, whether they have programmed before or not. In 2021, we changed the teaching language to English and the programming language to OCaml.

As it comes to functional programming, we cover higher-order recursive functions, polymorphic typing, and constructor types for lists, trees, and abstract syntax. We emphasize the role of correctness statements and practice inductive correctness proofs. We also cover asymptotic running time considering binary search (logarithmic), insertion sort (quadratic), merge sort (linearithmic), and other algorithms.

As it comes to the structure of programming languages, we study the different layers of syntax and semantics at the example of the idealized functional programming language Mini-OCaml. We describe the syntactic layers with grammars and the semantic layers with inference rules. Based on these formal descriptions, we program recursive descent parsers, type checkers and evaluators.

We also cover stateful programming with arrays and cells (assignable variables). We explain how lists and trees can be stored as linked blocks in an array, thus explaining memory consumption for constructor types.

There is a textbook[1] written for the German iterations of the course (1998 to 2020). The new English text realizes some substantial changes: OCaml rather than Standard ML as programming language, less details about the concrete programming language being used, more emphasis on correctness arguments and algorithms, informal type-theoretic explanations rather than formal set-theoretic definitions.

The current version of the text leaves room for improvement. More basic explanations with more examples could be helpful in many places. An additional chapter on imperative programming with loops and the

[1] Gert Smolka, *Programmierung — eine Einführung in die Informatik mit Standard ML*. Oldenbourg Verlag, 2008.

realization with stack machines with jumps (see the German textbook) would be interesting, but this extra material may not fit into the time-budget of a one-semester course.

At Saarland University, the course spans 15 weeks of teaching in the winter semester. Each week comes with two lectures (90 minutes each), an exercises assignment, office hours, tutorials, and a test (15 minutes). There is a midterm exam in December and a final exam (offered twice) after the lecture period has finished. The 2021 iteration of the course also came with a take home project over the holiday break asking the students to write an interpreter for Mini-OCaml. The take home project should be considered an important part of the course, given that it requires the students writing and debugging a larger program (about 250 lines), which is quite different from the small programs (up to 10 lines) the weekly assignments ask for.

1 Getting Started

In this chapter we start programming in OCaml. We use an interactive tool called an interpreter that checks and executes the declarations of a program one by one. We concentrate on recursive functions on integers computing things like powers, integer quotients, digit sums, and integer roots. We also formulate a general algorithm known as linear search as a higher-order function. We follow an approach known as functional programming where functions are designed at a mathematical level using types and equations before they are coded in a concrete programming language.

1.1 Programs and Declarations

An OCaml **program** is a sequence of **declarations**, which are executed in the order they are written. Our first program

```
let a = 2 * 3 + 2
let b = 5 * a
```

consists of two declarations. The first declaration **binds** the **identifier** a to the integer 8, and the second declaration binds the identifier b to the integer 40. This is clear from our understanding of the arithmetic operations "+" and "$*$".

To learn programming in OCaml, you want to use an interactive tool called an **interpreter**.[1] The user feeds the interpreter with text. The interpreter checks that the given text can be interpreted as a program formulated correctly according to the rules of the programming language. If this is the case, the interpreter determines a **type** for every identifier and every expression of the program. Our example program is formulated correctly and the declared identifiers a and b both receive the type **int** (for integer). After a program has been checked successfully, the interpreter will execute it. In our case, execution will bind the identifier a to the integer 8 and the identifier b to the integer 40. After the program has been executed successfully, the interpreter will show the values it has computed for the declared identifiers. If a program is not formulated correctly, the interpreter will show an error message indicating which rule of the language is violated. Once the interpreter

[1] A nice browser-based interpreter is `https://try.ocamlpro.com`.

has checked and executed a program, the user can extend it with further declarations. This way one can write the declarations of a program one by one.

At this point you want to start working with the interpreter. You will learn the exact rules of the language through experiments with the interpreter guided by the examples and explanations given in this text.

Here is a program redeclaring the identifier a:

```
let a = 2 * 3 + 2
let b = 5 * a
let a = 5
let c = a + b
```

The second declaration of the identifier a **shadows** the first declaration of a. Shadowing does not affect the binding of b since it is obtained before the second declaration of a is executed. After execution of the program, the identifiers a, b, and c are bound to the integers 5, 40, and 45, respectively.

The declarations we consider in this chapter all start with the **keyword** *let* and consist of a **head** and a **body** separated by the equality symbol "=". Keywords cannot be used as identifiers. The bodies of declarations are **expressions**. Expressions can be obtained with identifiers, constants, and operators. The **nesting of expressions** can be arranged with **parentheses**. For instance, the expression $2 \cdot 3 + 2 - x$ may be written with **redundant parentheses** as $((2 \cdot 3) + 2) - x$. The parentheses in $2 \cdot (x + y) - 3$ are not redundant and are needed to make the expression $x + y$ the right argument of the product with the left argument 2.

Every expression has a **type**. So far we have only seen expressions of the type *int*. The values of the type *int* are integers (whole numbers $\dots, -2, -1, 0, 1, -2, \dots$). In contrast to the infinite mathematical type $\mathbb{Z}$, OCaml's type *int* provides only a finite interval of **machine integers** realized efficiently by the hardware of the underlying computer. The endpoints of the interval can be obtained with the predefined identifiers *min_int* and *max_int*. All machine operations respect this interval. We have $max\_int + 1 = min\_int$, for instance.[2]

When we reason about programs, we will usually ignore the machine integers and just assume that all integers are available. As long as the

[2]It turns out that different interpreters realize different intervals for machine integers, even on the same computer. For instance, on the author's computer, in October 2021, the browser-based Try OCaml interpreter realizes *max_int* as $2^{31} - 1 = 2147483647$, while the official OCaml interpreter realizes *max_int* as $2^{61} - 1 = 4611686018427387903$.

numbers in a concrete computation are small enough, this simplification does not lead to wrong conclusions.

Every programming language provides machine integers for efficiency. There are techniques for realizing much larger intervals of integers based on machine integers in programming languages.

Exercise 1.1.1 Give an expression and a declaration. Explain the structure of a declaration. Explain nesting of expressions. Give a type.

Exercise 1.1.2 To what value does the program

```
let a = 2 let a = a * a let a = a * a
```

bind the identifier a?

Exercise 1.1.3 Give a machine integer x such that $x + 1 < x$.

1.2 Functions and Let Expressions

Things become interesting once we declare functions. The declaration

```
let square x = x * x
```

declares a function

$$square : \; int \to int$$

squaring its argument. The identifier *square* receives a **functional type** $int \to int$ describing functions that given an integer return an integer. Given the declaration of *square*, execution of the declaration

```
let a = square 5
```

binds the identifier a to the integer 25.

Can we compute a power x^8 with just three multiplications? Easy, we just square the integer x three times:

```
let pow8 x = square (square (square x))
```

This declaration gives us a function $pow8 : int \to int$.

Another possibility is the declaration

```
let pow8' x =
  let a = x * x in
  let b = a * a in
  b * b
```

declaring a function $pow8' : int \to int$ doing the three multiplications using two **local declarations**. Local declarations are obtained with **let expressions**

$$let\ d\ in\ e$$

combining a declaration d with an expression e using the keywords *let* and *in*. Note that the body of pow' nests two let expressions as $let\ d_1\ in\ (let\ d_2\ in\ e)$. OCaml lets you write redundant parentheses marking the nesting (if you want to). Let expressions must not be confused with top-level declarations (which don't use the keyword *in*).

Exercise 1.2.1 Write a function computing x^5 with just 3 multiplications. Write both a version with *square* and a version with local declarations. Write the version with local declarations with and without redundant parentheses marking the nesting.

1.3 Conditionals, Comparisons, and Booleans

The declaration

```
let abs x = if x < 0 then -x else x
```

declares a function $abs : int \to int$ returning the absolute value of an integer (e.g., $abs(-5) = 5$). The declaration of *abs* uses a **comparison** $x < 0$ and a **conditional** formed with the keywords `if`, `then`, and `else`.

The declaration

```
let max x y : int = if x <= y then y else x
```

declares a function $max : int \to int \to int$ computing the maximum of two numbers. This is the first function we see taking two arguments. There is an explicit **return type specification** in the declaration (appearing as "$: int$"), which is needed so that *max* receives the correct type.[3]

Given *max*, we can declare a function

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

returning the maximum of three numbers. This time no return type specification is needed since *max* forces the correct type $max3 : int \to int \to int \to int$.

Next we declare a three-argument maximum function using a local declaration:

[3] The complication stems from the design that in OCaml comparisons like $\leq$ also apply to types other than *int*.

```
let max3 x y z : int =
  let a = if x <= y then y else x in
  if a <= z then z else a
```

There is a type *bool* having two values *true* and *false* called **booleans**. The comparison operator "$\leq$" used for *max* (written "`<=`" in OCaml) is used with the functional type $int \to int \to bool$ saying that an expression $e_1 \leq e_2$ where both subexpressions e_1 and e_2 have type *int* has type *bool*.

The declaration

```
let test (x : int) y z = if x <= y then y <= z else false
```

declares a function $test : int \to int \to int \to bool$ testing whether its three arguments appear in order. The **type specification** given for the first argument x is needed so that *test* receives the correct type. Alternatively, type specifications could be given for one or all of the other arguments.

Exercise 1.3.1 Declare minimum functions analogous to the maximum functions declared above. For each declared function give its type (before you check it with the interpreter). Also, write the declarations with and without redundant parentheses to understand the nesting.

Exercise 1.3.2 Declare functions $int \to int \to bool$ providing the comparisons $x = y$, $x \neq y$, $x < y$, $x \leq y$, $x > y$, and $x \geq y$. Do this by just using conditionals and comparisons $x \leq y$. Then realize $x \leq y$ with $x \leq 0$ and subtraction.

1.4 Recursive Power Function

Our next goal is a function computing powers x^n using multiplication. We recall that powers satisfy the equations

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x \cdot x^n \end{aligned}$$

Using the equations, we can compute every power x^n where $n \geq 0$. For instance,

$$\begin{aligned} 2^3 &= 2 \cdot 2^2 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2^1 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2 \cdot 2^0 && \text{2nd equation} \\ &= 2 \cdot 2 \cdot 2 \cdot 1 && \text{1st equation} \\ &= 8 \end{aligned}$$

The trick is that the 2nd equation reduces larger powers to smaller powers, so that after repeated application of the 2nd equation the power x^0 appears, which can be computed with the 1st equation. What we see here is a typical example of a recursive computation.

Recursive computations can be captured with **recursive functions**. To arrive at a function computing powers, we merge the two equations for powers into a single equation using a conditional:

$$x^n = \text{IF } n < 1 \text{ THEN } 1 \text{ ELSE } x \cdot x^{n-1}$$

We now declare a recursive function implementing this equation:

```
let rec pow x n =
  if n < 1 then 1
  else x * pow x (n - 1)
```

The identifier *pow* receives the type $pow : int \to int \to int$. We say that the function *pow* applies itself. Recursive function applications are admitted if the declaration of the function uses the keyword `rec`.

We have demonstraed an important point about programming at the example of the power function: Recursive functions are designed at a mathematical level using equations. Once we have the right equations, we can implement the function in any programming language.

1.5 Integer Division

Given two integers $x \geq 0$ and $y > 0$, there exist unique integers $k, r \geq 0$ such that

$$x \;=\; k \cdot y + r$$

and

$$r < y$$

We call k the **quotient** and r the **remainder** of x and y. For instance, given $x = 11$ and $y = 3$, we have the quotient 3 and the remainder 2 (since $11 = 3 \cdot 3 + 2$). We speak of *integer division*, or *division with remainder*, or *Euclidean division*.

We may also characterize the quotient as the largest number k such that $k \cdot y \leq x$, and define the remainder as $r = x - k \cdot y$.

There is a nice geometrical interpretation of integer division. The idea is to place boxes of the same length y next to each other into a shelf of length x. The maximal number of boxes that can be placed into the shelf is the quotient k and the length of the space remaining is the remainder $r = x - k \cdot y < y$. For instance, if the shelf has length 11 and each box has length 3, we can place at most 3 boxes into the shelf, with 2 units of length remaining.[4]

Given that x and y uniquely determine k and r, we are justified in using the notations x/y and $x \% y$ for k and r. By definition of k and r, we have

$$x \;=\; (x/y) \cdot y \,+\, x \% y$$

and

$$x \% y \;<\; y$$

for all $x \geq 0$ and $y > 0$.

From our explanations it is clear that we can compute x/y and $x \% y$ given x and y. In fact, the resulting operations x/y and $x \% y$ are essential for programming and are realized efficiently for machine integers on computers. We refer to the operations as *division* and *modulo*, or just as "div" and "mod". Accordingly, we read the applications x/y and $x \% y$ as "x div y" and "x mod y". OCaml provides both operations as primitive operations using the notations x/y and x mod y.

Digit sum

With div and mod we can decompose the decimal representation of numbers. For instance, $367 \% 10 = 7$ and $367/10 = 36$. More generally, $x \% 10$ yields the last digit of the decimal representation of x, and $x/10$ cuts off the last digit of the decimal representation of x.

Knowing these facts, we can declare a recursive function computing the digit sum of a number:

```
let rec digit_sum x =
  if x < 10 then x
  else digit_sum (x / 10) + (x mod 10)
```

For instance, we have *digit_sum* $367 = 16$. We note that *digit_sum* terminates since the argument gets smaller upon recursion.

Exercise 1.5.1 (First digit) Declare a function that yields the first digit of the decimal representation of a number. For instance, the first digit of 367 is 3.

Exercise 1.5.2 (Maximal digit) Declare a function that yields the maximal digit of the decimal representation of a number. For instance, the maximal digit of 376 is 7.

[4] A maybe simpler geometrical interpretation of integer division asks how many boxes of height y can be stacked on each other without exceeding a given height x.

Digit reversal

We now write a function *rev* that given a number computes the number represented by the reversed digital representation of the number. For instance, we want $rev\ 76 = 67$, $rev\ 67 = 76$, and $rev\ 7600 = 67$. To write *rev*, we use an important algorithmic idea. The trick is to have an additional **accumulator argument** that is initially 0 and that collects the digits we cut off at the right of the main argument. For instance, we want the **trace**

$$rev'\ 456\ 0 \;=\; rev'\ 45\ \ 6 \;=\; rev'\ 4\ \ 65 \;=\; rev'\ 0\ \ 654 \;=\; 654$$

for the helper function rev' with the accumulator argument.

We declare *rev* and the helper function rev' as follows:

```
let rec rev' x a =
  if x <= 0 then a
  else rev' (x / 10) (10 * a + x mod 10)
let rev x = rev' x 0
```

We refer to rev' as the **worker function** for *rev* and to the argument a of rev' as the **accumulator argument** of rev'. We note that rev' terminates since the first argument gets smaller upon recursion.

Greatest common divisors

Recall the notion of greatest common divisors. For instance, the greatest common divisor of 34 and 85 is the number 17. In general, two numbers x, $y \geq 0$ such that $x + y > 0$ always have a unique greatest common divisor. We assume the following rules for greatest common divisors (gcds for short):[5]

1. The gcd of x and 0 is x.
2. If $y > 0$, the gcd of x and y is the gcd of y and $x \,\%\, y$.

The two rules suffice to declare a function $gcd : int \to int \to int$ computing the gcd of two numbers $x, y \geq 0$ such that $x + y > 0$:

```
let rec gcd x y =
  if y < 1 then x
  else gcd y (x mod y)
```

The function terminates for valid arguments since $(x \,\%\, y) < y$ for $x \geq 0$ and $y \geq 1$.

Computing div and mod with repeated subtraction

We can compute x/y and $x \,\%\, y$ using repeated subtraction. To do so, we simply subtract y from x as long as we do not obtain a negative

[5] We will prove the correctness of the rules in a later chapter.

number. Then x/y is the number of successful subtractions and b is the remaining number.

```
let rec my_div x y = if x < y then 0 else 1 + my_div (x - y) y
let rec my_mod x y = if x < y then x else my_mod (x - y) y
```

We remark that both functions terminate for $x \geq 0$ and $y > 0$ since the first argument gets smaller upon recursion.

Exercise 1.5.3 (Traces) Give traces for the following applications:

$$rev' \; 678 \; 0 \qquad rev' \; 6780 \; 0 \qquad gcd \; 90 \; 120 \qquad gcd \; 153 \; 33 \qquad my\_mod \; 17 \; 5$$

We remark that the functions rev', gcd, and my_mod employ a special form of recursion known as *tail recursion*. We will discuss tail recursion in §1.11.

1.6 Mathematical Level versus Coding Level

When we design a function for OCaml or another programming language, we do this at the mathematical level. The same is true when we reason about functions and their correctness. Designing and reasoning at the mathematical level has the advantage that it serves all programming languages, not just the concrete programming language we have chosen to work with (OCaml in our case). Given the design of a function at the mathematical level, we refer to the realization of the function in a concrete programming language as *coding*. Programming as we understand it emphasizes design and reasoning over coding.

It is important to distinguish between the mathematical level and the coding level. At the mathematical level, we ignore the type *int* of machine integers and instead work with infinite *mathematical types*. In particular, we will use the types

$$\begin{array}{rcll} \mathbb{N} & : & 0, 1, 2, 3, \ldots & \text{natural numbers} \\ \mathbb{N}^+ & : & 1, 2, 3, 4, \ldots & \text{positive integers} \\ \mathbb{Z} & : & \ldots, -2, -1, 0, 1, 2, \ldots & \text{integers} \\ \mathbb{B} & : & \mathsf{false}, \ \mathsf{true} & \text{booleans} \end{array}$$

When start with the design of a function, it is helpful to fix a mathematical type for the function. Once the type is settled, we can collect equations the function should satisfy. The goal here is to come up with a collection of equations that is sufficient for computing the function.

For instance, when we design a power function, we may start with the mathematical type

$$pow : \ \mathbb{Z} \to \mathbb{N} \to \mathbb{Z}$$

and the equation

$$pow\ x\ n \;=\; x^n$$

Together, the type and the equation *specify* the function we want to define. Next we need equations that can serve as defining equations for *pow*. The specifying equation is not good enough since we assume, for the purpose of the example, that the programming language we want to code in doesn't have a power operator. We now recall that powers x^n satisfy the equations

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x \cdot x^n \end{aligned}$$

In §1.4 we have already argued that rewriting with the two equations suffices to compute all powers we can express with the type given for *pow*. Next we adapt the equations to the function *pow* we are designing:

$$\begin{aligned} pow\ x\ 0 &= 1 \\ pow\ x\ n &= x \cdot pow\ x\ (n-1) && \text{if } n > 0 \end{aligned}$$

The second equation now comes with an **application condition** replacing the pattern $n + 1$ in the equation $x^{n+1} = x \cdot x^n$.

We observe that the equations are **exhaustive** and **disjoint**, that is, for all x and n respecting the type of *pow*, the left side of one and only one of the equations applies to the **application** *pow x n*. We choose the equations as **defining equations** for *pow* and summarize our design with the mathematical definition

$$\begin{aligned} &pow : \ \mathbb{Z} \to \mathbb{N} \to \mathbb{Z} \\ &pow\ x\ 0 \;:=\; 1 \\ &pow\ x\ n \;:=\; x \cdot pow\ x\ (n-1) \quad \text{if } n > 0 \end{aligned}$$

Note that we write defining equations with the symbol ":=" to mark them as defining.

We observe that the defining equations for *pow* are **terminating**. The **termination argument** is straightforward: Each **recursion step** issued by the second equation decreases the second argument n by 1. This ensures termination since a chain $x_1 > x_2 > x_3 > \cdots$ of natural numbers cannot be infinite.

Next we code the mathematical definition as a function declaration in OCaml:

```
let rec pow x n =
  if n < 1 then 1
  else x * pow x (n - 1)
```

The switch to OCaml involves several significant issues:

1. The type of *pow* changes to $int \to int \to int$ since OCaml has no special type for $\mathbb{N}$ (as is typical for execution-oriented programming languages). Thus the OCaml function admits arguments that are not admissible for the mathematical function. We speak of **spurious arguments**.
2. To make *pow* terminating for negative n, we return 1 for all $n < 1$. We can also use the equivalent comparison $n \leq 0$. If we don't scare away from nontermination for spurious arguments, we can also use the equality test $n = 0$.
3. The OCaml type *int* doesn't give us the full mathematical type $\mathbb{Z}$ of integers but just a finite interval of machine integers.

When we design a function, there is always a mathematical level governing the coding level for OCaml. One important point about the mathematical level is that it doesn't change when we switch to another programming language. In this text, we will concentrate on the mathematical level.

When we argue the *correctness* of the function *pow*, we do this at the mathematical level using the infinite types $\mathbb{Z}$ and $\mathbb{N}$. As it comes to the realization in OCaml, we just hope that the numbers involved for particular examples are small enough so that the difference between mathematical arithmetic and *machine arithmetic* doesn't show. The reason we ignore machine integers at the mathematical level is simplicity.

1.7 More about Mathematical Functions

We use the opportunity and give mathematical definitions for some functions we already discussed. A mathematical function definition consists of a type and a system of defining equations.

Remainder

$$
\begin{aligned}
&\% :\ \mathbb{N} \to \mathbb{N}^+ \to \mathbb{N} \\
&x \,\%\, y \ :=\ x && \text{if } x < y \\
&x \,\%\, y \ :=\ (x - y) \,\%\, y && \text{if } x \geq y
\end{aligned}
$$

Recall that $\mathbb{N}^+$ is the type of positive integers $1, 2, 3, \ldots$. By using the type $\mathbb{N}^+$ for the divisor we avoid a division by zero.

Digit sum

$$
\begin{aligned}
&D : \mathbb{N} \to \mathbb{N} \\
&D(x) \;:=\; x && \text{if } x < 10 \\
&D(x) \;:=\; D(x/10) + (x \,\%\, 10) && \text{if } x \geq 10
\end{aligned}
$$

Digit Reversal

$$
\begin{aligned}
&R : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&R\ 0\ a \;:=\; a \\
&R\ x\ a \;:=\; R\,(x/10)\,(10 \cdot a + (x \,\%\, 10)) && \text{if } x > 0
\end{aligned}
$$

A system of defining equations for a function must respect the type specified for the function. In particular, the defining equations must be *exhaustive* and *disjoint* for the type specified for the function.

Often, the defining equations of a function will be *terminating* for all arguments. This is the case for each of the three function defined above. In each case, the same termination argument applies: The first argument, which is a natural number, is decreased by each recursion step.

Curly braces

We can use curly braces to write several defining equations with the same left-hand side with just one left-hand side. For instance, we may write the defining equations of the digit sum function as follows:

$$
D\,x \;:=\; \begin{cases} x & \text{if } x < 10 \\ D(x/10) + (x \,\%\, 10) & \text{if } x \geq 10 \end{cases}
$$

Total and partial functions

Functions where the defining equations terminate for all arguments are called **total**, and function where this is not the case are called **partial**. Most mathematical functions we will look at are total, but there are a few partial functions that matter. If there is a way to revise the mathematical definition of a function such that the function becomes total, we will usually do so. The reason we prefer total functions over partial functions is that equational reasoning for total functions is much easier than it is for partial functions. For correct equational reasoning about partial functions we need side conditions making sure that all function applications occurring in an equation used for reasoning do terminate.

We say that a function **diverges** for an argument if it does not terminate for this argument. Here is a function that diverges for all numbers smaller than 10 and terminates for all other numbers:

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N} \\ f(n) &:= f(n) && \text{if } n < 10 \\ f(n) &:= n && \text{if } n \geq 10 \end{aligned}$$

Exercise 1.7.1 Define a function $\mathbb{N} \to \mathbb{N}$ that terminates for even numbers and diverges for odd numbers.

Graph of a function

Abstractly, we may see a function $f : X \to Y$ as the set of all argument-result pairs $(x, f(x))$ where x is taken from the argument type X. We speak of the **graph** of a function. The graph of a function completely forgets about the definition of the function.

When we speak of a mathematical function in this text, we always include the definition of the function with a type and a system of defining equations. This information is needed so that we can compute with the function.

In noncomputational mathematics, one usually means by a function just a set of argument-result pairs.

GCD function

It is interesting to look at the mathematical version of the gcd function from §1.5:

$$\begin{aligned} G &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ G\ x\ 0 &:= x \\ G\ x\ y &:= G\ y\ (x \,\%\, y) && \text{if } y > 0 \end{aligned}$$

The defining equations are terminating since the second argument is decreased upon recursion (since $x \,\%\, y < y$ if $y > 0$). Note that the type of G admits $x = y = 0$, a case where no greatest comon divisor exists.

1.8 A Higher-Order Function for Linear Search

A *boolean test for numbers* is a function $f : \mathit{int} \to \mathit{bool}$ expressing a condition for numbers. If $f(k) = \mathsf{true}$, we say that k *satisfies* f.

Linear search is an algorithm that given a boolean test $f : \mathit{int} \to \mathit{bool}$ and a number n computes the first number $k \geq n$ satisfying f by checking f for

$$k = n,\ n+1,\ n+2,\ \ldots$$

until f is satisfied for the first time. We realize linear search with an OCaml function

$$\mathit{first} : (\mathit{int} \to \mathit{bool}) \to \mathit{int} \to \mathit{int}$$

taking the test as first argument:

```
let rec first f k =
  if f k then k
  else first f (k + 1)
```

Functions taking functions as arguments are called **higher-order functions**. Higher-order functions are a key feature of functional programming.

Recall that we have characterized in §1.5 the integer quotient x/y as the maximal number k such that $k \cdot y \leq x$. Equivalently, we may characterize x/y as the first number $k \geq 0$ such that $(k+1) \cdot y > x$ (recall the shelf interpretation). This gives us the equation

$$x/y \;=\; \mathit{first}\ (\lambda k.\ (k+1) \cdot y > x)\ 0 \qquad \text{if } x \geq 0 \text{ and } y > 0 \tag{1.1}$$

The functional argument of *first* is described with a **lambda expression**

$$\lambda k.\ (k+1) \cdot y > x$$

Lambda expressions are a common mathematical notation for describing functions without giving them a name.[6]

We use Equation 1.1 to declare a function

$$\mathit{div} : \mathit{int} \to \mathit{int} \to \mathit{int}$$

computing quotients x/y:

```
let div x y = first (fun k -> (k + 1) * y > x) 0
```

From the declaration we learn that OCaml writes lambda expressions with the words "`fun`" and "`->`". Here is a trace showing how quotients x/y are computed with *first*:

$$\begin{aligned}
\mathit{div}\ 11\ 3 &= \mathit{first}\ (\lambda k.\ (k+1) \cdot 3 > 11)\ 0 && 1 \cdot 3 \leq 11\\
&= \mathit{first}\ (\lambda k.\ (k+1) \cdot 3 > 11)\ 1 && 2 \cdot 3 \leq 11\\
&= \mathit{first}\ (\lambda k.\ (k+1) \cdot 3 > 11)\ 2 && 3 \cdot 3 \leq 11\\
&= \mathit{first}\ (\lambda k.\ (k+1) \cdot 3 > 11)\ 3 && 4 \cdot 3 > 11\\
&= 3
\end{aligned}$$

[6]The greek letter "λ" is pronounced "lambda". Sometimes lambda expressions $\lambda x.e$ are written with the more suggestive notation $x \mapsto e$.

We remark that *first* is our first inherently partial function. For instance, the function

$$\mathit{first}\ (\lambda k.\ \mathsf{false})\ :\ \mathbb{N} \to \mathbb{N}$$

diverges for all arguments. More generally, the application $\mathit{first}\ f\ n$ diverges whenever there is no $k \geq n$ satisfying f.

Exercise 1.8.1 Declare a function div' such that $\mathit{div}\ x\ y = \mathit{div}'\ x\ y\ 0$ by specializing *first* to the test $\lambda k.\ (k+1) \cdot y > x$.

Exercise 1.8.2 Declare a function $\mathit{sqrt} : \mathbb{N} \to \mathbb{N}$ such that $\mathit{sqrt}(n^2) = n$ for all n. Hint: Use *first*.

Exercise 1.8.3 Declare a terminating function *bounded_first* such that $\mathit{bounded\_first}\ f\ n$ yields the first $k \geq 0$ such that $k \leq n$ and k satisfies f.

1.9 Partial Applications

Functions described with lambda expressions can also be expressed with declared functions. To have an example, we declare *div* with *first* and a helper function *test* replacing the lambda expression:

```
let test x y k = (k + 1) * y > x
let div x y = first (test x y) 0
```

The type of the helper function *test* is $\mathit{int} \to \mathit{int} \to \mathit{int} \to \mathit{bool}$. Applying *test* to x and y yields a function of type $\mathit{int} \to \mathit{bool}$ as required by *first*. We speak of a **partial application**. Here is a trace showing how quotients x/y are computed with *first* and *test*:

$$\begin{aligned}
\mathit{div}\ 11\ 3 &= \mathit{first}\ (\mathit{test}\ 11\ 3)\ 0 && 1 \cdot 3 \leq 11\\
&= \mathit{first}\ (\mathit{test}\ 11\ 3)\ 1 && 2 \cdot 3 \leq 11\\
&= \mathit{first}\ (\mathit{test}\ 11\ 3)\ 2 && 3 \cdot 3 \leq 11\\
&= \mathit{first}\ (\mathit{test}\ 11\ 3)\ 3 && 4 \cdot 3 > 11\\
&= 3
\end{aligned}$$

We may describe the partial applications of *test* with equivalent lambda expressions:

$$\begin{aligned}
\mathit{test}\ x\ y &= \lambda k.\ (k+1) \cdot y > x\\
\mathit{test}\ 11\ 3 &= \lambda k.\ (k+1) \cdot 3 > 11
\end{aligned}$$

We can also describe partial applications of *test* to a single argument with equivalent lambda expressions:

$$\begin{aligned} \mathit{test}\ x &= \lambda yk.\ (k+1)\cdot y > x &&= \lambda y.\ \lambda k.\ (k+1)\cdot y > x \\ \mathit{test}\ 11 &= \lambda yk.\ (k+1)\cdot y > 11 &&= \lambda y.\ \lambda k.\ (k+1)\cdot y > 11 \end{aligned}$$

Note that lambda expressions with two argument variables are notation for nested lambda expressions with single arguments. We can also describe the function *test* with a nested lambda expression:

$$\mathit{test} \ = \ \lambda xyk.\ (k+1)\cdot y > x \ = \ \lambda x.\ \lambda y.\ \lambda k.\ (k+1)\cdot y > x$$

Following the nesting of lambda expressions, we may see applications and function types with several arguments as nestings of applications and function types with single arguments:

$$\begin{aligned} e_1\ e_2\ e_3 &= (e_1\ e_2)\ e_3 \\ t_1 \to t_2 \to t_3 &= t_1 \to (t_2 \to t_3) \end{aligned}$$

Note that applications group to the left and function types group to the right.

We have considered *equations between functions* in the discussion of partial applications of *test*. We consider two functions as equal if they agree on all arguments. So functions with very different definitions may be equal. In fact, two functions are equal if and only if they have the same graph. We remark that there is no algorithm deciding equality of functions in general.

Exercise 1.9.1

a) Write $\lambda xyk.\ (k+1)\cdot y > x$ as a nested lambda expression.

b) Write *test* 11 3 10 as a nested application.

c) Write $\mathit{int} \to \mathit{int} \to \mathit{int} \to \mathit{bool}$ as a nested function type.

Exercise 1.9.2 Express the one-argument functions described by the expressions x^2, x^3 and $(x+1)^2$ with lambda expressions in mathematical notation. Translate the lambda expressions to expressions in OCaml and have them type checked. Do the same for the two-argument function described by the expression $x < k^2$.

Exercise 1.9.3 (Sum functions)

a) Define a function $\mathbb{N} \to \mathbb{N}$ computing the sum $0+1+2+\cdots+n$ of the first n numbers.

b) Define a function $\mathbb{N} \to \mathbb{N}$ computing the sum $0 + 1^2 + 2^2 + \cdots + n^2$ of the first n square numbers.

c) Define a function $\mathit{sum} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ computing for a given function f the sum $f(0) + f(1) + f(2) + \cdots + f(n)$.

d) Give partial applications of the function *sum* from (c) providing specialized sum functions as asked for by (a) and (b).

1.10 Inversion of Strictly Increasing Functions

A function $f : \mathbb{N} \to \mathbb{N}$ is **strictly increasing** if

$$f(0) < f(1) < f(2) < \cdots$$

Strictly increasing functions can be inverted using linear search. That is, given a strictly increasing function $f : \mathbb{N} \to \mathbb{N}$, we can construct a function $g : \mathbb{N} \to \mathbb{N}$ such that $g(f(n)) = n$ for all n. The construction is explained by the equation

$$\mathit{first}\ (\lambda k.\ f(k+1) > f(n))\ 0 \ = \ n \tag{1.2}$$

which in turn gives us the equation

$$(\lambda x.\ \mathit{first}\ (\lambda k.\ f(k+1) > x)\ 0)\ (f(n)) \ = \ n \tag{1.3}$$

For a concrete example, let $f(n) := n^2$. Equation 1.3 tells us that

$$g(x) \ := \ \mathit{first}\ (\lambda k.\ (k+1)^2 > x)\ 0$$

is a function $\mathbb{N} \to \mathbb{N}$ such that $g(n^2) = n$ for all n. Thus we know that

$$\mathit{sqrt}\ x \ := \ \mathit{first}\ (\lambda k.\ (k+1)^2 > x)\ 0$$

computes integer square roots $\lfloor \sqrt[2]{x} \rfloor$. For instance, we have $\mathit{sqrt}(1) = 1$, $\mathit{sqrt}(4) = 2$, and $\mathit{sqrt}(9) = 3$. The **floor operator** $\lfloor x \rfloor$ converts a real number x into the greatest integer $y \leq x$.

Exercise 1.10.1 Give a trace for *sqrt* 10.

Exercise 1.10.2 Declare a function sqrt' such that $\mathit{sqrt}\ x = \mathit{sqrt}'\ x\ 0$ by specializing *first* to the test $\lambda k.\ (k+1)^2 > x$.

Exercise 1.10.3 The **ceiling operator** $\lceil x \rceil$ converts a real number into the least integer y such that $x \leq y$.

a) Declare a function computing rounded down cube roots $\lfloor \sqrt[3]{x} \rfloor$.

b) Declare a function computing rounded up cube roots $\lceil \sqrt[3]{x} \rceil$.

Exercise 1.10.4 Let $y > 0$. Convince yourself that $\lambda x.\, x/y$ inverts the strictly increasing function $\lambda n.\, n \cdot y$.

Exercise 1.10.5 Declare inverse functions for the following functions:

a) $\lambda n. n^3$

b) $\lambda n. n^k$ for $k \geq 2$

c) $\lambda n. k^n$ for $k \geq 2$

Exercise 1.10.6 Declare a function $\mathit{inv} : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ that given a strictly increasing function f yields a function inverting f. Then express the functions from Exercise 1.10.5 using *inv*.

Exercise 1.10.7 Let $f : \mathbb{N} \to \mathbb{N}$ be strictly increasing. Convince yourself that the functions

$$\begin{aligned}
&\lambda x.\, \mathit{first}\,(\lambda k.\, f(k) = x)\, 0 \\
&\lambda x.\, \mathit{first}\,(\lambda k.\, f(k) \geq x)\, 0 \\
&\lambda x.\, \mathit{first}\,(\lambda k.\, f(k+1) > x)\, 0
\end{aligned}$$

all invert f and find out how they differ.

1.11 Tail Recursion

A special form of functional recursion is **tail recursion**. Tail recursion matters since it can be executed more efficiently than general recursion. Tail recursion imposes the restriction that recursive function applications can only appear in tail positions where they directly yield the result of the function. Hence recursive applications appearing as part of another application (operator or function) are not tail recursive. Typical examples of tail recursive functions are the functions *rev*′, *gcd*, *my_mod*, and *first* we have seen before. Counterexamples for tail recursive functions are the recursive functions *pow* (the recursive application is nested into a product) and *my_div* (the recursive application is nested into a sum).

Tail recursive functions have the property that their execution can be traced in a simple way. For instance, we have the tail recursive trace

$$\begin{aligned}
\mathit{gcd}\ 36\ 132 &= \mathit{gcd}\ 132\ 36 \\
&= \mathit{gcd}\ 36\ 24 \\
&= \mathit{gcd}\ 24\ 12 \\
&= \mathit{gcd}\ 12\ 0 \\
&= 12
\end{aligned}$$

For functions where the recursion is not tail recursive, traces look more complicated, for instance

$$\begin{aligned} pow\ 2\ 3 &= 2 \cdot pow\ 2\ 2 \\ &= 2 \cdot (2 \cdot pow\ 2\ 1) \\ &= 2 \cdot (2 \cdot (2 \cdot pow\ 2\ 0)) \\ &= 2 \cdot (2 \cdot (2 \cdot 1)) \\ &= 8 \end{aligned}$$

In imperative programming languages tail recursive functions can be expressed with loops. While imperative languages are designed such that loops should be used whenever possible, functional programming languages are designed such that tail recursive functions are preferable over loops.

Often recursive functions that are not tail recursive can be reformulated as tail recursive functions by introducing an extra argument serving as accumulator argument. Here is a tail recursive version of *pow*:

```
let rec pow' x n a =
  if n < 1 then a
  else pow' x (n - 1) (x * a)
```

We explain the role of the accumulator argument with a trace:

$$\begin{aligned} pow'\ 2\ 3\ 1 &= pow'\ 2\ 2\ 2 \\ &= pow'\ 2\ 1\ 4 \\ &= pow'\ 2\ 0\ 8 \\ &= 8 \end{aligned}$$

Exercise 1.11.1 (Factorials) In mathematics, the **factorial** of a positive integer n, denoted by $n!$, is the product of all positive integers less than or equal to n:

$$n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$$

For instance,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

In addition, $0!$ is defined as 1. We capture this specification with a recursive function defined as follows:

$$\begin{aligned} &! : \mathbb{N} \to \mathbb{N} \\ 0! &:= 1 \\ (n+1)! &:= (n+1) \cdot n! \end{aligned}$$

a) Declare a function $fac : int \to int$ computing factorials.

b) Define a tail recursion function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $n! = f\,1\,n$.

c) Declare a tail recursive function $fac' : int \to int \to int$ such that $fac'\ 1$ computes factorials.

1.12 Tuples

Sometimes we want a function that returns more than one value. For instance, the time for a marathon may be given with three numbers in the format $h : m : s$, where h is the number of hours, m is the number of minutes, and s is the number of seconds the runner needed. The time of Eliud Kipchoge's world record in 2018 in Berlin was $2 : 01 : 39$. There is the constraint that $m < 60$ and $s < 60$.

OCaml has **tuples** to represent collections of values as single values. To represent the marathon time of Kipchoge in Berlin, we can use the tuple

$$(2, 1, 39) \ : \ int \times int \times int$$

consisting of three integers. The product symbol "$\times$" in the **tuple type** is written as "$*$" in OCaml. The **component types** of a tuple are not restricted to int, and there can be $n \geq 2$ **positions**, where n is called the **length** of the tuple. We may have tuples as follows:

$$\begin{aligned}(2,3) \ &: \ int \times int \\ (7, true) \ &: \ int \times bool \\ ((2,3),\ (7, true)) \ &: \ (int \times int) \times (int \times bool)\end{aligned}$$

Note that the last example nests tuples into tuples. We mention that tuples of length 2 are called **pairs**, and that tuples of length 3 are called **triples**.

We can now write two functions

$$\begin{aligned}sec :&\ \ int \times int \times int \to int \\ hms :&\ \ int \to int \times int \times int\end{aligned}$$

translating between times given in total seconds and times given as (h, m, s) tuples:

```
let sec (h,m,s) = 3600 * h + 60 * m + s
let hms x =
  let h = x / 3600 in
  let m = (x mod 3600) / 60 in
  let s = x mod 60 in
  (h,m,s)
```

Exercise 1.12.1

a) Give a tuple of length 5 where the components are the values 2 and 3.

b) Give a tuple of type $int \times (int \times (bool \times bool))$.

c) Give a pair whose first component is a pair and whose second component is a triple.

Exercise 1.12.2 (Sorting triples) Declare a function *sort* sorting triples. For instance, we want $sort\ (3,2,1) = (1,2,3)$. Designing such a function is interesting. Given a triple (x, y, z), the best solution we know of first ensures $y \le z$ and then inserts x at the correct position. Start from the code snippet

```
let sort (x,y,z) =
  let (y,z) = if y <= z then (y,z) else (z, y) in
  if x <= y then ?...?
  else ?...?
```

where the local declaration ensures $y \le z$ using shadowing.

Exercise 1.12.3 (Medians) The median of three numbers is the number in the middle. For instance, the median of $5, 0, 1$ is 1. Declare a function that takes three numbers and yields the median of the numbers.

1.13 Exceptions and Spurious Arguments

What happens when we execute the native operation $5/0$ in OCaml? Execution is aborted and an **exception** is reported:

```
Exception: Division_by_zero
```

Exceptions can be useful when debugging erroneous programs. We will say more about strings and exceptions in later chapters.

There is no equivalent to exceptions at the mathematical level. At the mathematical level we use types like $\mathbb{N}^+$ or side conditions like $y \neq 0$ to exclude undefined applications like $x/0$.

When coding a mathematical function in OCaml, we need to replace mathematical types like $\mathbb{N}$ with the OCaml type *int*. This introduces **spurious arguments** not anticipated by the mathematical function. There are different ways to cope with spurious arguments:

1. Ignore the presence of spurious arguments. This is the best strategy when you solve exercises in this text.
2. Use a wrapper function raising exceptions when spurious arguments show up. The wrapper function facilitates the discovery of situations where functions are accidentally applied to spurious arguments.

As an example, we consider the coding of the mathematical remainder function

$$
\begin{aligned}
&\mathit{rem} : \ \mathbb{N} \to \mathbb{N}^+ \to \mathbb{N} \\
&\mathit{rem}\ x\ y \ := \ x && \text{if } x < y \\
&\mathit{rem}\ x\ y \ := \ \mathit{rem}\ (x - y)\ y && \text{if } x \geq y
\end{aligned}
$$

as the OCaml function

```
let rec rem x y = if x < y then x else rem (x - y) y
```

receiving the type $\mathit{int} \to \mathit{int} \to \mathit{int}$. In OCaml we now have the spurious situation that $\mathit{rem}\ x\ 0$ diverges for all $x \geq 0$. There other spurious situations whose analysis is tedious since machine arithmetic needs to be taken into account. Using the wrapper function

```
let rem_checked x y =
  if x >=0 && y > 0 then rem x y
  else invalid_arg "rem_checked"
```

all spurious situations are uniformly handled by throwing the exception

```
Invalid_argument "rem_checked"
```

There are several new features here:

- The lazy boolean and connective `x >= 0 && y > 0` tests two conditions and is equivalent to `if x >= 0 then y > 0 else false`.
- There is the string `"rem_checked"`. Strings are values like integers and booleans and have type *string*.
- The predefined function *invalid_arg* raises an exception saying that *rem_checked* was called with spurious arguments.[7]

When an exception is raised, execution of a program is aborted and the exception raised is reported.

We use the opportunity and introduce the **lazy boolean connectives** as abbreviations for conditionals:

$$
\begin{array}{lcll}
e_1 \ \&\& \ e_2 & \rightsquigarrow & \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE false} & \textbf{lazy and} \\
e_1 \ || \ e_2 & \rightsquigarrow & \text{IF } e_1 \text{ THEN true ELSE } e_2 & \textbf{lazy or}
\end{array}
$$

Exercise 1.13.1 Consider the declaration

```
let eager_or x y = x || y
```

Find expressions e_1 and e_2 such that the expressions $e_1 \ || \ e_2$ and $\mathit{eager\_or}\ e_1\ e_2$ behave differently. Hint: Choose a diverging expression for e_2 and keep in mind that execution of a function application

[7] OCaml says "invalid argument" for "spurious argument".

executes all argument expressions. In contrast, execution of a conditional IF e_1 THEN e_2 ELSE e_3 executes e_1 and then either e_2 or e_3, but not both.

Exercise 1.13.2 (Sorting triples) Recall Exercise 1.12.2. With lazy boolean connectives a function sorting triples can be written without much thinking by doing a naive case analysis considering the alternatives x is in the middle or y is in the middle or z is in the middle.

1.14 Polymorphic Functions

Consider the declaration of a projection function for pairs:

```
let fst (x,y) = x
```

What type does *fst* have? Clearly, $int \times int \to int$ and $bool \times int \to bool$ are both types admissible for *fst*. In fact, every type $t_1 \times t_2 \to t_1$ is admissible for *fst*. Thus there are infinitely many types admissible for *fst*.

OCaml solves the situation by typing *fst* with the polymorphic type

$$\forall\alpha\beta.\ \alpha \times \beta \to \alpha$$

A **polymorphic type** is a type scheme whose quantified variables (α and β in the example) can be instantiated with all types. The **instances** of the polymorphic type above are all types $t_1 \times t_2 \to t_1$ where the types t_1 and t_2 can be freely chosen. When a polymorphically typed identifier is used in an expression, it can be used with any instance of its polymorphic type. Thus $fst\,(1, 2)$ and $fst\,(\mathtt{true}, 5)$ are both well-typed expressions.

Here is a polymorphic swap function for pairs:

```
let swap (x,y) = (y,x)
```

OCaml will type *swap* with the polymorphic type

$$\forall\alpha\beta.\ \alpha \times \beta \to \beta \times \alpha$$

This is in fact the **most general polymorphic type** that is admissible for *swap*. Similarly, the polymorphic type given for *fst* is the most general polymorphic type admissible for *fst*. OCaml will always derive most general types for function declarations.

Exercise 1.14.1 Declare functions admitting the following polymorphic types:

a) $\forall\alpha.\ \alpha \to \alpha$

b) $\forall\alpha\beta.\ \alpha \to \beta \to \alpha$

c) $\forall\alpha\beta\gamma.\ (\alpha \times \beta \to \gamma) \to \alpha \to \beta \to \gamma$

d) $\forall\alpha\beta\gamma.\ (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$

e) $\forall\alpha\beta.\ \alpha \to \beta$

1.15 Iteration

Given a function $f : t \to t$, we write $f^n(x)$ for the n-fold application of f to x. For instance, $f^0(x) = x$, $f^1(x) = f(x)$, $f^2(x) = f(f(x))$, and $f^3(x) = f(f(f(x)))$. More generally, we have

$$f^{n+1}(x) = f^n(fx)$$

Given an **iteration** $f^n(x)$, we call f the **step function** and x the **start value** of the iteration.

With iteration we can compute sums, products, and powers of non-negative integers just using additions $x + 1$:

$$\begin{aligned} x + n &= x + 1 \ \cdots \ + 1 &&= (\lambda a.a + 1)^n(x) \\ n \cdot x &= 0 + x \ \cdots \ + x &&= (\lambda a.a + x)^n(0) \\ x^n &= 1 \cdot x \ \cdots \ \cdot x &&= (\lambda a.a \cdot x)^n(1) \end{aligned}$$

Exploiting commutativity of addition and multiplication, we arrive at the defining equations

$$\begin{aligned} \mathit{succ}\ x &:= x + 1 \\ \mathit{add}\ x\ n &:= \mathit{succ}^n(x) \\ \mathit{mul}\ n\ x &:= (\mathit{add}\ x)^n(0) \\ \mathit{pow}\ x\ n &:= (\mathit{mul}\ x)^n(1) \end{aligned}$$

We define a polymorphic **iteration operator**

$$\begin{aligned} &\mathit{iter} :\ \forall\alpha.\ (\alpha \to \alpha) \to \mathbb{N} \to \alpha \to \alpha \\ &\mathit{iter}\ f\ 0\ x \ := \ x \\ &\mathit{iter}\ f\ (n+1)\ x \ := \ \mathit{iter}\ f\ n\ (f\ x) \end{aligned}$$

so that we can obtain iterations $f^n(x)$ as applications $\mathit{iter}\, f\, n\, x$ of the operator. Note that the function *iter* is polymorphic, higher-order, and tail-recursive. In OCaml, we will use the declaration

```
let rec iter f n x =
  if n < 1 then x
  else iter f (n - 1) (f x)
```

Functions for addition, multiplication, and exponentiation can now be declared as follows:

```
let succ x = x + 1
let add x y = iter succ y x
let mul x y = iter (add y) x 0
let pow x y = iter (mul x) y 1
```

Note that these declarations are non-recursive. Thus termination needs only be checked for *iter*, where it is obvious (2nd argument is decreased).

Exercise 1.15.1 Declare a function testing evenness of numbers by iterating on booleans. What do you have to change to obtain a function checking oddness?

Exercise 1.15.2 We have the equation

$$f^{n+1}(x) \;=\; f(f^n(x))$$

providing for an alternative, non-tail-recursive definition of an iteration operator. Give the mathematical definition and the declaration in OCaml of an iteration operator using the above equation.

1.16 Iteration on Pairs

Using iteration and successor as basic operations on numbers, we have defined functions computing sums, products, and powers of nonnegative numbers. We can also define a **predecessor function**[8]

$$\begin{aligned} \mathit{pred}:\;& \mathbb{N}^+ \to \mathbb{N} \\ \mathit{pred}\,(n+1) \;&:=\; n \end{aligned}$$

just using iteration and successor (the successor of an integer x is $x+1$). The trick is to iterate on pairs. We start with the pair $(0,0)$ and iterate with a step function f such that $n+1$ iterations yield the pair $(n, n+1)$. For instance, the iteration

$$f^5(0,0) = f^4(0,1) = f^3(1,2) = f^2(2,3) = f(3,4) = (4,5)$$

using the **step function**

$$f(a,k) = (k, k+1)$$

yields the predecessor of 5 as the first component of the computed pair. More generally, we have

$$(n, n+1) \;=\; f^{n+1}\,(0,0)$$

We can now declare a predecessor function as follows:

[8] the predecessor of an integer x is $x - 1$.

```
let pred n = fst (iter (fun (a,k) -> (k, succ k)) n (0,0))
```

Iteration on pairs is a powerful computation scheme. Our second example concerns the sequence of **Fibonacci numbers**

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

which is well known in mathematics. The sequence is obtained by starting with $0, 1$ and then adding new elements as the sum of the two preceding elements. We can formulate this method as a recursive function

$$\begin{aligned} &\mathit{fib} :\ \mathbb{N} \to \mathbb{N} \\ &\quad \mathit{fib}(0) \ := \ 0 \\ &\quad \mathit{fib}(1) \ := \ 1 \\ &\mathit{fib}\,(n+2) \ := \ \mathit{fib}\,(n) + \mathit{fib}\,(n+1) \end{aligned}$$

Incidentally, this is the first recursive function we see where the recursion is **binary** (two recursive applications) rather than **linear** (one recursive application). Termination follows with the usual argument that each recursion step decreases the argument.

If we look again at the rule generating the Fibonacci sequence, we see that we can compute the sequence by starting with the pair $(0, 1)$ and iterating with the step function $f(a, b) = (b, a + b)$. For instance,

$$f^5(0,1) = f^4(1,1) = f^3(1,2) = f^2(2,3) = f(3,5) = (5,8)$$

yields the pair $(\mathit{fib}(5),\ \mathit{fib}(6))$. More generally, we have

$$(\mathit{fib}(n),\ \mathit{fib}(n+1)) \ = \ (\lambda(a,b).(b, a+b))^n\,(0,1)$$

Thus we can declare an iterative Fibonacci function as follows:

```
let fibi n = fst (iter (fun (a,b) -> (b, a + b)) n (0,1))
```

In contrast to the previously defined function *fib*, function *fibi* requires only tail recursion as provided by *iter*.

Exercise 1.16.1 Declare a function computing the sum $0+1+2+\cdots+n$ by iteration starting from the pair $(0, 1)$.

Exercise 1.16.2 Declare a function $f : \mathbb{N} \to \mathbb{N}$ computing the sequence

$$0,\ 1,\ 1,\ 2,\ 4,\ 7,\ 13,\ \ldots$$

obtained by starting with $0, 1, 1$ and then adding new elements as the sum of the three preceding elements. For instance, $f(3) = 2$, $f(4) = 4$, and $f(5) = 7$.

Exercise 1.16.3 Functions defined with iteration can always be elaborated into tail-recursive functions not using iteration. If the iteration is on pairs, one can use separate accumulator arguments for the components of the pairs. Follow this recipe and declare a tail-recursive function fib' such that $\mathit{fib}'\, n\ 0\ 1 = \mathit{fib}(n)$.

Exercise 1.16.4 Recall the definition of factorials $n!$ from Exercise 1.11.1.

a) Give a step function f such that $(n!, n) = f^n(1, 0)$.
b) Declare a function *faci* computing factorials with iteration.
c) Declare a tail-recursive function fac' such that $\mathit{fac}'\, n\ 1\ 0 = n!$. Follow the recipe from Exercise 1.16.3.

1.17 Computing Primes

A **prime number** is an integer greater 1 that cannot be obtained as the product of two integers greater 1. There are infinitely many prime numbers. The sequence of prime numbers starts with

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ \cdots$$

We want to declare a function $\mathit{nth\_prime} : \mathbb{N} \to \mathbb{N}$ that yields the elements of the sequence starting with $\mathit{nth\_prime}\ 0 = 2$. Assuming a primality test $\mathit{prime} : \mathit{int} \to \mathit{bool}$, we can declare *nth_prime* as follows:

```
let next_prime x = first prime (x + 1)
let nth_prime n = iter next_prime n 2
```

Note that $\mathit{next\_prime}\, x$ yields the first prime greater than x. Also note that the elegant declarations of *next_prime* and *nth_prime* are made possible by the higher-order functions *first* and *iter*.

It remains to come up with a **primality test** (a test checking whether an integer is a prime number). Here are 4 equivalent characterizations of primality of x assuming that x, k, and n are natural numbers:

1. $x \geq 2 \wedge \forall k \geq 2. \forall n \geq 2.\ \ x \neq k \cdot n$
2. $x \geq 2 \wedge \forall k,\ 2 \leq k < x.\ \ x \,\%\, k > 0$
3. $x \geq 2 \wedge \forall k > 1,\ k^2 \leq x.\ \ x \,\%\, k > 0$
4. $x \geq 2 \wedge \forall k,\ 1 < k \leq \sqrt[2]{x}.\ \ x \,\%\, k > 0$

Characterization (1) is close to the informal definition of prime numbers. Characterizations (2), (3), and (4) are useful for our purposes since they can be realized algorithmically. Starting from (1), we see that $x - 1$ can be used as an upper bound for k and n. Thus we have to test $x = k \cdot n$

only for finitely many k and n, which can be done algorithmically. The next thing we see is that is suffices to have k because n can be kept implicit using the remainder operation. Finally, we see that it suffices to test for k such that $k^2 \leq x$ since we can assume $n \geq k$. Thus we can sharpen the upper bound from $x - 1$ to $\sqrt[2]{x}$.

Here we choose the second characterization to declare a primality test in OCaml and leave the realization of the computationally faster fourth characterization as an exercise.

To test the bounded universal quantification in the second characterization, we declare a higher-order function

$$\mathit{forall} : \ \mathit{int} \to \mathit{int} \to (\mathit{int} \to \mathit{bool}) \to \mathit{bool}$$

such that

$$\mathit{forall}\ m\, n\, f = \mathsf{true} \ \longleftrightarrow\ \forall k, m \leq k \leq n.\ \ f\, k = \mathsf{true}$$

using the defining equations[9]

$$\mathit{forall}\ m\ n\ f \ := \begin{cases} \mathsf{true} & m > n \\ f\, m \ \texttt{\&\&}\ \mathit{forall}\ (m+1)\ n\ f & m \leq n \end{cases}$$

The function *forall* terminates since $|n + 1 - m| \geq 0$ decreases with every recursion step.

We now define a primality test based on the second characterization:

$$\mathit{prime}\ x := x \geq 2\ \ \texttt{\&\&}\ \ \mathit{forall}\ 2\ (x-1)\ (\lambda k.\ x \,\%\, k > 0)$$

The discussion of primality tests makes it very clear that programming involves mathematical reasoning.

Efficient primality tests are important in cryptography and other areas of computer science. The naive versions we have discussed here are known as trial division algorithms and are too slow for practical purposes.

Exercise 1.17.1 Express *forall* with *iter*.

Exercise 1.17.2 Declare a test $\mathit{exists} : \mathit{int} \to \mathit{int} \to (\mathit{int} \to \mathit{bool}) \to \mathit{bool}$ such that $\mathit{exists}\, m\, n\, f = \mathsf{true} \longleftrightarrow \exists k,\ m \leq k \leq n.\ fk = \mathsf{true}$ in two ways:

a) Directly following the design of *forall*.

[9] We use the curly brace as an abbreviation to write two equations as a single equation.

b) Using *forall* and boolean negation $not : bool \to bool$.

Exercise 1.17.3 Declare a primality test based on the fourth characterization (i.e., upper bound $\sqrt[2]{x}$). Convince yourself with an OCaml interpreter that testing with upper bound $\sqrt[2]{x}$ is much faster on large primes (check 479,001,599 and 87,178,291,199).

Exercise 1.17.4 Explain why the following functions are primality tests:

a) $\lambda x.\ x \geq 2$ && $first\ (\lambda k.\ x \,\%\, k = 0)\ 2\ =\ x$

b) $\lambda x.\ x \geq 2$ && $(first\ (\lambda k.\ k^2 \geq x \mid\mid x \,\%\, k = 0)\ 2)^2 > x$

Hint for (b): Let k be the number the application of *first* yields. Distinguish three cases: $k^2 < x$, $k^2 = x$, and $k^2 > x$.

Exercise 1.17.5 Convince yourself that the four characterization of primality given above are equivalent.

1.18 Polymorphic Exception Raising and Equality Testing

Recall the predefined function *invalid_arg* discussed in §1.13. Like every function in OCaml, *invalid_arg* must be accommodated with a type. It turns out that the natural type for *invalid_arg* is a polymorphic type:

$$invalid\_arg :\ \forall\alpha.\ string \to \alpha$$

With this type an application of *invalid_arg* can be typed with whatever type is required by the context of the application. Since evaluation of the application raises an exception and doesn't yield a value, the return type of the function doesn't matter for evaluation.

Like functions, operations must be accommodated with types. So what is the type of the equality test? OCaml follows common mathematical practice and admits the equality test for all types. To this purpose, the operator testing equality is accommodated with a polymorphic type[10]:

$$= :\ \forall\alpha.\ \alpha \to \alpha \to bool$$

The polymorphic type promises more than the equality test delivers. There is the general problem that a meaningful equality test for functions cannot be realized computationally. OCaml bypasses the problem by

[10] In fact, OCaml also equips the less-than test ($<$) with the same polymorphic type and lifts this test to compound types such as tuples and constructor types.

the crude provision that an equality test on functions raises an invalid argument exception.

We assume that functions at the mathematical level always return values and do not raise exceptions. We handle division by zero by assuming that it returns some value, say 0. Similarly, we handle an equality test for functions by assuming that it always returns `true`. When reasoning at the mathematical level, we will avoid situations where the ad hoc definitions come into play, following common mathematical practice.

1.19 Summary

After working through this chapter you should be able to design and code functions computing powers, integer quotients and remainders, digit sums, digit reversals, and integer roots. You should understand that the design of functions happens at a mathematical level using mathematical types and equations. A given design can then be refined into a program in a given programming language. In this text we are using OCaml as programming language, assuming that this is the first programming language you see.[11]

You also saw a first higher-order function *first*, which can be used to obtain integer quotients and integer roots with a general scheme known as linear search. You will see many more examples of higher-order functions expressing basic computational schemes.

The model of computation we have assumed in this chapter is rewriting with defining equations. In this model, recursion appears in a natural way. You will have noticed that recursion is the feature where things get really interesting. We also have discussed tail recursion, a restricted form of recursion with nice properties we will study more carefully as we go on. All recursive functions we have seen in this chapter have equivalent tail recursive formulations (often using an accumulator argument).

Finally, we have seen tuples, which are compound values combining several values into a single value.

[11] Most readers will have done some programming in some programming language before starting with this text. Readers of this group often face the difficulty that they invest too much energy on mapping back the new things they see here to the form of programming they already understand. Since functional programming is rather different from other forms of programming, it is essential that you open yourself to the new ideas presented here. Keep in mind that a good programmer quickly adapts to new ways of thinking and to new programming languages.

2 Lists

Lists are a basic mathematical data structure providing a recursive representation for finite sequences. Lists are essential for programming. OCaml, and functional programming languages in general, accommodate lists in a mathematically clean way. Three interesting problems we will attack with lists are decimal representation, sorting, and prime factorization:

$$\begin{aligned} \mathit{dec}\ 735 &= [7, 3, 5] \\ \mathit{sort}\ [7, 2, 7, 6, 3, 4, 5, 3] &= [2, 3, 3, 4, 5, 6, 7, 7] \\ \mathit{prime\_fac}\ 735 &= [3, 5, 7, 7] \end{aligned}$$

2.1 Nil and Cons

A list represents a finite sequence $[x_1, \ldots, x_n]$ of values. All **elements** of a list must have the same type. A **list type** $\mathcal{L}(t)$ contains all lists whose elements are of type t; we speak of **lists over** t. For instance,

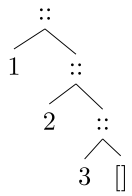
$$\begin{aligned} [1, 2, 3] &: \mathcal{L}(\mathbb{Z}) \\ [\mathsf{true}, \mathsf{true}, \mathsf{false}] &: \mathcal{L}(\mathbb{B}) \\ [(\mathsf{true}, 1), (\mathsf{true}, 2), (\mathsf{false}, 3)] &: \mathcal{L}(\mathbb{B} \times \mathbb{Z}) \\ [[1, 2], [3], []] &: \mathcal{L}(\mathcal{L}(\mathbb{Z})) \end{aligned}$$

All lists are obtained from the **empty list** $[]$ using the binary constructor **cons** written as "::":

$$\begin{aligned} [1] &= 1 :: [] \\ [1, 2] &= 1 :: (2 :: []) \\ [1, 2, 3] &= 1 :: (2 :: (3 :: [])) \end{aligned}$$

The empty list $[]$ also counts as a constructor and is called **nil**. Given a nonempty list $x :: l$ (i.e., a list obtained with cons), we call x the **head** and l the **tail** of the list.

It is important to see lists as trees. For instance, the list $[1, 2, 3]$ may be depicted as the tree



The tree representation shows how lists are obtained with the constructors nil and cons. It is important to keep in mind that the bracket notation $[x_1, \dots, x_n]$ is just notation for a list obtained with n applications of cons from nil. Also keep in mind that every list is obtained with either the constructor nil or the constructor cons.

Notationally, cons acts as an infix operator grouping to the right. Thus we can omit the parentheses in $1 :: (2 :: (3 :: []))$. Moreover, we have $[x_1, \dots, x_n] = x_1 :: \cdots :: x_n :: []$.

Given a list $[x_1, \dots, x_n]$, we call the values $x_1, \dots, x_n$ the **elements** or the **members** of the list.

Despite the fact that tuples and lists both represent sequences, tuple types and list types are quite different:

- A tuple type $t_1 \times \cdots \times t_n$ admits only tuples of length n, but may fix different types for different components.
- A list type $\mathcal{L}(t)$ admits lists $[x_1, \dots, x_n]$ of any length but fixes a single type t for all elements.

Exercise 2.1.1 Give the types of the following lists and tuples.

a) $[1, 2, 3]$

b) $(1, 2, 3)$

c) $[(1, 2), (2, 3)]$

d) $((1, 2), (2, 3))$

e) $[[1, 2], [2, 3]]$

2.2 Basic List Functions

The fact that all lists are obtained with nil and cons facilitates the definition of basic operations on lists. We start with the definition of a polymorphic function

$$
\begin{aligned}
\mathit{length} :\ & \forall \alpha.\ \mathcal{L}(\alpha) \to \mathbb{N} \\
\mathit{length}\ [] \ &:= \ 0 \\
\mathit{length}\ (x :: l) \ &:= \ 1 + \mathit{length}\ l
\end{aligned}
$$

that yields the **length** of a list. We have $length\,[x_1, \dots, x_n] = n$. Another prominent list operation is **concatenation**:

$$\begin{aligned}
@ :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\
[]\ @\ l_2 \ &:= \ l_2 \\
(x :: l_1)\ @\ l_2 \ &:= \ x :: (l_1\ @\ l_2)
\end{aligned}$$

We have $[x_1, \dots, x_m]\ @\ [y_1, \dots, y_n] = [x_1, \dots, x_m, y_1, \dots, y_n]$. We call $l_1\ @\ l_2$ the **concatenation** of l_1 and l_2.

How can we define an operation **reversing** lists:

$$rev\,[x_1, \dots, x_n] = [x_n, \dots, x_1]$$

For instance, $rev\,[1,2,3] = [3,2,1]$. To define rev, we need defining equations for nil and cons. The defining equation for nil is obvious, since the reversal of the empty list is the empty list. For cons we use the equation $rev\,(x :: l) = rev\,(l)\ @\ [x]$ which expresses a basic fact about reversal. This brings us to the following definition of **list reversal**:

$$\begin{aligned}
rev :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\
rev\ []\ &:= \ [] \\
rev\ (x :: l)\ &:= \ rev\,(l)\ @\ [x]
\end{aligned}$$

We can also define a tail recursive list reversal function. As usual we need an accumulator argument. The resulting function combines reversal and concatenation:

$$\begin{aligned}
rev_append :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\
rev_append\ []\ l_2 \ &:= \ l_2 \\
rev_append\ (x :: l_1)\ l_2 \ &:= \ rev_append\ l_1\ (x :: l_2)
\end{aligned}$$

We have $rev\,(l) = rev\_append\ l\ []$. The following trace shows the defining equations of rev_append at work:

$$\begin{aligned}
rev_append\ [1,2,3]\ [] \ &= \ rev_append\ [2,3]\ [1] \\
&= \ rev_append\ [3]\ [2,1] \\
&= \ rev_append\ []\ [3,2,1] \\
&= \ [3,2,1]
\end{aligned}$$

The functions defined so far are all defined by **list recursion**.[1] List recursion means that there is no recursion for the empty list, and that

[1] List recursion is better known as *structural recursion on lists*.

for every nonempty list $x :: l$ the recursion is on the tail l of the list. List recursion always terminates since lists are obtained from nil with finitely many applications of cons.

Another prominent list operation is

$$\begin{aligned} map :\ & \forall\alpha\beta.\ (\alpha \to \beta) \to \mathcal{L}(\alpha) \to \mathcal{L}(\beta) \\ map\ f\ []\ &:=\ [] \\ map\ f\ (x :: l)\ &:=\ f\,x :: map\,f\,l \end{aligned}$$

We have $map\,f\,[x_1,\dots,x_n] = [f\,x_1,\dots,f\,x_n]$. We say that *map* applies the function f **pointwise** to a list. Note that *map* is defined once more with list recursion.

Finally, we define a function that yields the list of all integers between two numbers m and n (including m and n):

$$\begin{aligned} & seq :\ \mathbb{Z} \to \mathbb{Z} \to \mathcal{L}(\mathbb{Z}) \\ & seq\ m\ n\ :=\ \text{IF } m > n \text{ THEN } []\ \text{ELSE } m :: seq\ (m+1)\,n \end{aligned}$$

For instance, $seq\ {-1}\ 5 = [-1, 0, 1, 2, 4, 5]$. This time the recursion is not on a list but on the number m. The recursion terminates since every recursion step decreases $n - m$ and recursion stops once $n - m < 0$.

Exercise 2.2.1 Define a function $null : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathbb{B}$ testing whether a list is the empty list. Do not use the equality test.

Exercise 2.2.2 Consider the expression $1 :: 2 :: []\ @\ 3 :: 4 :: []$.

1. Put in the redundant parentheses.
2. Give the list the expression evaluates to in bracket notation.
3. Give the tree representation of the list the expression evaluates to.

Exercise 2.2.3 Decide for each of the following equations whether it is well-typed, and, in case it is well-tyoed, whether it is true. Assume $l_1, l_2 :\ List(\mathbb{N})$.

a) $1 :: 2 :: 3 = 1 :: [2, 3]$
b) $1 :: 2 :: 3 :: [] = 1 :: (2 :: [3])$
c) $l_1 :: [2] = l_1\ @\ [2]$
d) $(l_1\ @\ [2])\ @\ l_2 = l_1\ @\ (2 :: l_2)$
e) $(l_1 :: 2)\ @\ l_2 = l_1\ @\ (2 :: l_2)$
f) $map\ (\lambda x.\ x^2)\ [1, 2, 3] = [1, 4, 9]$
g) $rev\,(l_1\ @\ l_2) = rev\,l_2\ @\ rev\,l_1$

Exercise 2.2.4 Create a tail-recursive version of *append* by means of other functions defined in this section.

2.3 List Functions in OCaml

Given a mathematical definition of a list function, it is straightforward to declare the function in OCaml. The essential new construct are so-called **match expressions** making it possible to discriminate between empty and nonempty lists. Here is a declaration of a length function:

```
let rec length l =
  match l with
  | [] -> 0
  | x :: l -> 1 + length l
```

The match expression realizes a case analysis on lists using separate **rules** for the empty list and for nonempty lists. The left hand sides of the rules are called **patterns**. The cons pattern applies to nonempty lists and binds the **local variables** x and l to the head and the tail of the list.

Note that the pattern variable l introduced by the second rule of the match shadows the argument variable l in the declaration of *length*. The shadowing could be avoided by using a different variable name for the pattern variable (for instance, l').

Below are declarations of OCaml functions realizing the functions *append*, *rev_append*, and *map* defined before. Note how the defining equations translate into rules of match expressions.

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: l1 -> x :: append l1 l2

let rec rev_append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: l1 -> rev_append l1 (x :: l2)

let rec map f l =
  match l with
  | [] -> []
  | x :: l -> f x :: map f l
```

For each of the function declarations, OCaml infers the polymorphic type we have specified with the mathematical definition.

We remark that OCaml realizes the bracket notation for lists using semicolons to separate elements. For instance, the list $[1, 2, 3]$ is written as $[1; 2; 3]$ in OCaml.

OCaml provides predefined functions for lists as **fields** of a predefined **standard module** *List*. For instance:

```
List.length     : 'a list -> int
List.append     : 'a list -> 'a list -> 'a list
List.rev_append : 'a list -> 'a list -> 'a list
List.rev        : 'a list -> 'a list
List.map        : ('a -> 'b) -> 'a list -> 'b list
```

The above listing uses OCaml notation:

- **Dot notation** is used to name the fields of modules; for instance, *List.append* denotes the field *append* of the module *List*.
- List types $\mathcal{L}(t)$ are written in reverse order as "*t list*".
- Type variables are written with a leading quote, for instance, "`'a`"
- The quantification prefix of polymorphic types is suppressed, relying on the assumption that all occurring type variables are quantified.

Here are further notational details concerning lists in OCaml:

- List concatenation *List.append* is also available through the infix operator "@".
- The infix operators "::" and "@" both group to the right, and "::" takes its arguments before "@". For instance,

$$1 :: 2 :: [3;4]@[5] \quad \rightsquigarrow \quad (1 :: (2 :: [3;4]))@[5]$$

- The operators "::" and "@" take their arguments before comparisons and after arithmetic operations.

Exercise 2.3.1 Declare a function *seq* following the mathematical definition in the previous section.

Exercise 2.3.2 (Init) Declare a polymorphic function *init* such that $init\ n\ f = [f(0), \dots, f(n-1)]$ for $n \geq 0$. Note that n is the length of the result list. Write your function with a tail-recursive helper function. Make sure your function agrees with OCaml's predefined function *List.init*. Use *List.init* to declare polymorphic functions that yield lists $[f(m), f(m+1), \dots, f(m+n-1)]$ and $[f(m), f(m+1), \dots, f(n)]$.

Exercise 2.3.3 Declare a function $\mathit{flatten} : \forall\alpha.\ \mathcal{L}(\mathcal{L}(\alpha)) \to \mathcal{L}(\alpha)$ concatenating the lists appearing as elements of a given list:

$$\mathit{flatten}\,[l_1, \dots, l_n] = l_1 \mathbin{@} \cdots \mathbin{@} l_n \mathbin{@} []$$

For instance, we want $\mathit{flatten}\,[[1,2],[],[3],[4,5]] = [1,2,3,4,5]$.

Exercise 2.3.4 (Decimal Numbers) With lists we have a mathematical representation for decimal numbers. For instance, the decimal representation for the natural number 1234 is the list [1,2,3,4].

a) Declare a function $dec : \mathbb{N} \to \mathcal{L}(\mathbb{N})$ that yields the decimal number for a natural number. For instance, we want $dec\,1324 = [1, 3, 2, 4]$.
b) Declare a function $num : \mathcal{L}(\mathbb{N}) \to \mathbb{N}$ that converts decimal numbers into numbers: $num\,(dec\ n) = n$.

Hint: Declare num with a tail-recursive function num' such that, for instance,

$$\begin{aligned} num\,[1, 2, 3] &= num'\,[1, 2, 3]\,0 \\ &= num'\,[2, 3]\,1 \\ &= num'\,[3]\,12 \\ &= num'\,[\,]\,123 = 123 \end{aligned}$$

Exercise 2.3.5 Declare functions

$$\begin{aligned} zip :\ & \forall\alpha\beta.\ \mathcal{L}(\alpha) \to \mathcal{L}(\beta) \to \mathcal{L}(\alpha \times \beta) \\ unzip :\ & \forall\alpha\beta.\ \mathcal{L}(\alpha \times \beta) \to \mathcal{L}(\alpha) \times \mathcal{L}(\beta) \end{aligned}$$

such that

$$\begin{aligned} zip\ [x_1, \dots, x_n]\ [y_1, \dots, y_n] \ &= \ [(x_1, y_1), \dots, (x_n, y_n)] \\ unzip\ [(x_1, y_1), \dots, (x_n, y_n)] \ &= \ ([x_1, \dots, x_n],\ [y_1, \dots, y_n]) \end{aligned}$$

2.4 Fine Points About Lists

We speak of the collection of list types $\mathcal{L}(t)$ as a **type family**. We may describe the family of list types with the grammar

$$\mathcal{L}(\alpha) \ ::= \ [\,] \ \mid \ \alpha :: \mathcal{L}(\alpha)$$

fixing the **constructors** nil and cons. Speaking semantically, every value of a list type is obtained with either the constructor nil or the constructor cons. Moreover, $[\,]$ is a value that is a member of every list type $\mathcal{L}(t)$, and $v_1 :: v_2$ is a value that is a member of a list type $\mathcal{L}(t)$ if the value v_1 is a member of the type t and the value v_2 is a member of the type $\mathcal{L}(t)$.

As it comes to type checking, the constructors nil and cons are accommodated with polymorphic types:

$$\begin{aligned} [\,] \ &: \ \forall\alpha.\ \mathcal{L}(\alpha) \\ (::) \ &: \ \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \end{aligned}$$

OCaml comes with the peculiarity that constructors taking arguments (e.g., cons) can only be used when applied to all arguments. We can bypass this restriction by declaring a polymorphic function applying the cons constructor:

$$\begin{aligned} cons :\; & \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\ cons\ x\ l \;&:=\; x :: l \end{aligned}$$

OCaml provides this function as *List.cons*.

2.5 Membership and List Quantification

We define a polymorphic function that tests whether a value appears as element of a list:

$$\begin{aligned} mem :\; & \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathbb{B} \\ mem\ \_\ [\,] \;&:=\; \mathsf{false} \\ mem\ x\ (y :: l) \;&:=\; (x = y)\ \mathsf{||}\ mem\ x\ l \end{aligned}$$

Note that *mem* tail-recurses on the list argument. Also recall the discussion of OCaml's polymorphic equality test in §1.18. We will write $x \in l$ to say that x is an element of the list l.

The structure of the membership test can be generalized with a polymorphic function that for a test and a list checks whether some element of the list satisfies the test:

$$\begin{aligned} exists :\; & \forall\alpha.\ (\alpha \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathbb{B} \\ exists\ p\ [\,] \;&:=\; \mathsf{false} \\ exists\ p\ (x :: l) \;&:=\; p\,x\ \mathsf{||}\ exists\ p\ l \end{aligned}$$

The expression

$$exists\ (\lambda x.\ x = 5) \;:\; \mathcal{L}(\mathbb{Z}) \to \mathbb{B}$$

now gives us a test that checks whether a lists of numbers contains the number 5. More generally, we have

$$mem\ x\ l \;=\; exists\ ((=)x)\ l$$

Exercise 2.5.1 Declare *mem* and *exists* in OCaml. For *mem* consider two possibilities, one with *exists* and one without helper function.

Exercise 2.5.2 Convince yourself that *exists* is tail-recursive (by eliminating the derived form $\mathsf{||}$).

Exercise 2.5.3 Declare a tail-recursive function

$$forall : \ \forall\alpha.\ (\alpha \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathbb{B}$$

testing whether all elements of a list satisfy a given test. Consider two possibilities, one without a helper function, and one with *exists* exploiting the equivalence $(\forall x \in l.\ p(x)) \longleftrightarrow (\neg\exists x \in l.\ \neg p(x))$.

Exercise 2.5.4 Declare a function $count : \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathbb{N}$ that counts how often a value appears in a list. For instance, we want $count\ 5\ [2, 5, 3, 5] = 2$.

Exercise 2.5.5 (Inclusion) Declare a function

$$incl : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B}$$

which tests whether all elements of the first list are elements of the second list.

Exercise 2.5.6 (Repeating lists) A list is **repeating** if it has an element appearing at two different positions. For instance, $[2, 5, 3, 5]$ is repeating and $[2, 5, 3]$ is not repeating.

a) Declare a function testing whether a list is repeating.

b) Declare a function testing whether a list is non-repeating.

c) Declare a function that given a list l yields a non-repeating list containing the same elements as l.

2.6 Head and Tail

In OCaml we can declare polymorphic functions

$$\begin{aligned} hd &: \ \forall\alpha.\ \mathcal{L}(\alpha) \to \alpha \\ tl &: \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \end{aligned}$$

that yield the head and the tail of nonempty lists:

```
let hd l =
  match l with
  | [] -> failwith "hd"
  | x :: _ -> x

let tl l =
  match l with
  | [] -> failwith "tl"
  | _ :: l -> l
```

Both functions raise exceptions when applied to the empty list. Note the use of the underline symbol "_" for pattern variables that are not used in a rule. Also note the use of the predefined function

$$\mathit{failwith} : \ \forall\alpha.\ \mathit{string} \to \alpha$$

raising an exception *Failure s* carrying the string s given as argument. Interesting is the polymorphic type of *failwith* making it possible to type an application of *failwith* with whatever type is required by the context of the application.

At the mathematical level we don't admit exceptions. Thus we cannot define a polymorphic function

$$\mathit{hd} : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \alpha$$

since we don't have a value we can return for the empty list over α.

Exercise 2.6.1 Define a polymorphic function $\mathit{tl} : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ returning the tail of nonempty lists. Do not use exceptions.

Exercise 2.6.2 Declare a polymorphic function $\mathit{last} : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \alpha$ returning the last element of a nonempty lists.

2.7 Position Lookup

The **positions** of a list are counted from left to right starting with the number 0. For instance, the list $[5, 6, 5]$ has the positions $0, 1, 2$; moreover, the element at position 1 is 6, and the value 5 appears at the positions 0 and 2. The empty list has no position. More generally, a list of length n has the positions $0, \dots, n-1$.

We declare a tail-recursive *lookup function*

$$\mathit{nth} : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathit{int} \to \alpha$$

that given a list and a position returns the element at the position:

```
let rec nth l n =
  match l with
  | [] -> failwith "nth"
  | x :: l -> if n < 1 then x else nth l (n-1)
```

The function raises an exception if $l = []$ or $n \geq \mathit{length}\ l$. Note that *nth l 0* yields the head of l if l is nonempty. Also note that *nth* terminates since it recurses on the list argument. Here is a trace:

$$\mathit{nth}\ [0, 2, 3, 5]\ 2 \ = \ \mathit{nth}\ [2, 3, 5]\ 1 \ = \ \mathit{nth}\ [3, 5]\ 0 \ = \ 3$$

Exercise 2.7.1 What is the result of $nth\ [0,2,3,5]\ (-2)$?

Exercise 2.7.2 Declare a function *nth_checked* that raises an invalid-argument exception (see §1.13) if $n < 0$ and otherwise agrees with *nth*. Check that your function behaves the same as the predefined function *List.nth*.

Exercise 2.7.3 Define a function $pos : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathbb{Z} \to \mathbb{B}$ testing whether a number is a position of a list.

Exercise 2.7.4 Declare a function $find : \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathbb{N}$ that returns the first position of a list a given value appears at. For instance, we want $find\ 1\ [3,1,1] = 1$. If the value doesn't appear in the list, a failure exception should be raised.

2.8 Option Types

The lookup function for lists

$$nth : \ \forall\alpha.\ \mathcal{L}(\alpha) \to int \to \alpha$$

raises an exception if the position argument is not valid for the given list. There is the possibility to avoid the exception by changing the result type of *nth* to an option type

$$nth\_opt : \ \forall\alpha.\ \mathcal{L}(\alpha) \to int \to \mathcal{O}(\alpha)$$

that in addition to the values of α has an extra value None that can be used to signal that the given position is not valid. In fact, OCaml comes with a type family

$$\mathcal{O}(\alpha) \ ::= \ \mathsf{None} \ | \ \mathsf{Some}\ \alpha$$

whose values are obtained with two polymorphic constructors

$$\begin{aligned} \mathsf{Some} &: \ \forall\alpha.\ \alpha \to \mathcal{O}(\alpha) \\ \mathsf{None} &: \ \forall\alpha.\ \mathcal{O}(\alpha) \end{aligned}$$

such that Some injects the values of a type t into $\mathcal{O}(t)$ and None represents the extra value. We can declare a lookup function returning options as follows:

```
let rec nth_opt l n =
  match l with
  | [] -> None
  | x :: l -> if n < 1 then Some x else nth_opt l (n-1)
```

An option may be seen as a list that is either empty or a singleton list $[x]$. In fact, it is possible to replace option types with list types. Doing this gives away information as it comes to type checking.

Exercise 2.8.1 Declare a function *nth_opt_checked* that raises an invalid-argument exception if $n < 0$ and otherwise agrees with *nth_opt*.

Exercise 2.8.2 Declare a function

$$nth\_list : \ \forall\alpha.\ \mathcal{L}(\alpha) \to int \to \mathcal{L}(\alpha)$$

that agrees with *nth_opt* but returns a list with at most one element.

Exercise 2.8.3 Declare a function $find\_opt : \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathcal{O}(\mathbb{N})$ that returns the first position of a list a given value appears at. For instance, we want $find\_opt\ 7\ [3,7,7] = Some\ 1$ and $find\_opt\ 2\ [3,7,7] = None$.

2.9 Generalized Match Expressions

OCaml provides pattern matching in more general form than the basic list matches we have seen so far. A good example for explaining **generalized match expressions** is a function

$$eq : \ \forall\alpha.\ (\alpha \to \alpha \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B}$$

testing equality of lists using an equality test for the base type given as argument:

```
let rec eq (p: 'a -> 'a -> bool) l1 l2 =
  match l1, l2 with
  | [], [] -> true
  | x::l1, y::l2 -> p x y && eq p l1 l2
  | _, _ -> false
```

The generalized match expression in the declaration matches on two values and uses a final **catch-all rule**. Evaluation of a generalized match tries the patterns of the rules in the order they are given and commits to the first rule whose pattern matches. The pattern of the final rule will always match and will thus be used if no other rule applies. One speaks of a *catch all rule*.

Note that the above declaration specifies the type of the argument p using a type variable α. Without this specification OCaml would infer the more general type $\forall\alpha\beta.\ (\alpha \to \beta \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathcal{L}(\beta) \to \mathbb{B}$ for *eq*.

The generalized match in the above declaration translates to a simple match with nested simple matches:

```
match l1 with
| [] ->
  begin match l2 with
    | [] -> true
    | _ :: _ -> false
    end
| x::l1 ->
  begin match l2 with
    | [] -> false
    | y::l2 -> p x y && eq p l1 l2
  end
```

The keywords `begin` and `end` provide a notational variant for a pair `( ··· )` of parentheses.

The notions of *disjointness* and *exhaustiveness* established for defining equations in §1.6 carry over to generalized match expressions. We will only use exhaustive match expressions but occasionally use non-disjoint match expressions where the order of the rules matters (e.g., catch-all rules). We remark that simple match expressions are always disjoint and exhaustive.

We see generalized match expressions as derived forms that compile into simple match expressions.

OCaml also has match expressions for tuples. For instance,

```
let fst a =
  match a with
  | (x, _) -> x
```

declares a projection function $fst : \forall\alpha\beta.\ \alpha \times \beta \to \alpha$ for pairs. In fact, match expressions for tuples are native in OCaml and the uses of tuple patterns we have seen in let expressions, lambda abstractions, and declarations all compile into match expressions for tuples.

Patterns in OCaml may also contain numbers and other constants. For instance, we may declare a function that tests whether a list starts with the numbers 1 and 2 as follows:

```
let test l =
  match l with
  | 1 :: 2 :: _ -> true
  | _ -> false
```

Exercise 2.9.1 Declare a function testing whether a list starts with the numbers 1 and 2 just using simple match expressions for lists.

Exercise 2.9.2 Declare a function $swap : \forall\alpha\beta.\ \alpha \times \beta \to \beta \times \alpha$ swapping the components of a pair using a simple match expression for tuples.

Exercise 2.9.3 (Maximal element) Declare a function that yields the maximal element of a list of numbers. If the list is empty, a failure exception should be raised.

Exercise 2.9.4 Translate the expression

```
fun l -> match l with
  | 0::x::_-> Some x
  | x::1::_ -> Some x
  | _ -> None
```

into an expression only using simple matches.

2.10 Sublists

A **sublist** of a list l is obtained by deleting $n \geq 0$ positions of l. For instance, the sublists of $[1, 2]$ are the lists

$$[1, 2], \ [2], \ [1], \ []$$

We observe that the empty list $[]$ has only itself as a sublist, and that a sublist of a nonempty list $x :: l$ is either a sublist of l, or a list $x :: l'$ where l' is a sublist of l. Using this observation, it is straightforward to define a function that yields a list of all sublists of a list:

$$\begin{aligned}
pow :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\mathcal{L}(\alpha)) \\
pow\ [] \ &:= \ [[]] \\
pow\ (x :: l) \ &:= \ pow\ l \ @\ map\ (\lambda\, l.\, x :: l)\ (pow\ l)
\end{aligned}$$

We call $pow\ l$ the **power list** of l.

A sublist test "l_1 is sublist of l_2" needs a more involved case analysis:

$$\begin{aligned}
is_sublist :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B} \\
is_sublist\ l_1\ [] \ &:= \ l_1 = [] \\
is_sublist\ []\ (y :: l_2) \ &:= \ \mathsf{true} \\
is_sublist\ (x :: l_1)\ (y :: l_2) \ &:= \ is_sublist\ (x :: l_1)\ l_2 \ \texttt{||} \\
& \qquad x = y \ \texttt{\&\&}\ is_sublist\ l_1\ l_2
\end{aligned}$$

We remark that a computationally naive sublist test can be obtained with the power list function and the membership test:

$$is\_sublist\ l_1\ l_2 \ = \ mem\ l_1\ (pow\ l_2)$$

Exercise 2.10.1 (Graded power list)
Declare a function $gpow : \forall\alpha.\ \mathbb{N} \to \mathcal{L}(\alpha) \to \mathcal{L}(\mathcal{L}(\alpha))$ such that $gpow\ k\ l$ yields a list containing all sublists of l of length k.

Exercise 2.10.2 (Prefixes, Segments, Suffixes)
Given a list $l = l_1 \mathbin{@} l_2 \mathbin{@} l_3$, we call l_1 a **prefix**, l_2 a **segment**, and l_3 a **suffix** of l. The definition is such that prefixes are segments starting at the beginning of a list, and suffixes are segments ending at the end of a list. Moreover, every list is a prefix, segment, and suffix of itself.

a) Convince yourself that segments are sublists.

b) Give a list and a sublist that is not a segment of the list.

c) Declare a function that yields a list containing all prefixes of a list.

d) Declare a function that yields a list containing all suffixes of a list.

e) Declare a function that yields a list containing all segments of a list.

Exercise 2.10.3 (Splits) Given a list $l = l_1 \mathbin{@} l_2$, we call the pair (l_1, l_2) a **split** of l. Declare a function that yields a list containing all splits of a list.

Exercise 2.10.4 Declare a function $\mathit{filter} : \forall\alpha.\ (\alpha \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ that given a test and a list yields the sublist of all elements that pass the test. For instance, we want $\mathit{filter}\ (\lambda x.\ x > 2)\ [2, 5, 1, 5, 2] = [5, 5]$.

2.11 Folding Lists

Consider the list

$$a_1 :: (a_2 :: (a_3 :: []))$$

If we replace cons with $+$ and nil with 0, we obtain the expression

$$a_1 + (a_2 + (a_3 + 0))$$

which evaluates to the sum of the elements of the list. If we replace cons with $\cdot$ and nil with 1, we obtain the expression

$$a_1 \cdot (a_2 \cdot (a_3 \cdot 1))$$

which evaluates to the product of the elements of the list. More generally, we obtain the expression

$$f\, a_1\, (f\, a_2\, (f\, a_3\ b))$$

if we replace cons with a function f and nil with a value b. Even more generally, we can define a function *fold* such that $\mathit{fold}\, f\, l\, b$ yields the value of the expression obtained from l by replacing cons with f and nil with b:

$$\begin{aligned} \mathit{fold} :\ & \forall\alpha\beta.\ (\alpha \to \beta \to \beta) \to \mathcal{L}(\alpha) \to \beta \to \beta \\ \mathit{fold}\ f\ []\ b \ &:=\ b \\ \mathit{fold}\ f\ (a :: l)\ b \ &:=\ f\ a\ (\mathit{fold}\ f\ l\ b) \end{aligned}$$

We have the following equations:

$$\begin{aligned}
\mathit{fold}\ (+)\ [x_1,\ldots,x_n]\ 0 &= x_1+\cdots+x_n+0\\
\mathit{fold}\ (\lambda ab.\ a^2+b)\ [x_1,\ldots,x_n]\ 0 &= x_1^2+\cdots+x_n^2+0\\
l_1 @ l_2 &= \mathit{fold}\ (::)\ l_1\ l_2\\
\mathit{flatten}\ l &= \mathit{fold}\ (@)\ l\ []\\
\mathit{length}\ l &= \mathit{fold}\ (\lambda ab.\ b+1)\ l\ 0\\
\mathit{rev}\ l &= \mathit{fold}\ (\lambda ab.\ b @ [a])\ l\ []
\end{aligned}$$

Folding of lists is similar to iteration with numbers in that both recursion schemes can express many functions without further recursion.

There is a tail-recursive variant *foldl* of *fold* satisfying the equation $\mathit{foldl}\ f\ l\ b = \mathit{fold}\ f\,(\mathit{rev}\ l)\ b$:

$$\begin{aligned}
&\mathit{foldl} :\ \forall\alpha\beta.\ (\alpha\to\beta\to\beta)\to\mathcal{L}(\alpha)\to\beta\to\beta\\
&\quad \mathit{foldl}\ f\ []\ b \;:=\; b\\
&\mathit{foldl}\ f\ (a :: l)\ b \;:=\; \mathit{foldl}\ f\ l\ (f\ a\ b))
\end{aligned}$$

One says that *fold* folds a list from the right

$$\mathit{fold}\ f\ [a_1,a_2,a_3]\ b \;=\; f\ a_1\ (f\ a_2\ (f\ a_3\ b))$$

that *foldl* folds a list from the left

$$\mathit{foldl}\ f\ [a_1,a_2,a_3]\ b \;=\; f\ a_3\ (f\ a_2\ (f\ a_1\ b))$$

If the order of the folding is not relevant, the tail-recursive version *foldl* is preferable over *fold*.

OCaml provides the function *fold* as *List.fold_right*. OCaml also provides a function *List.fold_left*, which however varies the argument order of our function *foldl*. To avoid confusion, we will not use *List.fold_left* in this chapter but instead use our function *foldl*.

Exercise 2.11.1 Using *fold*, declare functions that yield the concatenation, the flattening, the length, and the reversal of lists.

Exercise 2.11.2 Using *foldl*, declare functions that yield the length, the reversal, and the concatenation of lists.

Exercise 2.11.3 We have the equations

$$\begin{aligned}
\mathit{foldl}\ f\ l\ b &= \mathit{fold}\ f\,(\mathit{rev}\ l)\ b\\
\mathit{fold}\ f\ l\ b &= \mathit{foldl}\ f\,(\mathit{rev}\ l)\ b
\end{aligned}$$

a) Show that the second equation follows from the first equation using the equation $\mathit{rev}\,(\mathit{rev}\ l) = l$.
b) Obtain *fold* from *foldl* not using recursion.
c) Obtain *foldl* from *fold* not using recursion.

2.12 Insertion Sort

A sequence $x_1, \dots, x_n$ of numbers is called **sorted** if its elements appear in order: $x_1 \leq \cdots \leq x_n$. **Sorting** a sequence means to rearrange the elements such that the sequence becomes sorted. We want to define a function

$$sort : \; \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$$

such that *sort* l is a sorted rearrangement of l. For instance, we want $\mathit{sort}\,[5,3,2,7,2] = [2,2,3,5,7]$.

For now, we only consider sorting for lists of numbers. Later it will be easy to generalize to other types and other orders.

There are different sorting algorithms. Probably the easiest one is **insertion sort**. For insertion sort one first defines a function that inserts a number x into a list such that the result list is sorted if the argument list is sorted.[2] Now sorting a list l is easy: We start with the empty list, which is sorted, and insert the elements of l one by one. Once all elements are inserted, we have a sorted rearrangement of l. Here are definitions of the necessary functions:

$$\begin{aligned}
&\mathit{insert} : \; \mathbb{Z} \to \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z}) \\
&\qquad \mathit{insert}\ x\ [\,] \;:=\; [x] \\
&\mathit{insert}\ x\ (y :: l) \;:=\; \text{IF } x \leq y \text{ THEN } x :: y :: l \text{ ELSE } y :: \mathit{insert}\ x\ l
\end{aligned}$$

$$\begin{aligned}
&\mathit{isort} : \; \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z}) \\
&\qquad \mathit{isort}\ [\,] \;:=\; [\,] \\
&\mathit{isort}\ (x :: l) \;:=\; \mathit{insert}\ x\ (\mathit{isort}\ l)
\end{aligned}$$

Make sure you understand every detail of the definition. We offer a trace:

$$\begin{aligned}
\mathit{isort}\ [3,2] &= \mathit{insert}\ 3\ (\mathit{isort}\ [2]) \\
&= \mathit{insert}\ 3\ (\mathit{insert}\ 2\ (\mathit{isort}\ [\,])) \\
&= \mathit{insert}\ 3\ (\mathit{insert}\ 2\ [\,]) \\
&= \mathit{insert}\ 3\ [2] \;=\; 2 :: \mathit{insert}\ 3\ [\,] \;=\; 2 :: [3] \;=\; [2,3]
\end{aligned}$$

Note that *isort* inserts the elements of the input list reversing the order they appear in the input list.

[2] More elegantly, we may say that the insertion function preserves sortedness.

Comparisons are polymorphically typed

Declaring the functions *insert* and *isort* in OCaml is now routine. There is, however, the surprise that OCaml derives polymorphic types for *insert* and *isort* if no type specification is given:

$$\begin{aligned} insert :\ & \forall\alpha.\ \alpha \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\ isort :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \end{aligned}$$

This follows from the fact that OCaml accommodates comparisons with the polymorphic type

$$\forall\alpha.\ \alpha \to \alpha \to \mathbb{B}$$

it also uses for the equality test (§1.18). We will explain later how comparisons behave on tuples and lists. For boolean values, OCaml realizes the order $\mathsf{false} < \mathsf{true}$. Thus we have

$$isort\ [\mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}] \ = \ [\mathsf{false}, \mathsf{false}, \mathsf{true}, \mathsf{true}]$$

Exercise 2.12.1 Declare a function $sorted : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathbb{B}$ that tests whether a list is sorted. Use tail recursion. Write the function with a generalized match and show how the generalized match translates into simple matches.

Exercise 2.12.2 Declare a function $perm : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B}$ that tests whether two lists are equal up to reordering.

Exercise 2.12.3 (Sorting into descending order)
Declare a function $sort\_desc : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ that reorders a list such that the elements appear in descending order. For instance, we want $sort\_desc\,[5,3,2,5,2,3] = [5,5,3,3,2,2]$.

Exercise 2.12.4 (Sorting with duplicate deletion)
Declare a function $dsort : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)$ that sorts a list and removes all duplicates. For instance, $dsort\,[5,3,2,5,2,3] = [2,3,5]$.

Insertion order

Sorting by insertion inserts the elements of the input list one by one into the empty list. The order in which this is done does not matter for the result. The function *isort* defined above inserts the elements of the input list reversing the order of the input list. If we define *isort* as

$$isort\ l \ := \ fold\ insert\ l\ []$$

we preserve the insertion order. If we switch to the definition

$$isort\ l\ :=\ foldl\ insert\ l\ []$$

we obtain a tail-recursive insertion function inserting the elements of the input list in the order they appear in the input list.

Exercise 2.12.5 (Count Tables) Declare a function

$$table:\ \forall\alpha.\ \mathcal{L}(\alpha)\to\mathcal{L}(\alpha\times\mathbb{N}^+)$$

such that $(x,n)\in table\ l$ if and only if x occurs $n>0$ times in l. For instance, we want

$$table\ [4,2,3,2,4,4]\ =\ [(4,3),(2,2),(3,1)]$$

Make sure *table* lists the count pairs for the elements of l in the order the elements appear in l, as in the example above.

2.13 Generalized Insertion Sort

Rather than sorting lists using the predefined order $\leq$, we may sort lists using an order given as argument:

$$\begin{aligned}
&insert:\ \forall\alpha.\ (\alpha\to\alpha\to\mathbb{B})\to\alpha\to\mathcal{L}(\alpha)\to\mathcal{L}(\alpha)\\
&\qquad insert\ p\ x\ []\ :=\ [x]\\
&insert\ p\ x\ (y::l)\ :=\ \text{IF}\ \ p\,x\,y\ \ \text{THEN}\ x::y::l\ \ \text{ELSE}\ \ y::insert\ p\ x\ l
\end{aligned}$$

$$\begin{aligned}
&gisort:\ \forall\alpha.\ (\alpha\to\alpha\to\mathbb{B})\to\mathcal{L}(\alpha)\to\mathcal{L}(\alpha)\\
&gisort\ p\ l\ :=\ fold\ (insert\ p)\ l\ []
\end{aligned}$$

Now the function $gisort\,(\leq)$ sorts as before in ascending order, while the function $gisort\,(\geq)$ sorts in descending order:

$$gisort\ (\geq)\ [1,3,3,2,4]\ =\ [4,3,3,2,1]$$

When we declare the functions *insert* and *gisort* in OCaml, we can follow the mathematical definitions. Alternatively, we can declare *gisort* using a local declaration for *insert*:

```
let gisort p l =
  let rec insert x l =
    match l with
    | [] -> [x]
    | y :: l -> if p x y then x :: y :: l else y :: insert x l
  in
  foldl insert l []
```

This way we avoid the forwarding of the argument p.

Exercise 2.13.1 Declare a function

$$reorder : \forall\alpha\beta.\ \mathcal{L}(\alpha \times \beta) \to \mathcal{L}(\alpha \times \beta)$$

that reorders a list of pairs such that the first components of the pairs are ascending. If there are several pairs with the same first component, the original order of the pairs should be preserved. For instance, we want $reorder\,[(5,3),(3,7),(5,2),(3,2)] = [(3,7),(3,2),(5,3),(5,2)]$. Declare *reorder* as a one-liner using the sorting function *gisort*.

2.14 Lexicographic Order

We now explain how we obtain an order for lists over t from an order for the base type t following the principle used for ordering words in dictionaries. We speak of a **lexicographic ordering**. Examples for the lexicographic ordering of lists of integers are

$$[] < [-1] < [-1,-2] < [0] < [0,0] < [0,1] < [1]$$

The general principle behind the lexicographic ordering can be formulated with two rules:

- $[] < x :: l$
- $x_1 :: l_1 < x_2 :: l_2$ if either $x_1 < x_2$, or $x_1 = x_2$ and $l_1 < l_2$.

Following the rules, we define a function that yields a test for the lexicographic order $\leq$ of lists given a test for an order $\leq$ of the base type:

$$\begin{aligned}
&lex : \ \forall\alpha.\ (\alpha \to \alpha \to \mathbb{B}) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B}\\
&\qquad\qquad lex\ p\ []\ l_2 \ := \ \mathsf{true}\\
&\qquad lex\ p\ (x_1 :: l_1)\ []\ := \ \mathsf{false}\\
&lex\ p\ (x_1 :: l_1)\ (x_2 :: l_2) \ := \ p\,x_1\,x_2\ \texttt{\&\&}\\
&\qquad\qquad\qquad\qquad \text{IF } p\,x_2\,x_1 \text{ THEN } lex\ p\ l_1\ l_2 \text{ ELSE } \mathsf{true}
\end{aligned}$$

Note that the predicate p must be **reflexive**, i. e. $p\,x\,x = \mathsf{true}$ because we test for equality by $p\,x_1\,x_2$ && $p\,x_2\,x_1$.

Often, comparison functions like p are implemented by returning an integer that encodes the comparison result in the following way:

$$\begin{aligned}
cmp\,x\,y < -1 &\quad \text{if } x < y\\
cmp\,x\,y = 0 &\quad \text{if } x = y\\
cmp\,x\,y > 1 &\quad \text{if } x > y
\end{aligned}$$

The advantage of such an implementation is, that one can keep strict less/greater than and equality apart which is more informative. Using such a predicate *cmp*, the condition in the definition of *lex* above can be phrase a bit more concisely:

```
let rec lex cmp l l' = match l, l' with
  | [], [] -> 0
  | [], _  -> -1
  | _,  [] -> 1
  | x :: l, y :: l' -> let c = cmp x y in
                       if c <> 0 then c else lex l l'
```

Exercise 2.14.1 (Lexicographic order for pairs) The idea of lexicographic order extends to pairs and to tuples in general.

a) Explain the lexicographic order of pairs of type $t_1 \times t_2$ given orders for the component types t_1 and t_2.

b) Declare a function

$$lexP : \ \forall \alpha\beta.\ (\alpha \to \alpha \to \mathbb{B}) \to (\beta \to \beta \to \mathbb{B}) \to \alpha \times \beta \to \alpha \times \beta \to \mathbb{B}$$

testing the lexicographic order of pairs. For instance, we want

$$lexP\ (\leq)\ (\geq)\ (1,2)\ (1,3) = \mathsf{false}$$

and $lexP\ (\leq)\ (\geq)\ (0,2)\ (1,3) = \mathsf{true}$.

2.15 Prime Factorization

Every integer greater than 1 can be written as a product of prime numbers; for instance,

$$\begin{aligned} 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ 147 &= 3 \cdot 7 \cdot 7 \\ 735 &= 3 \cdot 5 \cdot 7 \cdot 7 \end{aligned}$$

One speaks of the *prime factorization* of a number. Recall that a prime number is an integer greater than 1 that cannot be obtained as the product of two integers greater than 1 (§1.17).

It is straightforward to compute the smallest prime factor of a number $x \geq 2$: We simply search for the first $k \geq 2$ **dividing** x (i.e., $x \,\%\, k = 0$). Such a k exists since x divides x. Moreover, the first k dividing x is always prime since otherwise there would be a smaller number greater than 1 dividing k and thus also x.

If we can compute smallest prime factors, we can compute prime factorizations by dividing with the prime factor found and continuing recursively. We realize this algorithm with an OCaml function

$$prime\_fac : \ int \to \mathcal{L}(int)$$

declared as follows:

```
let rec prime_fac x =
  if x < 2 then []
  else let k = first (fun k -> x mod k = 0) 2 in
    k :: prime_fac (x / k)
```

We have $prime\_fac\ 735 = [3, 5, 7, 7]$, for instance.

As is, the algorithm is slow on large prime numbers (try 479,001,599). The algorithm can be made much faster by stopping the linear search for the first k dividing x once $k^2 > x$ since then the first k dividing x is x. Moreover, if we have a least prime factor $k < x$, it suffices to start the search for the next prime factor at k since we know that no number smaller than k divides x. We realize the optimized algorithm as follows:

```
let rec prime_fac' k x =
  if k * k > x then [x]
  else if x mod k = 0 then k :: prime_fac' k (x / k)
  else prime_fac' (k + 1) x
```

We have $prime\_fac'\ 2\ \ 735 = [3, 5, 7, 7]$, for instance. Interestingly, the optimized algorithm is simpler than the naive algorithm we started from (as it comes to code, but not as it comes to correctness). It makes sense to define a wrapper function for $prime\_fac'$ ensuring that $prime\_fac'$ is applied with admissible arguments:

```
let prime_fac x = if x < 2 then [] else prime_fac' 2 x
```

We used several mathematical facts to derive the optimized prime factorization algorithm:

1. If $2 \leq x < k^2$ and no number $2 \leq n \leq k$ divides x, then x is a prime number.
2. If $2 \leq k < x$, and k divides x, and no number $2 \leq n < k$ divides x, then k is the least prime factor of x.
3. If $2 \leq k < x$, and k divides x, and no number $2 \leq n < k$ divides x, then no number $2 \leq n < k$ divides x/k.

The correctness of the algorithm also relies on the fact that the **safety condition**

- $2 \le k \le x$ and no number $2 \le n < k$ divides x

propagates from every initial application of $prime\_fac'$ to all recursive applications. We say that the safety condition is an **invariant** for the applications of $prime\_fac'$.

It suffices to argue the termination of $prime\_fac'$ for the case that the safety condition is satisfied. In this case the $x - k \ge 0$ is decreased by every recursion step.

Exercise 2.15.1 Give traces for the following applications:

a) $prime\_fac'$ 2 7   b) $prime\_fac'$ 2 8   c) $prime\_fac'$ 2 15

Exercise 2.15.2 Declare a function that yields the least prime factor of an integer $x \ge 2$. Make sure that at most $\sqrt[2]{x}$ remainder operations are necessary.

Exercise 2.15.3 Declare a primality test using at most $\sqrt[2]{x}$ remainder operations for an argument $x \ge 2$.

Exercise 2.15.4 Dieter Schlau has simplified the naive prime factorization function:

```
let rec prime_fac x =
  if x < 2 then []
  else let k = first (fun k -> x mod k = 0) 2 in
    k :: prime_fac (x / k)
```

Explain why Dieter's function is correct.

2.16 Key-Value Maps

One way of representing a function is by tabulation. For each preimage (key) we store the image (value). That is often called a key-value map. One way to implement a key-value map is to use a list of key-value pairs of the form:

$$(key,\ value)$$

For example, a mapping from identifiers to values

$$[\texttt{"x"} \mapsto 5,\ \texttt{"y"} \mapsto 7,\ \texttt{"z"} \mapsto 2]$$

can be represented by the following list:

$$[(\texttt{"x"}, 5),\ (\texttt{"y"}, 7),\ (\texttt{"z"}, 2)] : \mathcal{L}(string \times int)$$

Following this consideration, we now define **maps** as values of the type family

$$map\ \alpha\ \beta \ := \ \mathcal{L}(\alpha \times \beta)$$

The two most important functions on maps are

$$\begin{aligned} lookup &: \ \forall \alpha\beta.\ map\ \alpha\ \beta \to \alpha \to \mathcal{O}(\beta) \\ update &: \ \forall \alpha\beta.\ map\ \alpha\ \beta \to \alpha \to \beta \to map\ \alpha\ \beta \end{aligned}$$

where *lookup* yields the value for a given key provided the map contains a pair for the key, and *update* updates the map with a given key-value pair. For instance,

$$\begin{aligned} &lookup\ [(\text{"x"}, 5),\ (\text{"y"}, 13),\ (\text{"z"}, 2)]\ \text{"y"} \ = \ 13 \\ &update\ [(\text{"x"}, 5),\ (\text{"y"}, 7),\ (\text{"z"}, 2)]\ \text{"y"}\ 13 \ = \ [(\text{"x"}, 5),\ (\text{"y"}, 13),\ (\text{"z"}, 2)] \end{aligned}$$

The defining equations for *lookup* and *update* are as follows:

$$\begin{aligned} lookup\ []\ a &:= \ \mathsf{None} \\ lookup\ ((a', b) :: l)\ a &:= \ \text{IF } a' = a \text{ THEN } \mathsf{Some}\ b \\ &\ \ \text{ELSE } lookup\ l\ a \end{aligned}$$

$$\begin{aligned} update\ []\ a\ b &:= \ [(a, b)] \\ update\ ((a', b') :: l)\ a\ b &:= \ \text{IF } a' = a \text{ THEN } (a, b) :: l \\ &\ \ \text{ELSE } (a', b') :: update\ l\ a\ b \end{aligned}$$

Note that OCaml provides support for key-value maps in its List module and calls them **association lists**.

Exercise 2.16.1 Give the values of the following expressions:

a) $update\ (update\ (update\ []\ \text{"x"}\ 7)\ \text{"y"}\ 2)\ \text{"z"}\ 5$

b) $lookup\ (update\ l\ \text{"x"}\ 13)\ \text{"x"}$

c) $lookup\ (update\ l\ a\ 7)\ a$

Exercise 2.16.2 Decide for each of the following equations whether it is true in general.

a) $lookup\ (update\ l\ a\ b)\ a = \mathsf{Some}\ \ b$

b) $lookup\ (update\ l\ a'\ b)\ a = lookup\ l\ a \quad \text{if } a' \neq a$

c) $update\ (update\ l\ a\ b)\ a'\ b' = update\ (update\ l\ a'\ b')\ a\ b \quad \text{if } a \neq a'$

d) $lookup\ (update\ (update\ l\ a\ b)\ a'\ b')\ a = \mathsf{Some}\ b \quad \text{if } a \neq a'$

Exercise 2.16.3 (Boundedness) Declare a function

$$bound : \forall\alpha\beta.\ map\ \alpha\ \beta \to \alpha \to bool$$

that checks whether a map binds a given key. Note that you can define *bound* using *lookup*.

Exercise 2.16.4 (Deletion) Declare a function

$$delete : \forall\alpha\beta.\ map\ \alpha\ \beta \to \alpha \to map\ \alpha\ \beta$$

deleting the entry for a given key. We want $lookup\ (delete\ l\ a)\ a\ =\ \mathsf{None}$ for all environments l and all keys a.

Exercise 2.16.5 (Maps with memory) Note that *lookup* searches maps from left to right until it finds a pair with the given key. This opens up the possibility to keep previous values in the map by modifying *update* so that it simply appends the new key-value pair in front of the list:

$$update\ l\ a\ b\ \ :=\ \ (a, b) :: l$$

Redo the previous exercises for the new definition of *update*. Also define a function

$$lookup\_all : \forall\alpha\beta.\ map\ \alpha\ \beta \to \alpha \to \mathcal{L}(\beta)$$

that yields the list of all values for a given key.

Exercise 2.16.6 (Maps as functions) Maps can be realized with functions if all maps are constructed from the empty map

$$empty : \forall\alpha\beta.\ map\ \alpha\ \beta$$

with *update* and the only thing that matters is that *lookup* yields the correct results. Assume the definition

$$map\ \alpha\ \beta\ \ :=\ \ \alpha \to \mathsf{Some}\ \beta$$

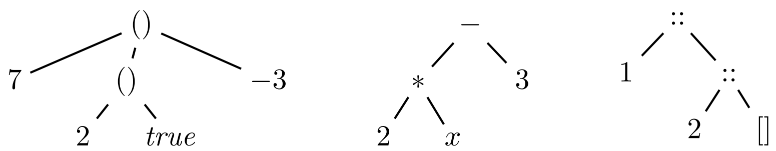
and define *empty* and the operations *update* and *lookup* accordingly. Test your solution with

$$lookup\ (update\ (update\ (update\ empty\ \text{"y"}\ 7)\ \text{"x"}\ 2)\ \text{"y"}\ 5)\ \text{"y"}\ \ =\ \ \mathsf{Some}\ 5$$

Note that you can still define the operations *bound* and *delete* from Exercises 2.16.1 and 2.16.2.

3 Constructor Types and Trees

Lists as we have introduced them in the last chapter are just one instance of a more general concept: **constructor types**[1]. We start the chapter with a general introduction to constructor types. We will see that constructor types basically model **tree** structures. Trees are a very important concept in programming because many important data structures are based on trees. These include nested tuples, expressions and lists:



3.1 Constructor Types

A **constructor type** is a type that is defined by a set of constructors. Each constructor has a name and zero or more parameters. Like functions, constructors have a functional type: they take arguments and return an instance of the data type the constructor belongs to. However, constructors are very different from functions and this has important consequences. First, constructors do not have a body. When a constructor is applied to its arguments, the result is a value of the constructor type that the constructor belongs to. There is no further computation going on. This also entails that constructors cannot be partially applied: all of their arguments must be given. Second, constructors are no values: they cannot be bound to variables, nor can they be passed to functions as arguments nor can they be returned by functions.

Let us start with a simple example: the type `bool` has two constructors: `true` and `false`:

```
type bool = false | true
```

While `bool` is part of the OCaml standard library, we can define our own constructor types such as

[1]Constructor types are often also called algebraic data types (ADTs). Badly enough, the abbreviation ADT is also used for abstract data types which is a different concept.

```
type weekday =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday
```

It is a convention that the constructors of user-defined constructor types start with a capital letter. A constructor type in which each constructor has no parameters, like the type `weekday`, is called an **enumeration type**. Note that the vertical bar in front of the first constructor is optional.

Of course, we can write functions that operate on constructor types:

```
let day_of_week w = match w with
  | Monday -> 1
  | Tuesday -> 2
  | Wednesday -> 3
  | Thursday -> 4
  | Friday -> 5
  | Saturday -> 6
  | Sunday -> 7
```

which has the type $weekday \to int$.

Each constructor of a constructor type defines a **variant**. Using the **match** expression, we can perform a case distinction on the variant of a value of a constructor type. OCaml will print a warning if we forget a case. As mentioned in the previous section, matches can take shortcuts using catch-all rules:

```
let is_weekend w = match w with
  | Saturday -> true
  | Sunday -> true
  | _ -> false
```

OCaml also offers a shortcut for functions that are defined by a match expression:

```
let is_weekend = function
  | Saturday -> true
  | Sunday -> true
  | _ -> false
```

Here, the `function` keyword defines an implicit match expression with a single argument. And even shorter, cases with the same outcome can be agglomerated:

```
let is_weekend = function Saturday | Sunday -> true | _ -> false
```

Another example for an enumeration type is

```
type comparison = LT | EQ | GT
```

which encodes the result of a comparison. We can use this type to make the lexicographic comparison function of Section 2.14 even more readable:

```
let rec lex cmp l l' = match l, l' with
  | [], [] -> EQ
  | [], _  -> LT
  | _,  [] -> GT
  | x :: l, y :: l' -> let c = cmp x y in
                       if c <> EQ then c else lex l l'
```

Constructors can also be defined to take arguments. Consider the following type that defines geometric shapes:

```
type shape =
  | Circle of float
  | Rectangle of float * float
  | Triangle of float * float * float
```

Whenever we want to construct a value of type `shape`, we have to specify which constructor we want to use and pass the appropriate arguments. For example:

```
let c = Circle 1.0
let r = Rectangle (1.0, 2.0)
let t = Triangle (1.0, 2.0, 3.0)
```

The match expression also extends to constructors with arguments:

```
let area = function
  | Circle r -> Float.pi *. r *. r
  | Rectangle (w, h) -> w *. h
  | Triangle (a, b, c) ->
      let s = (a +. b +. c) /. 2.0 in
      sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
```

Note that a branch of a match expression that corresponds to a constructor with arguments will introduce new variables that are bound to the arguments of the constructor and are visible in the right-hand side of the branch.

Constructor types can also be defined **recursively**. For example, the Peano definition of the natural numbers can be directly captured using a constructor type:

```
type nat = O | S nat
```

This definition captures the informal statements: “0 is a natural number. Every natural number n has a successor $S\ n$ that is also a natural number.” It also corresponds to a **unary number representation**: The number is represented by nesting depth of the constructor S. Note that our decimal system of writing down numbers is “just an optimization” because it allows us to write down a number n with $\lceil \log_{10}(n+1) \rceil$ digits instead of n nested applications of the constructor S.

```
let eleven  : int = 11
let eleven' : nat = S (S (S (S (S (S (S (S (S (S (S O))))))))))
```

Simple operations such as an equality test can be defined on `nat` using recursion:

```
let rec eq n m = match n, m with
  | O, O -> true
  | O, S _ -> false
  | S _, O -> false
  | S n, S m -> eq n m
```

Another example for a recursive constructor type is the **structure** of simple arithmetic expressions:

```
type exp =
    | C of int            (* constant *)
    | V of string         (* variable *)
    | A of exp * exp      (* addition *)
    | M of exp * exp      (* multiplication *)
```

The following expressions are values of type `exp`:

```
let a = A (C 1, C 2)
let b = A (C 1, V "x")
let c = M (V "x", V "x")
let d = A (M (C 2, V "x"), C 1)
```

The following function evaluates expressions with a given **environment**, i. e. a key-value map from variables to values which we implement according to Section 2.16.

```
type env = (string * int) list
let rec eval env = function
  | C i -> i
  | V x -> lookup env x
  | A (x, y) -> (eval env x) + (eval env y)
  | M (x, y) -> (eval env x) * (eval env y)
```

Note that the function `eval` is recursive and recursion follows the **structure** of the constructor type. We will later see that we can use constructor types to represent the entire syntax of a programming language.

Finally, constructor types also can have polymorphic constructors. We have seen two polymorphic constructor types already: The option type

```
type 'a option =
  | Some of 'a
  | None
```

and the list type which is in addition also recursive. It is defined equivalently to the following type but uses the special syntax `[]` for `Nil` and `::` for `Cons`.

```
type 'a my_list =
  | Nil
  | Cons of 'a * 'a list
```

Exercise 3.1.1 Implement a function `lt: nat -> nat -> bool` that checks whether a given `nat` is less than another.

Exercise 3.1.2 Adapt the definitions of add, mul, pow in Section 1.15 to work on the type `nat` and play with simple examples.

Exercise 3.1.3 Make yourself clear that `nat` is isomorphic to `unit list` where `unit` is the type that contains one single value `()`. Reimplement the functions of the previous exercises on that list type.

Exercise 3.1.4 Express the conditional expression `if b then e1 else e2` as a match expression.

Exercise 3.1.5 Write a function $derive : exp \to string \to exp$ that computes the derivative of an expression with respect to a variable.

Exercise 3.1.6 Write a function $vars : exp \to string\ list$ that creates a list of all variables used in an expression of type exp.

Exercise 3.1.7 Write a function that checks if a given expression is a sub-expression of another one.

Exercise 3.1.8 Write a function that counts the number of occurrences of a variable in an expression.

3.2 Rose Trees

We begin with a particularly simple class of trees, which we call **rose trees**. Rose trees are formed according to a recursive construction rule:

If t_0, …, t_{n-1} are rose trees, then the list $[t_0; \ldots; t_{n-1}]$ is a rose tree. Rose trees are therefore nested lists. The simplest rose tree is the empty list. Starting from the empty list, more complex rose trees can then be formed. In OCaml, we represent rose trees according to the following type declaration:

```
type tree = T of tree list
```

Here are examples of rose trees:

```
let t1 = T[]
let t2 = T[t1; t1; t1]
let t3 = T[T[t2]; t1; t2]
```

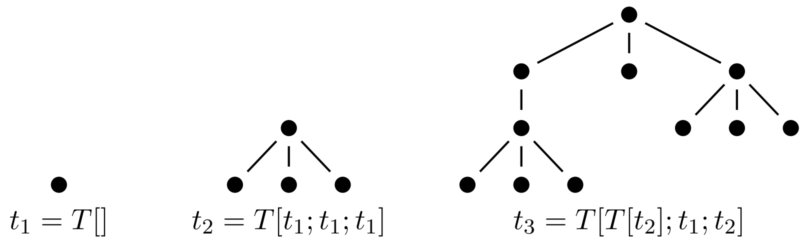
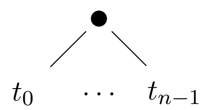


Figure 3.1: Graphical representation of rose trees.

Since we only consider rose trees (and no other types of trees) in this section, we will refer to them simply as trees in the following.

The graphical representation of trees is very helpful for understanding trees. The graphical representation of a tree $T[t_0; \ldots; t_{n-1}]$ is obtained by connecting the graphical representations of the **successor trees** t_0, …, t_{n-1} with a **node** • and n strokes, so-called **edges**:



The top node of the graphical representation of a tree is called the **root**. Nodes from which no edge leads to a lower node are called **leaves**. Nodes that are not leaves are called **inner nodes**.

As an example, consider the graphical representation of the tree t_3 in Fig. 3.1. It consists of 11 nodes and 10 edges. 7 of the nodes are leaves, the remaining 4 are inner nodes.

In general, the representation of a tree contains at least one node. Furthermore, the number of edges is always one less than the number of nodes, because we can assign to each node that is different from the root the edge pointing to it from above.

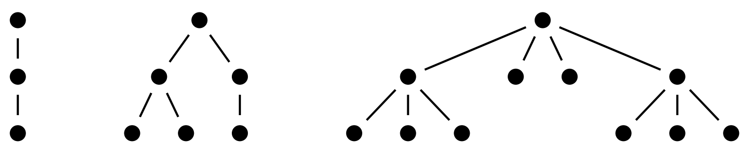
The tree $T[]$ is called **atomic**. All other trees are called **compound**. That means that there is exactly one atomic tree.

Let us briefly explain where the term tree comes from. If we rotate the graphical representation of a rose tree by 180 degrees, we get an image reminiscent of a branching natural tree. Incidentally, the concept of a family tree is based on the same pictorial idea.

Exercise 3.2.1 Declare a function `compound: tree -> bool` that tests whether a tree is compound.

Exercise 3.2.2 Draw the graphical representation of the tree $T[t_2; t_1; t_2]$.

Exercise 3.2.3 Specify expressions that describe the trees with the graphical representations shown below. You can use the identifiers t_1 and t_2 declared above.



Exercise 3.2.4 Let the graphical representation of a tree be given. Assume that the representation contains $n \geq 1$ edges and answer the following questions:

1. What is the minimum and the maximum number of nodes in the representation, respectively?
2. What is the minimum and the maximum number of leaves in the representation, respectively?
3. What is the minimum and the maximum number of inner nodes in the representation, respectively?

3.2.1 Successor Trees

Let $t = T[t_0; \dots; t_{n-1}]$ be a tree. We denote the number n as the **arity** of t and $t_0, \dots, t_{n-1}$ as the **successor trees** of t. Moreover, we denote t_k as the **k-th successor tree** of t (for $k \in \{0, \dots, n-1\}$). Here are functions that provide the arity and successor trees of a tree:

```
let arity (T ts) = List.length ts
let succtree (T ts) k = List.nth ts k
```

For a given tree t and a positive number k, the function `succtree` computes the k-th successor tree of t. The successor trees are numbered consecutively starting with 0 as stated above. If the tree has no k-th successor tree, the function application fails with an exception.

3.2.2 Shape of Arithmetic Expressions

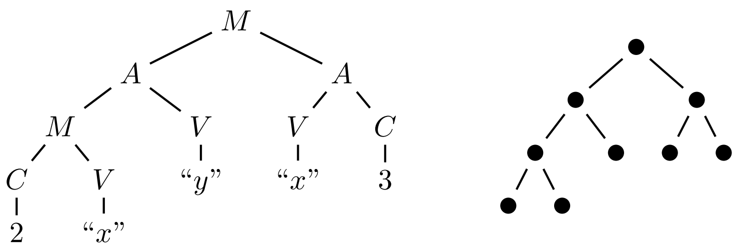
The arithmetic expressions from Section 3.1

```
type exp = C of int | V of string
         | A of exp * exp | M of exp * exp
```

can be understood as trees whose nodes are annotated with additional information. Here, the constructors `C` and `V` describe 0-ary trees and the constructors `A` and `M` describe 2-ary trees. We can make this connection between arithmetic expressions and rose trees precise by means of a function `shape: exp -> tree`, which computes the shape of an expression:

```
let rec shape e = match e with
  | C _ -> T[]
  | V _ -> T[]
  | A (e1, e2) -> T[shape e1; shape e2]
  | M (e1, e2) -> T[shape e1; shape e2]
```

Here is an example of an expression and its shape:



Exercise 3.2.5 Give the shape of the expression $(x + 3)(y + 7)$.

Exercise 3.2.6 Give an expression with the shape $T[T[t_1; t_1]; t_1]$.

3.2.3 Lexicographic Tree Ordering

The lexicographic ordering principle for lists (see Section 2.14), when applied recursively, yields an ordering for rose trees, which is called **lexicographic tree ordering**:

```
let rec compareTree (T ts1) (T ts2) = List.compare compareTree
    ts1 ts2
```

Exercise 3.2.7 Order the following trees according to the lexicographic ordering:

$$t_1 \qquad t_2 \qquad t_3 \qquad T[T[t_1]] \qquad T[t_1; T[T[t_1]]] \qquad T[T[T[T[t_1]]]]$$

Exercise 3.2.8 Declare a function equivalent to `compareTree` without using the function `List.compare`.

3.3 Subtrees

Since the successor trees of a tree may again be formed with successor trees, it is useful to speak of the **subtrees** of a tree. We define subtrees as follows:

1. If t is a tree, then t is a subtree of t.
2. If t' is a successor tree of a tree t, then each subtree of t' is a subtree of t.

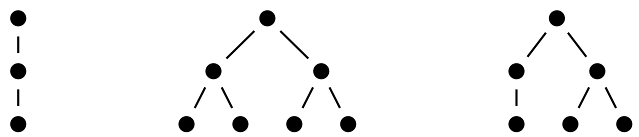
Here is a function that tests for a tree t and a tree t' whether t is a subtree of t':

```
let rec subtree t (T ts) = t = T ts || List.exists (subtree t)
    ts
```

We distinguish between subtrees and their **occurrences** in a tree. For example, the tree t_1 occurs three times as a subtree of t_2, and t_2 occurs twice as a subtree of t_3 (see Fig. 3.1). The occurrences of the subtrees of a tree correspond exactly to the nodes of its graphical representation. Thus, the ways of speaking for nodes carry over to the occurrences of subtrees. Here is a function that counts how often a tree occurs as a subtree in a tree:

```
let rec count t (T ts) = if t = T ts then 1 else foldl (+) (
    map (count t) ts) 0
```

Of particular interest are linear and binary trees. A tree is **linear** if its arity is 0 or 1 and each of its successor trees is linear. A tree is **binary** if its arity is 0 or 2 and each of its successor trees is binary. Here are three examples:



The left tree is linear, the middle tree is binary, and the right tree is neither linear nor binary.

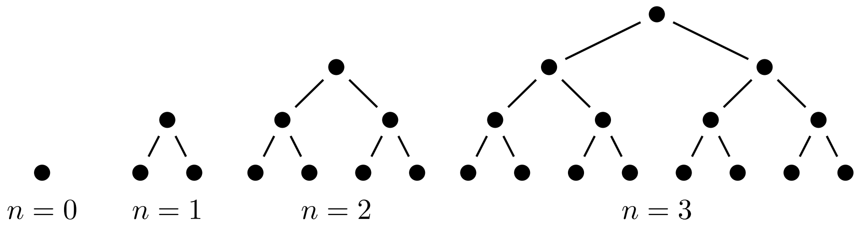
Here is a function that tests whether a tree is linear:

```
let rec linear (T ts) = match ts with
  | [] -> true
  | [t] -> linear t
  | _ -> false
```

Exercise 3.3.1 Specify a tree with 5 nodes that has exactly two subtrees. How many such trees are there?

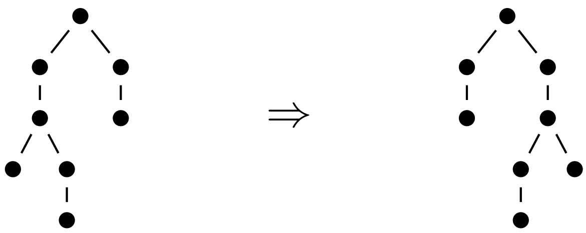
Exercise 3.3.2 Write a function `binary: tree -> bool` that tests whether a tree is binary.

Exercise 3.3.3 Write a function `tree: int -> tree` that returns binary trees for a given $n \geq 0$ as follows:



Make sure that the identical successor trees of the 2-ary subtrees are calculated only once each. This ensures that your function quickly computes a result even for $n = 1000$. Use the function `iter`.

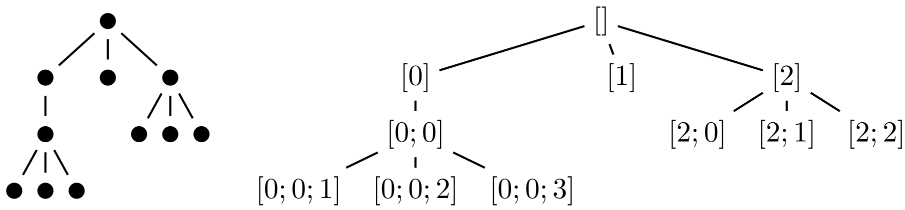
Exercise 3.3.4 Mirroring reverses the ordering of the successor trees of the subtrees of a tree:



Write a function `mirror: tree -> tree` that mirrors trees.

3.4 Addresses

In the graphical representation of a tree, each node can be described by a list called an **address**, which says how to get from the root to that node:



The root of a tree always has the address $[]$. The address $[2;1]$ means that, starting from the root, one should first follow the second and then the first downward edge (numbering from left to right, starting with zero). Here is a function that for a tree and an address computes the corresponding subtree for an address:

```
let rec subtree t xs = match xs with
  | [] -> t
  | x::xr -> subtree (succtree t x) xr
```

An address a (a list of positive integers) is called **valid** for a tree t if `subtree` computes a subtree for t and a. If `subtree` throws an exception, we speak of an invalid address:

Proposition 3.4.1 Two trees are exactly the same if and only if they have the same valid addresses.

We now have a total of three options for describing a tree:

1. Using an expression of the type `tree`.
2. Using the set of its valid addresses.
3. Using its graphical representation.

The representations of trees using expressions and valid addresses are suitable for programming, while the graphical representation of trees is very clear for humans but not suitable for programming.

For the next proposition, we need to define another term: A list xs is called a [**proper**] **prefix** of a list ys if there is a [non-empty] list zs with $xs @ zs = ys$.

Proposition 3.4.2 For each tree t, the following holds:

1. $[]$ is a valid address for t.
2. If a is a valid address for t, then any prefix of a is a valid address for t.
3. If $a @ [n-1]$ is a valid address for t and $0 \leq k \leq n-1$, then $a @ [k]$ is a valid address for t.

Exercise 3.4.3 Consider the graphical representation of the tree t_3 in Fig. 3.1.

1. Specify the address of the root.
2. Give the addresses of the inner nodes (there are exactly 4).
3. Specify the addresses of the occurrences of the subtree t_2.

Exercise 3.4.4 The arithmetic expressions from Section 3.1 can be thought of as trees as discussed in Section 3.2.2. Analogous to `subtree` write a function `subexp: exp -> int list -> exp` that computes the corresponding partial expression for an expression and an address. For example, `subexp` should compute the partial expression $2y$ for the expression $x(2y + 3)$ and the address $[1; 0]$. An exception shall be thrown for invalid addresses.

Exercise 3.4.5 Write functions of type `tree -> int list -> bool` as follows:

1. `node` tests whether an address denotes a node of a tree.
2. `root` tests whether an address denotes the root of a tree.
3. `inner` tests whether an address denotes an inner node of a tree.
4. `leaf` tests whether an address denotes a leaf of a tree.

Exercise 3.4.6 Write a function `prefix: 'a list -> 'a list -> bool` that tests whether a list is a prefix of a list.

Exercise 3.4.7 Declare `subtree` using `foldl`.

3.4.1 Successors and Predecessors

The valid addresses of a tree correspond exactly to the nodes in the graphical representation of the tree. This means that we can represent nodes by addresses and that we can define terms for nodes using addresses. Let a tree t be given and let a and a' be valid addresses for t. We say that

- the node denoted by a' is the k**-th successor** of the node denoted by a if $a' = a @ [k]$.
- the node denoted by a' is a **successor** of the node denoted by a if there exists a number k with $a' = a @ [k]$.
- the node denoted by a is the **predecessor** of the node denoted by a' if there exists a number k with $a @ [k] = a'$.
- the node denoted by a' is **subordinate** to the node denoted by a if there exists a list ks with $a' = a @ ks$.
- the node denoted by a is **superior** to the node denoted by a' if there exists a list ks with $a @ ks = a'$.

With the help of the graphical representation of trees, assure yourself of the meaning of the new terms. Obviously, a is the predecessor of a' exactly when a' is a successor of a, and a is superior to a' exactly when a' is subordinate to a.

Proposition 3.4.8 Let t be a tree. Then, all nodes except the root have exactly one predecessor. The root has no predecessor.

Exercise 3.4.9 Draw a tree with at least two nodes a and a' such that a is superior to node a', but a is not the predecessor of a'.

Exercise 3.4.10 Write a function `pred: int list -> int list -> bool` that tests for two addresses a and a', whether there is a tree in which a denotes the predecessor of the node denoted by a'.

Exercise 3.4.11 Write a function `superior: int list -> int list -> bool` that tests for two addresses a and a' whether there is a tree in which a denotes a node that is superior to the node denoted by a'.

3.5 Size and Depth

We define the **size** of a tree as the number of its nodes. For example, tree t_2 in Fig. 3.1 has size 4. We want to write a function `size: tree -> int` that determines the size of a tree. For this, we need recursion equations. Our starting point is the equation

$$\mathit{size}\,(T[t_0; \dots; t_{n-1}]) = 1 + \mathit{size}\ t_0 + \dots + \mathit{size}\ t_{n-1}$$

which states that the size of a tree is equal to one plus the sum of the sizes of its successor trees. This equation is suitable as a recursion equation because the successor trees are smaller than the tree formed from them. We implement the recursion over the list of successor trees described by the dot notation with the functions `foldl` and `map`:

```
let rec size (T ts) = foldl (+) (map size ts) 1
```

With the help of an example, convince yourself that the recursion tree for a function call $\mathit{size}\ t$ has the same form as the argument tree t, since for each node of t exactly one call of size is required. We define the **depth** of a tree as the maximum length of addresses valid for it. In illustrative terms, the depth of a tree is thus the maximum number of edges that can be traversed in its graphical representation on the way down from the root. For example, the tree t_3 in Fig. 3.1 has a depth of 3.

We want to define a function `depth: tree -> int` that determines the depth of a tree. Our starting point is the equation

$$\mathit{depth}\,(T[t_0; \dots; t_{n-1}]) = 1 + \max\{\mathit{depth}\ t_0; \dots; \mathit{depth}\ t_{n-1}\} \quad \text{for } n > 0$$

which states that the depth of a compound tree is equal to one plus the maximum depth of its successor trees. With a little trick, we can also generalise this equation to the case $n = 0$:

$$\mathit{depth}\,(T[t_0; \dots; t_{n-1}]) = 1 + \max\{-1; \mathit{depth}\ t_0; \dots; \mathit{depth}\ t_{n-1}\} \text{ for } n \geq 0$$

This equation is suitable as a recursion equation because the subtrees are smaller than the tree formed from them. This yields the following function:

```
let rec depth (T ts) = 1 + foldl Int.max (map depth ts) (-1)
```

Exercise 3.5.1 The **width** of a tree is the number of its leaves. For example, the tree t_3 has a width of 7. Declare a function `breadth: tree -> int` that determines the width of a tree.

Exercise 3.5.2 The **degree** of a tree is the maximum arity of its subtrees. For example, the tree t_3 has degree 3. Declare a function `degree: tree -> int` that determines the degree of a tree.

Exercise 3.5.3 Rewrite the function `size` given above such that it does not need `map`. Help: Perform the recursive application of `size` in the link function for `foldl`.

Exercise 3.5.4 Rewrite the function `depth` above such that it does not use `map`.

Exercise 3.5.5 For the arithmetic expressions from Section 3.1, declare functions `size: exp -> int` and `depth: exp -> int` that compute the size and depth of expressions. For example, for the expression $x(2y+3)$, `size` and `depth` should return the results 7 and 3, respectively.

3.6 Folding

Similar to lists, a folding function can be specified for trees, with which many functions on trees can be calculated without explicit recursion:

```
let rec fold f (T ts) = f (map (fold f) ts)
```

The idea behind the function `fold` is to evaluate a tree with a step function `f: 'a list -> 'a`, which determines the value of the tree from the values of the successor trees. For example, `fold` can be used to determine the size of a tree as follows:

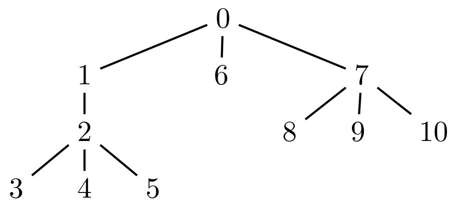
```
let size t = fold (fun ts -> foldl (+) ts 1) t
```

Exercise 3.6.1 With the help of `fold`, declare a function

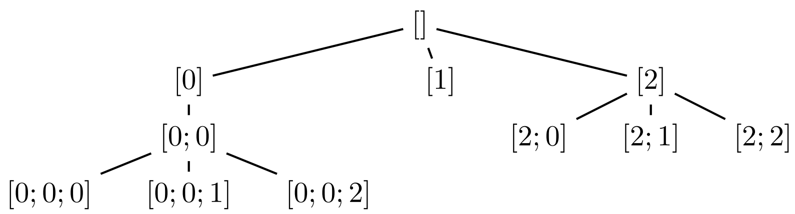
1. `depth: tree -> int` that determines the depth of a tree.
2. `breadth: tree -> int` that determines the breadth of a tree.
3. `degree: tree -> int` that determines the degree of a tree.
4. `mirror: tree -> tree` that mirrors a tree.

3.7 Pre-Ordering and Post-Ordering

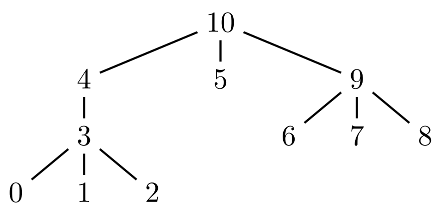
There are different ways to number the nodes in the graphical representation of a tree. With the so-called **pre-numbering**, one starts with the root and the number zero and numbers the successor trees by recursion from left to right. The following numbering results for the tree t_3 from Fig. 3.1:



The **pre-ordering** of the nodes expressed by the pre-numbering corresponds exactly to the ordering given by the lexicographic ordering (Section 2.14) of the addresses:



In so-called **post-numbering**, you start with the leftmost leaf and the number zero and work your way up step by step from left to right and from bottom to top. For our example tree, this results in the following numbering:



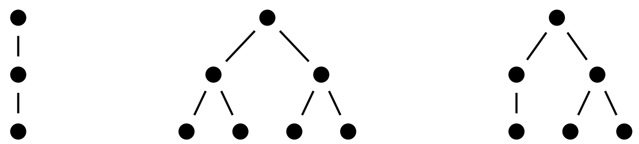
The ordering of the nodes expressed by the post-numbering is called the **post-ordering**.

The pre- and post-numbering of the nodes of the graphical representation of a tree can be explained graphically with the so-called **standard tour**. The standard tour visits the nodes of a tree according to the following rules:

1. the tour starts and ends at the root.
2. the tour follows the edges of the tree. Each edge of the tree is traversed exactly twice, first from top to bottom and then from bottom to top.

The successor trees of a node are visited according to their ordering (i. e. subtrees further to the left before subtrees further to the right). A node with n successor trees is visited exactly $n+1$ times by the standard tour. The pre-numbering results from numbering the nodes in the course of the standard tour at the first visit; the post-numbering results from numbering the nodes at the last visit. Be aware that the functions `size` and `depth` from Section 3.5 run through the argument tree according to the standard tour.

Exercise 3.7.1 For each of the following trees, give the pre- and post-numbering, respectively:



Exercise 3.7.2 The arithmetic expressions from Section 3.1 can be understood as binary trees, where the constructors `C` and `V` return atomic trees and the constructors `A` and `M` return compound trees.

1. Declare a function `prelin: exp -> int list list` that returns the addresses of an expression in pre-ordering.
2. Declare a function `postlin: exp -> int list list` that returns the addresses of an expression in post-ordering.

Exercise 3.7.3 Declare two functions `prelin` and `postlin` of the type `tree -> int list list` that return the addresses of a tree in pre- and post-ordering, respectively.

3.7.1 Subtree Access with Pre-Numbers

The numbers obtained by pre-numbering identify the nodes of a tree. We now want to implement a function that returns the subtree that is identified by a given tree and a number interpreted as a **pre-number**. For example, for the tree t_3 and the number 2, the tree t_2 is to be obtained (see Fig. 3.1). To implement the desired subtree access, we declare a more general function `prest: tree list -> int -> tree`, which computes the corresponding subtree for a list of trees and a given number. All subtrees in the list are numbered consecutively according to the pre-ordering, such that each subtree in the list can be addressed with a number:

$$\begin{aligned}
prest\ [t_3]\ 2 &= t_2 \\
prest\ [t_2; t_3]\ 0 &= t_2 \\
prest\ [t_2; t_3]\ 4 &= t_3 \\
prest\ [t_2; t_3]\ 6 &= t_2
\end{aligned}$$

The function `prest` can be easily described with two recursion equations:

$$\begin{aligned}
prest\ (t :: tr)\ 0 &= t \\
prest\ ((T\ ts) :: tr)\ k &= prest\ (ts\ @\ tr)\ (k-1) \qquad \text{for } k > 0
\end{aligned}$$

Exercise 3.7.4 Write a function `prest': tree -> int -> tree` that returns the corresponding subtree for a tree and a pre-number. If the given number is not a pre-number of the tree, an exception is to be thrown.

Exercise 3.7.5 The list argument of `prest` can be seen as a composition of the tree of which we want to compute the subtree access and an accumulator of type `tree list`. Write a function `presta: tree list -> tree -> int -> tree`, such that $prest\ [t]\ n = presta\ []\ t\ n$ applies to all trees and all pre-numbers n of t.

3.7.2 Subtree Access with Post-Numbers

You already guessed it: We now want to write a function `post` that returns the subtree identified by a post-number for a tree. As with the subtree access with pre-numbers, we work with a list of subtrees still to be processed, which we will call an **agenda** in the following. However, a subtree t can now be entered into the agenda in two ways: Either it still has to be completely processed (entry as $I\,t$), or its subtrees are already on the agenda before it (entry as $F\,t$):

$$\begin{aligned}
post\ (F\,t :: es)\ 0 &= t \\
post\ (F\,t :: es)\ k &= post\ es\ (k-1) \qquad \text{for } k > 0 \\
post\ (I(T[t_0; \dots; t_{n-1}]) :: es)\ k &= post\ ([I\,t_0; \dots; I\,t_{n-1}; \\
&\qquad\qquad F(T[t_0; \dots; t_{n-1}])]\ @\ es)\ k
\end{aligned}$$

The termination of these recursion equations results from the fact that the agenda becomes smaller with each recursion step. By the size of the agenda we mean the sum of the sizes of the entries, whereby an entry $I\,t$ has twice the size of t and an entry $F\,t$ has the size 1.

Exercise 3.7.6 Declare a function `post': tree -> int -> tree` that, given a tree and a post-number, computes the corresponding subtree. Use the following type declaration to implement the agenda of `post`:

```
type 'a entry = I of 'a | F of 'a
```

3.7.3 Linearisations

Trees can be represented by lists over $\mathbb{N}$. Each node is represented by its arity. In the case of **pre-linearisation**

```
let rec pre (T ts) = length ts :: List.concat (map pre ts)
```

the ordering of the nodes is determined according to the pre-ordering:

$$\begin{aligned} T[] &\rightsquigarrow [0] \\ T[T[]] &\rightsquigarrow [1;0] \\ T[T[];T[]] &\rightsquigarrow [2;0;0] \\ T[T[];T[T[];T[]];T[T[]]] &\rightsquigarrow [3;0;2;0;0;1;0] \end{aligned}$$

In the case of **post-linearisation**

```
let rec post (T ts) = List.concat (map post ts) @ [length ts]
```

the ordering of the nodes is determined according to the post-ordering:

$$\begin{aligned} T[] &\rightsquigarrow [0] \\ T[T[]] &\rightsquigarrow [0;1] \\ T[T[];T[]] &\rightsquigarrow [0;0;2] \\ T[T[];T[T[];T[]];T[T[]]] &\rightsquigarrow [0;0;0;2;0;1;3] \end{aligned}$$

Post-linearisation is sometimes referred to as **Polish notation**, after its discoverer Jan Łukasiewicz.

Both linearisations possess the property that a tree can be reconstructed from its linearisation. We will get to know the reconstruction algorithms later in the context of practical applications.

Exercise 3.7.7 Give both the pre- and post-linearisation of the tree $T[T[];T[T[]];T[T[];T[]]]$.

Exercise 3.7.8 Are there lists over $\mathbb{N}$ that do not represent trees according to the pre- or post-linearisation?

3.8 Balancedness

We call a tree **balanced** if the addresses of its leaves all have the same length. In illustrative terms, this means that all the leaves have the same distance from the root. For example, trees t_1 and t_2 in Fig. 3.1 are balanced, while t_3 is not. To test whether a tree is balanced, the following characterisation is useful.

Proposition 3.8.1 A tree is balanced if its successor trees are all balanced and all have the same depth.

Thus, to test the balancedness of a tree, we can proceed as follows: We calculate the depth of all successor trees and at the same time test that all subtrees are balanced. If a successor tree is not balanced or has a different depth from the other subtrees, we raise an exception. This results in a recursive function that follows the structure of the function `depth`.

We begin with a variant of the function `depth` from Section 3.5:

```
let rec depth' (T ts) = match ts with
  | [] -> 0
  | t::tr -> 1 + foldl Int.max (map depth' tr) (depth' t)
```

that has separate cases for atomic and compound trees. We change the case for compound trees in such a way that it additionally tests whether all successor trees have the same depth:

```
exception Unbalanced
let check n m = if n = m then n else raise Unbalanced
let rec depthb (T ts) = match ts with
  | [] -> 0
  | t::tr -> 1 + foldl check (map depthb tr) (depthb t)
```

If the property checked by `check` is fulfilled for all subtrees of a tree, the entire tree is balanced. However, if this property is violated for one of the subtrees, this subtree and, thus, the entire tree is unbalanced. The function `depthb`, therefore, terminates properly if and only if the tree is balanced. In this case, it returns the depth of the tree as a result. By catching the exception `Unbalanced`, we can obtain a function that tests whether a tree is balanced:

```
let balanced t = try (let _ = depthb t in true) with
    Unbalanced -> false
```

Exercise 3.8.2 Declare a function `balanced` using a let expression such that the exception `Unbalanced`, as well as the functions `check` and `depthb` are declared locally in the body of the function `balanced`.

3.9 Finitary Sets and Directed Trees

A set is called **pure** if each of its elements is a pure set. The simplest pure set is the empty set. A set is called **finitary** if it is finite and each of its elements is a finitary set. The set $\{\emptyset, \{\emptyset\}\}$ is finitary and pure. The infinite set $\{\emptyset,\ \{\emptyset\},\ \{\{\emptyset\}\},\ \dots\}$ is pure but not finitary. Obviously, every finitary set is a pure set.

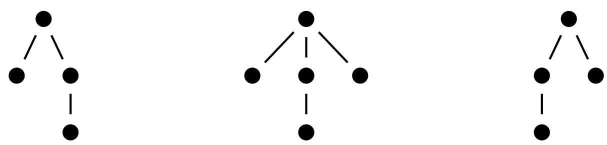
Pure sets are more interesting than they may appear at first sight: All mathematical objects can be represented by pure sets. Pure sets are, therefore, a universal data structure for the representation of mathematical objects.

We will restrict ourselves here to finitary sets, whose recursive structure corresponds to that of rose trees. However, lists are not involved when speaking about sets. Therefore, ordering and multiple occurrences of the elements do not play a role. We describe the connection between rose trees and finitary sets by the equation

$$\text{Set } (T[t_0; \dots; t_{n-1}]) = \{\text{Set } (t_0), \dots, \text{Set } (t_{n-1})\}$$

that assigns a finitary set Set t to each rose tree t by recursion. For example, Set $(T[T[]]) = \{\emptyset\}$. Set t is the set whose elements are exactly the sets represented by the successor tree of t.

Convince yourself that any finitary set can be represented by a rose tree. As expected, the representation of finitary sets by trees is not unique. For example, the set $\{\emptyset, \{\emptyset\}\}$ can be represented by any of the following trees:



We get an unambiguous representation for finitary sets if we restrict ourselves to trees whose subtree lists are strictly sorted (according to the lexicographic tree ordering, see Section 3.2.3). We refer to such trees as **directed**. Of the trees shown above, only the one on the left is directed. Here is a function that tests whether a tree is directed:

```
let rec strict ts = match ts with
  | t::t'::tr -> compareTree t t' = -1 && strict (t'::tr)
  | _ -> true
let rec directed (T ts) = strict ts && List.for_all directed
    ts
```

We want to develop an algorithm that decides whether two rose trees describe the same set. We start with the following equivalences:

1. Set t_1 = Set t_2 if and only if Set $t_1 \subseteq$ Set t_2 and Set $t_2 \subseteq$ Set t_1.
2. Set $t_1 \subseteq$ Set t_2 if and only if Set $t_1' \in$ Set t_2 for every successor tree t_1' of t_1.
3. Set $t_1 \in$ Set t_2 if and only if Set t_1 = Set t_2' for some successor tree t_2' of t_2.

With this, we can trace the test "Set t_1 = Set t_2" back to the test "Set $t_1 \subseteq$ Set t_2", and the test "Set $t_1 \subseteq$ Set t_2" back to the test "Set $t_1 \in$ Set t_2". Finally, we can trace the test "Set $t_1 \in$ Set t_2" back to the test "Set t_1 = Set t_2". This **interleaved recursion** terminates because with each run, at least one of the argument trees becomes smaller and the other does not become larger. We implement the algorithm with the following three functions

```
let rec eqset x y = subset x y && subset y x
and subset (T xs) y = List.for_all (fun x -> member x y) xs
and member x (T ys) = List.exists (eqset x) ys
```

which all have the type `tree -> tree -> bool`. For the declaration of `subset` and `member`, we use the keyword **`and`** instead of **`let`** in order to allow for the interleaved recursion between the functions `eqset`, `subset`, and `member`.

Exercise 3.9.1 Interleaved recursion can always be traced back to simple recursion. Assure yourself of this using the function `eqset` as an example: Declare a semantically equivalent function that only uses simple recursion.

Exercise 3.9.2 Declare a function `direct: tree -> tree` that, for a given tree, computes a directed tree that represents the same set. Use the polymorphic sort function from Section 2.13 and the compare function `compareTree` from Section 3.2.3.

Exercise 3.9.3 Assume that finitary sets are represented by directed trees.

1. Declare functions of the type `tree -> tree -> tree` that compute the union $X \cup Y$, the intersection $X \cap Y$, and the difference $X - Y$ for two sets X, Y.
2. Declare functions of the type `tree -> tree -> bool` that test for two sets X, Y whether $X \in Y$ or $X \subseteq Y$ holds, respectively.

Exercise 3.9.4 (Set Representation of Natural Numbers) Natural numbers can be uniquely represented by finitary sets according to the following equation:

$$\text{Set } n = \text{if } n = 0 \text{ then } \emptyset \text{ else Set}(n-1)$$

For example, it holds that Set $3 = \{\{\{\emptyset\}\}\}$.

1. Declare a function `code: int -> tree` that, for a given $n \in \mathbb{N}$, computes the directed tree representing the finitary set that represents n.
2. Declare a function `decode: tree -> int` such that $decode(code\ n) = n$ for all $n \in \mathbb{N}$. For arguments that are not a representation of a natural number, `decode` shall raise an invalid argument exception.
3. Declare two functions `add` and `mul` of the type `tree -> tree -> tree` , which add and multiply the representations of natural numbers, respectively. Do not use operations for `int`.

Exercise 3.9.5 (Set Representation of Pairs) Pairs can be uniquely represented by sets according to the following equation:

$$\mathrm{Set}(x, y) = \{\{x\},\ \{x, y\}\}$$

Familiarise yourself with this fact by implementing two functions `code : tree * tree -> tree` and `decode: tree -> tree * tree`, for which it holds that

1. for all directed trees t_1, t_2, $decode(code(t_1, t_2)) = (t_1, t_2)$ and
2. given directed trees, `code` and `decode` both compute directed trees.

If `decode` detects that its argument cannot be the encoding of a pair, it shall raise an invalid argument exception.

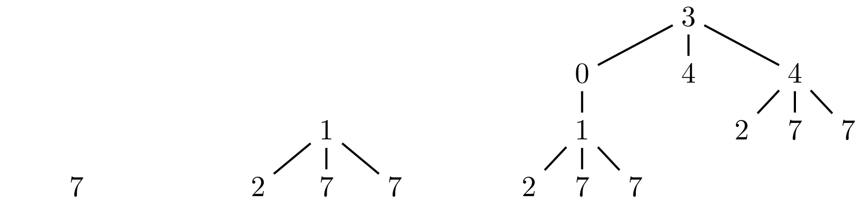
3.10 Labelled Trees

In practice, the nodes of tree-like objects usually carry information. For example, the nodes of arithmetic expressions carry information about whether they describe addition or multiplication operations or certain constants or variables. We now want to take a closer look at this phenomenon. For this purpose, we work with so-called **labelled trees**, in which each node is provided with a value from a **basic type**:

```
type 'a ltree = L of 'a * 'a ltree list
```

A labelled tree over a type t thus consists of a value of t and a list of labelled trees over t. Here are examples of trees over `int`:

```
let t1 = L(7, [])
let t2 = L(1, [L(2, []); t1; t1])
let t3 = L(3, [L(0, [t2]); L(4, []); t2])
```



$t_1 = L(7, [])$ $t_2 = L(1, [L(2, []); t_1; t_1])$ $t_1 = L(3, [L(0, [t_2]); L(4, []); t_2])$

For a labelled tree over a type t, each node is assigned a value from t, which is called the node's **label**. We call the label of a tree's root the **head** of the tree:

```
let head (L(x, _)) = x
```

By definition, a labelled tree is determined by its head and its subtree list. By the **shape** of a labelled tree we mean the rose tree obtained by omitting the labels:

```
let rec shape (L(_, ts)) = T(map shape ts)
```

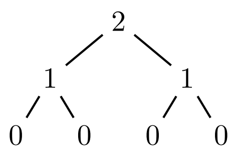
All the terms that we have defined for rose trees are transferred to labelled trees via the shape. Here is a function that tests whether two labelled trees have the same shape:

```
let sameShape t t' = shape t = shape t'
```

Exercise 3.10.1 Declare functions that determine the size and depth of labelled trees.

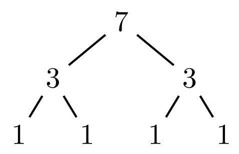
Exercise 3.10.2 Declare functions `leftmost` and `rightmost` of type `'a ltree -> 'a` that compute the label of the leftmost and rightmost leaf of the tree, respectively.

Exercise 3.10.3 Declare a function `ltreed: int -> int ltree` that for $n \geq 0$ computes a balanced binary tree with depth n, whose subtrees are labelled with their depth. For $n = 2$, `ltreed` shall compute the tree



Use the function `iterup`.

Exercise 3.10.4 Declare a function `ltrees: int -> int ltree` that for $n \geq 0$ computes a balanced binary tree with depth n, whose subtrees are labelled with their size. For $n = 2$, `ltrees` shall compute the tree



Use the function `iter`.

Exercise 3.10.5 Declare a function `sum: int ltree -> int` that computes the sum of the labels of a tree. If a label appears several times, it should also be included several times in the total.

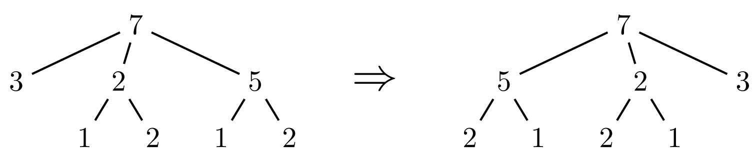
Exercise 3.10.6 Declare a function `lmap: ('a -> 'b) -> 'a ltree -> 'b ltree` that applies a function on every label in a given tree. Use the function `map` for lists.

Exercise 3.10.7 Declare a function `forall: ('a -> bool) -> 'a ltree -> bool` that tests whether a given function evaluates to `true` for all labels of a tree.

Exercise 3.10.8 Declare a function `prestl: 'a ltree -> int -> 'a` that, for a tree and a pre-number, computes the label of the node identified by the number. Use the function `prest` in Section 3.7 as a guide.

Exercise 3.10.9 Declare a function `find: ('a -> bool) -> 'a ltree -> 'a option` that, for a given function and a tree, computes the first label of the tree according to pre-ordering for which the function evaluates to `true`. Use the function `prest` in Section 3.7 as a guide.

Exercise 3.10.10 Mirroring reverses the ordering of the successor trees of the subtrees of a tree:



Write a function `mirror: 'a ltree -> 'a ltree` that mirrors trees.

Exercise 3.10.11 Declare a compare function `compareLtree: ('a -> 'a -> int) -> 'a ltree -> 'a ltree -> int` for labelled trees that combines an ordering for the labels with the lexicographic tree ordering. For example, *compareLtree Int.compare* t_1 t_2 shall evaluate to 1.

3.11 Projections

By the **pre-projection** *prep* t of a labelled tree t we refer to the list of its labels (with multiple occurrences) ordered according to their pre-ordering:

$$prep(L(x, [t_0; \dots ; t_{n-1}])) = [x] \mathbin{@} prep\ t_0 \mathbin{@} \dots \mathbin{@} prep\ t_{n-1}$$

For example, it holds that $prep\ t_3 = [3; 0; 1; 2; 7; 7; 4; 1; 2; 7; 7]$. Accordingly, by the **post-projection** of a labelled tree, we refer to the list of its labels (with multiple occurrences) ordered according to their post-ordering:

$$pop(L(x, [t_0; \dots ; t_{n-1}])) = pop\ t_0 \mathbin{@} \dots \mathbin{@} pop\ t_{n-1} \mathbin{@} [x]$$

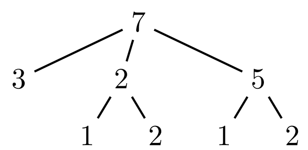
For instance, it holds that $prep\ t_3 = [2; 7; 7; 1; 0; 4; 2; 7; 7; 1; 3]$.

Exercise 3.11.1 Declare a function [basewidth=0.48em]|prep: 'a ltree -> 'a list| that computes the **pre-projection** of a labelled tree.

Exercise 3.11.2 Declare a function [basewidth=0.5em]|pop: 'a ltree -> 'a list| that computes the **post-projection** of a labelled tree.

Exercise 3.11.3 The **boundary** of a labelled tree is the list of labels of its leaves, in the order of their occurrence from left to right and with multiple occurrences. The frontier of the tree t_3 is $[2; 7; 7; 4; 2; 7; 7]$. Declare a function `frontier: 'a ltree -> 'a list` that computes the boundary of a label.

Exercise 3.11.4 The ***n*-th level** of a labelled tree is the list of labels of those nodes that have distance n from the root, arranged according to the graphical arrangement of the nodes. As an example, let us consider the following tree:

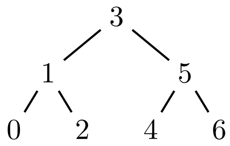


Its zeroth level is $[7]$, the first level is $[3; 2; 5]$, the second level is $[1; 2; 1; 2]$, and the third and all subsequent levels are $[]$. Declare a function `level: 'a ltree -> int -> 'a list` that computes the n-th level of a tree.

Exercise 3.11.5 For the **in-projection** of a binary tree, the labels of the inner nodes appear after the labels of the respective left and before the labels of the respective right successor tree:

$$inpro(L(x, [t_1; t_2])) = inpro\ t1 \mathbin{@} [x] \mathbin{@} inpro\ t_2$$

For example, the tree



has the in-projection $[0; 1; 2; 3; 4; 5; 6]$. Declare a function `inpro: 'a ltree -> 'a list` that computes the in-projection of a binary tree. When the function is applied to a tree that is not binary, an invalid argument exception shall be thrown.

Remarks

In this chapter, we have precisely described the concept of a tree-like object with the help of executable standard models for rose trees and labelled trees. Standard models play an important role in computer science because they can be used to develop concepts and algorithms in a general form that can be transferred to various applications.