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 January 31, 2024

Contents

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 $\mathit{int} \to \mathit{int} \to \mathit{bool}$ 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 $\mathit{test} : \mathit{int} \to \mathit{int} \to \mathit{int} \to \mathit{bool}$ 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 $\mathit{int} \to \mathit{int} \to \mathit{bool}$ 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 Section 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

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

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 Section 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:

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

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 Section 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 Section 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 (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 *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 *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 $sum : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ 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 $f(n) := n^2$. (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

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

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:

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

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:

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

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 $\mathit{fac} : \mathit{int} \to \mathit{int}$ 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 $\mathit{fac}' : \mathit{int} \to \mathit{int} \to \mathit{int}$ such that $\mathit{fac}'\ \mathit{1}$ 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

$$(2, 1, 39) \ : \ \mathit{int} \times \mathit{int} \times \mathit{int}$$

consisting of three integers. The product symbol "×" 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:

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

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

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

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 $y \leq z$ 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 $y \leq z$ 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

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

as the OCaml function

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

receiving the type $int \to int \to int$. In OCaml we now have the spurious situation that $rem\ x\ 0$ 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:

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

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 $eager\_or\ e_1\ e_2$ 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 $fst\,(\texttt{true},5)$ 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 $x + 1$:

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

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

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

We define a polymorphic **iteration operator**

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{iter} :\ \forall\alpha.\ (\alpha \to \alpha) \to \mathbb{N} \to \alpha \to \alpha} \\ \mathit{iter}\ f\ 0\ x & := & x \\ \mathit{iter}\ f\ (n+1)\ x & := & \mathit{iter}\ f\ n\ (f\ x) \end{array}$$

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**

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

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

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

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

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

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 $\mathit{nth\_prime}\ \mathit{0} = \mathit{2}$. 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. $x \geq 2 \land \forall k \geq 2.\, \forall n \geq 2.\;\; x \neq k \cdot n$
2. $x \geq 2 \land \forall k,\; 2 \leq k < x.\;\; x \,\%\, k > 0$
3. $x \geq 2 \land \forall k > 1,\; k^2 \leq x.\;\; x \,\%\, k > 0$
4. $x \geq 2 \land \forall k,\, 1 < k \leq \sqrt[2]{x}.\;\; x \,\%\, k > 0$

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]

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

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:

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

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

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 Section 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

[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.

cannot be realized computationally. OCaml bypasses the problem by 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 $[x_1, \dots, x_n]$ 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,

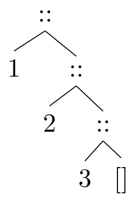
$$\begin{aligned} [1,2,3] &: \mathcal{L}(\mathbb{Z}) \\ [\mathsf{true},\mathsf{true},\mathsf{false}] &: \mathcal{L}(\mathbb{B}) \\ [(\mathsf{true},1),(\mathsf{true},2),(\mathsf{false},3)] &: \mathcal{L}(\mathbb{B}\times\mathbb{Z}) \\ [[1,2],[3],[]] &: \mathcal{L}(\mathcal{L}(\mathbb{Z})) \end{aligned}$$

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

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

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

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

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

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

$$\mathit{rev}\,[x_1,\dots,x_n] = [x_n,\dots,x_1]$$

For instance, $\mathit{rev}\,[1,2,3] = [3,2,1]$. 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 $\mathit{rev}\,(x :: l) = \mathit{rev}\,(l) \mathbin{@} [x]$ which expresses a basic fact about reversal. This brings us to the following definition of **list reversal**:

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

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

$$\begin{aligned} \mathit{rev\_append} :\ & \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \\ \mathit{rev\_append}\ [\,]\ l_2 &\coloneqq l_2 \\ \mathit{rev\_append}\ (x :: l_1)\ l_2 &\coloneqq \mathit{rev\_append}\ l_1\ (x :: l_2) \end{aligned}$$

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

$$\begin{aligned} \mathit{rev\_append}\ [1,2,3]\ [\,] &= \mathit{rev\_append}\ [2,3]\ [1] \\ &= \mathit{rev\_append}\ [3]\ [2,1] \\ &= \mathit{rev\_append}\ [\,]\ [3,2,1] \\ &= [3,2,1] \end{aligned}$$

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

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

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):

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

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 $init\ n\ f = [f(0), \ldots, f(n-1)]$ 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 $[f(m), f(m+1), \ldots, f(m+n-1)]$ and $[f(m), f(m+1), \ldots, f(n)]$.

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:

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

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:

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

OCaml provides this function as $\mathit{List.cons}$.

2.5 Membership and List Quantification

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

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

Note that mem tail-recurses on the list argument. Also recall the discussion of OCaml's polymorphic equality test in Section 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:

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

The expression

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

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

$$\mathit{mem}\ x\ l \;=\; \mathit{exists}\ ((=)x)\ l$$

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 Section 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

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

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 Section 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 $\mathit{fst} : \forall\alpha\beta.\ \alpha \times \beta \to \alpha$ 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 $\mathit{swap} : \forall\alpha\beta.\ \alpha \times \beta \to \beta \times \alpha$ 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:

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

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:

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

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

$$\mathit{is\_sublist}\ l_1\ l_2 \ = \ \mathit{mem}\ l_1\ (\mathit{pow}\ l_2)$$

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

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

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 $\mathit{foldl}\ f\ l\ b = \mathit{fold}\ f\,(\mathit{rev}\ l)\ b$:

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

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

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

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:

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

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 (Section 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

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

Exercise 2.12.1 Declare a function $\mathit{sorted} : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathbb{B}$ 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 $\mathit{perm} : \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathbb{B}$ that tests whether two lists are equal up to reordering.

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

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

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

$$\mathit{isort}\ l \ := \ \mathit{fold}\ \mathit{insert}\ l\ []$$

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

$$\mathit{isort}\ l \ :=\ \mathit{foldl}\ \mathit{insert}\ l\ []$$

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

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

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

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

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:

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

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

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

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

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:

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

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 (Section 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**

- $2 \leq k \leq x$ and no number $2 \leq n < k$ 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 $x - k \geq 0$ 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 $x \geq 2$. 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 $x \geq 2$.

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:

$$(\mathit{key},\ \mathit{value})$$

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:

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

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:

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

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

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

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

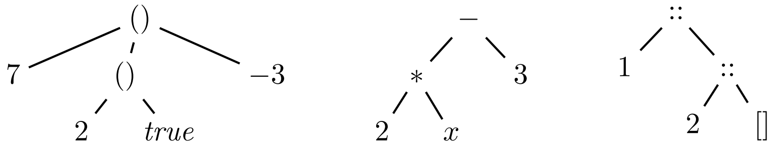
and define *empty* and the operations *update* and *lookup* accordingly. Test your solution with

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

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* → *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]
```

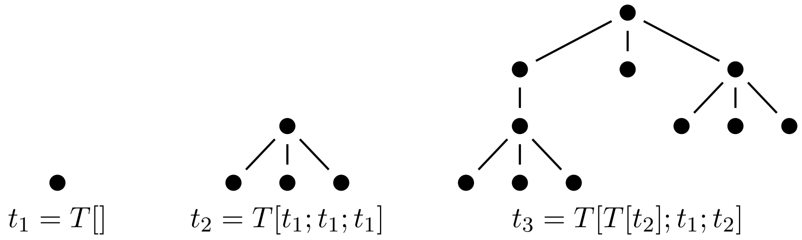
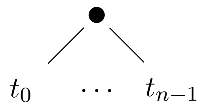


Figure 3.2.1: Graphical representation of rose trees.

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. 3.2.1. 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.

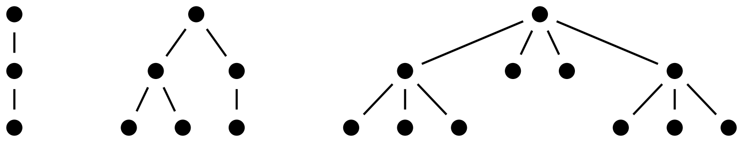
The tree $T[]$ is called **atomic**. All other trees are called **compound**. That means that there is exactly one atomic tree.

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

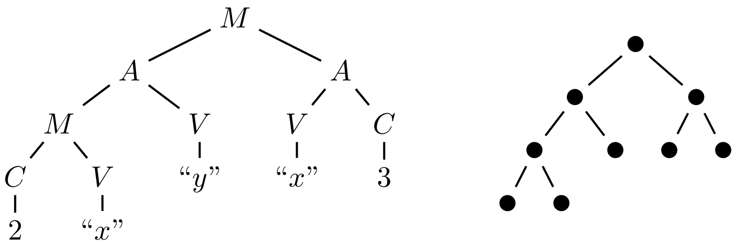
The arithmetic expressions from Section 3.1

```
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.

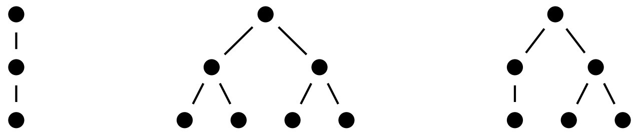
Here is a function that tests for a tree t and a tree t' whether t is a subtree of t':

```
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. 3.2.1). 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.

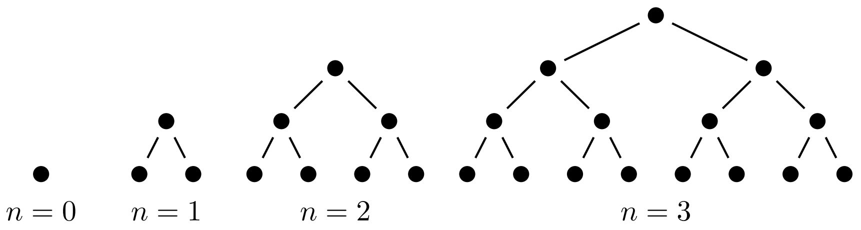
Here is a function that tests whether a tree is linear:

```
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?

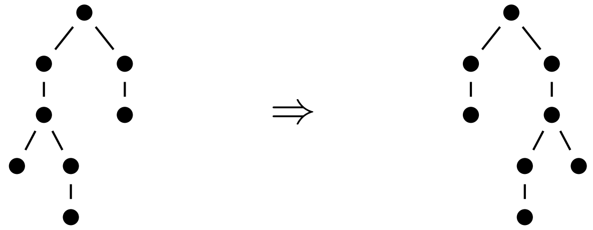
Exercise 3.3.2 Write a function `binary: tree -> bool` that tests whether a tree is binary.

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`.

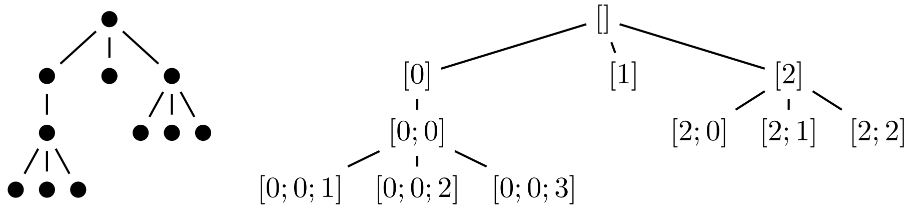
Exercise 3.3.4 Mirroring reverses the ordering of the successor trees of the subtrees of a tree:



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 $xs \mathbin{@} zs = ys$.

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 $a \mathbin{@} [n-1]$ is a valid address for t and $0 \leq k \leq n-1$, then $a \mathbin{@} [k]$ is a valid address for t.

Exercise 3.4.3 Consider the graphical representation of the tree t_3 in Fig. 3.2.1.

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

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. 3.2.1 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

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

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. 3.2.1 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

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

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 $n = 0$:

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

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:

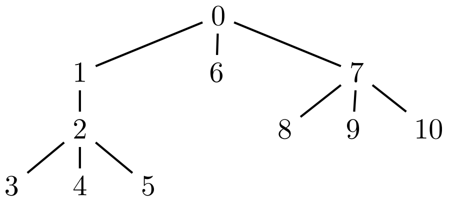
```
let size t = fold (fun ts -> foldl (+) ts 1) t
```

Exercise 3.6.1 With the help of `fold`, declare a function

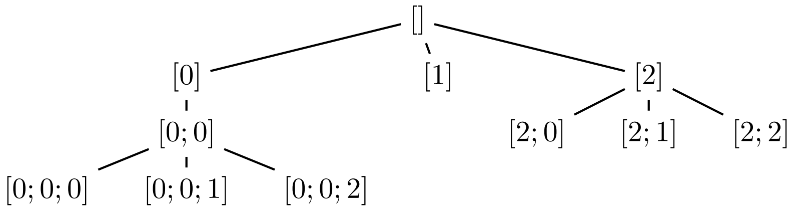
1. `depth: tree -> int` that determines the depth of a tree.
2. `breadth: tree -> int` that determines the breadth of a tree.
3. `degree: tree -> int` that determines the degree of a tree.
4. `mirror: tree -> tree` that mirrors a tree.

3.7 Pre-Ordering and Post-Ordering

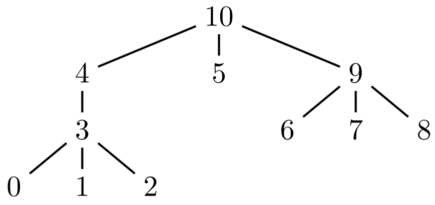
There are different ways to number the nodes in the graphical representation of a tree. With the so-called **pre-numbering**, one starts with the root and the number zero and numbers the successor trees by recursion from left to right. The following numbering results for the tree t_3 from Fig. 3.2.1:



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:



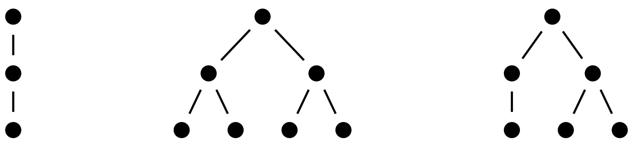
The ordering of the nodes expressed by the post-numbering is called the **post-ordering**.

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. 3.2.1). 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:

$$\begin{aligned}
\mathit{prest}\ [t_3]\ 2 &= t_2\\
\mathit{prest}\ [t_2; t_3]\ 0 &= t_2\\
\mathit{prest}\ [t_2; t_3]\ 4 &= t_3\\
\mathit{prest}\ [t_2; t_3]\ 6 &= t_2
\end{aligned}$$

The function `prest` can be easily described with two recursion equations:

$$\begin{aligned}
\mathit{prest}\ (t :: \mathit{tr})\ 0 &= t\\
\mathit{prest}\ ((T\ \mathit{ts}) :: \mathit{tr})\ k &= \mathit{prest}\ (\mathit{ts}\ @\ \mathit{tr})\ (k-1) \qquad \text{for } k > 0
\end{aligned}$$

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 $\mathit{prest}\ [t]\ n = \mathit{presta}\ []\ t\ n$ 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 $F\,t$):

$$\begin{aligned}
\mathit{post}\ (F\,t :: \mathit{es})\ 0 &= t\\
\mathit{post}\ (F\,t :: \mathit{es})\ k &= \mathit{post}\ \mathit{es}\ (k-1) \qquad \text{for } k > 0\\
\mathit{post}\ (I(T[t_0; \dots; t_{n-1}]) :: \mathit{es})\ k &= \mathit{post}\ ([I\,t_0; \dots; I\,t_{n-1};\\
&\qquad F(T[t_0; \dots; t_{n-1}])]\ @\ \mathit{es})\ k
\end{aligned}$$

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. 3.2.1 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 $\{\emptyset,\ \{\emptyset\},\ \{\{\emptyset\}\},\ \ldots\}$ is pure but not finitary. Obviously, every finitary set is a pure set.

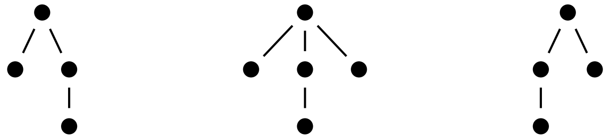
Pure sets are more interesting than they may appear at first sight: All mathematical objects can be represented by pure sets. Pure sets are, therefore, a universal data structure for the representation of mathematical objects.

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

$$\text{Set } (T[t_0; \ldots; t_{n-1}]) = \{\text{Set } (t_0), \ldots, \text{Set } (t_{n-1})\}$$

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:

$$\text{Set}(x, y) = \{\{x\},\ \{x, y\}\}$$

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.

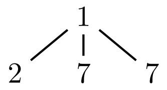
3.10 Labelled Trees

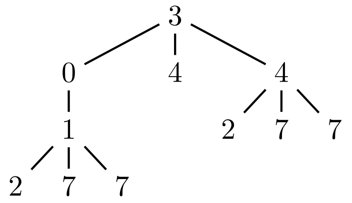
In practice, the nodes of tree-like objects usually carry information. For example, the nodes of arithmetic expressions carry information about whether they describe addition or multiplication operations or certain constants or variables. We now want to take a closer look at this phenomenon. For this 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)
```

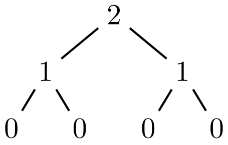
All the terms that we have defined for rose trees are transferred to labelled trees via the shape. Here is a function that tests whether two labelled trees have the same shape:

```
let sameShape t t' = shape t = shape t'
```

Exercise 3.10.1 Declare functions that determine the size and depth of labelled trees.

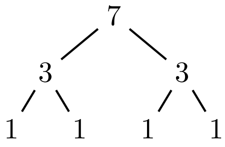
Exercise 3.10.2 Declare functions `leftmost` and `rightmost` of type `'a ltree -> 'a` that compute the label of the leftmost and rightmost leaf of the tree, respectively.

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.

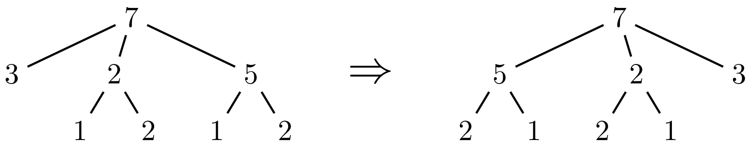
Exercise 3.10.6 Declare a function `lmap: ('a -> 'b) -> 'a ltree -> 'b ltree` that applies a function on every label in a given tree. Use the function `map` for lists.

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:

$$prep(L(x, [t_0; \dots ; t_{n-1}])) = [x] @ prep\ t_0 @ \dots @ prep\ t_{n-1}$$

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:

$$pop(L(x, [t_0; \dots ; t_{n-1}])) = pop\ t_0 @ \dots @ pop\ t_{n-1} @ [x]$$

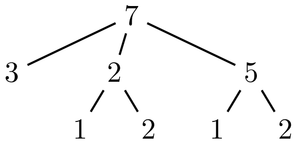
For instance, it holds that *prep* $t_3 = [2; 7; 7; 1; 0; 4; 2; 7; 7; 1; 3]$.

Exercise 3.11.1 Declare a function |prep: 'a ltree -> 'a list| that computes the **pre-projection** of a labelled tree.

Exercise 3.11.2 Declare a function |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:

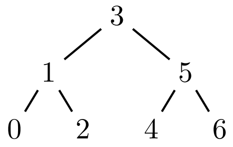


Its zeroth level is $[7]$, the first level is $[3; 2; 5]$, the second level is $[1; 2; 1; 2]$, and the third and all subsequent levels are $[]$. Declare a function `level: 'a ltree -> int -> 'a list` that computes the n-th level of a tree.

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:

$$inpro(L(x, [t_1; t_2])) = inpro\ t1 @ [x] @ inpro\ t_2$$

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.

4 Syntax and Semantics

In this chapter, we study a small idealized sublanguage of OCaml we call Mini-OCaml and discuss the syntactic and semantic structures that underlie Mini-OCaml and programming languages in general. To do so, we shall use mathematical tools known as grammars and derivation systems.

4.1 Overview

A programming language is defined by its **syntax** and **semantics** of which there are typically four different flavours:

Concrete Syntax is the set of “orthographic” rules we have to follow when we write down programs as text. Concrete syntax contains ornaments such as parentheses, semicolons, keywords, etc. On the one hand these are necessary to unambiguously encode the program structure. On the other hand they make the program easier to read for humans. For example, concrete syntax deals with issues such as operator precedence and associativity that are necessary to resolve ambiguities that arise from the infix notation of arithmetic expressions that we as humans are used to.

Abstract Syntax captures the mere **structure** of programs. In abstract syntax, we don’t care about semicolons, parentheses, keywords, etc. Abstract syntax gives rise to the **abstract syntax tree** (AST) of a program (Section 4.2).

Static Semantics is defined on the abstract syntax of the program and assigns an interpretation of the program that is independent of the program’s inputs. Typically, static semantics defines name binding rules, well-typedness, and other well-formedness properties of programs. The purpose of **semantic analysis** is to rule out exceptional cases for the dynamic semantics that would make the program **crash** and to infer information about the program that do not depend on the program’s inputs.

Dynamic Semantics defines the execution of the program. In OCaml, this mostly consists in specifying how expressions are evaluated on concrete values.

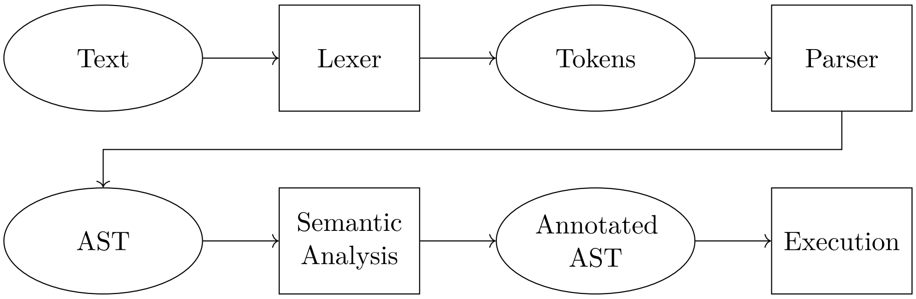


Figure 4.1.1: Overview of the different analyses and transformations a program undergoes before execution. (AST = abstract syntax tree.)

Whenever we run a program, the program undergoes several analyses and transformations that operate on the different flavours of syntax and semantics described above. Fig. 4.1.1 gives a visual summary.

First, **syntax analysis** checks if the program adheres to the concrete syntax of the programming language. Syntax analysis usually entails two steps, called lexing (Section 4.4) and parsing (Section 4.5). If syntax analysis fails, the program is not syntactically well-formed and is rejected. For example, the text `let 123abc = 1 in 1` contains a **lexical error** because identifiers in OCaml must start with a letter and therefore `123abc` is not a valid identifier and the program is not syntactically correct. The text `let = x 2 + in x` contains a **syntax error** because after `let` an identifier is expected. Therefore, this text is not a syntactically correct OCaml program. If syntax analysis succeeds, the parser transforms the program into its abstract syntax tree (AST).

Then, **semantic analysis** (Section 4.7 and Section 4.8) checks if the program adheres to the static semantics of the programming language and infers useful information about the program such as the types of expressions. This information is then annotated to the abstract syntax tree. If semantic analysis fails, the program is not well-formed and is rejected. For example, the program `let x = 1 in y + x` is syntactically correct but does not adhere to the static semantics of OCaml because the identifier `y` is not bound. Similarly, the program `let x = 1 in x 2` is syntactically correct but not well-typed because `x` is not a function but an integer.

After syntax and semantic analysis succeeded, the program is ready to be executed. Execution can happen in different ways. One way is to take the rules of the dynamic semantics and apply them to the abstract syntax tree of the program on a piece of paper. This way, we can manually compute the result of the program **in mathematics** and trace its execution step by step. Another way is to implement the dynamic semantics in a computer program. This program is called and **interpreter** and mechanises the manual application of the rules of the dynamic semantics (Section 4.10). A third way is to translate the program into another programming language, typically into some kind of **machine language** that the hardware of our computer can interpret. A program that performs this kind of translation is called a **compiler**.

We will now work through all the different levels of syntax and semantics and the corresponding analyses using our small idealised language Mini-OCaml.

4.2 Abstract Syntax

Mini-OCaml computes with **values** including booleans, numbers, and functions. Computations are described with **expressions**, which can take the following forms:

Constants for booleans and numbers (e.g., `true` and 17).

Variables such as x, y, z that can be bound to values.

Operator applications $e_1 \, o \, e_2$ where e_1 and e_2 are expressions and o is an operator (e.g., $+$ or $\leq$).

Conditionals $\textsc{if } e_1 \textsc{ then } e_2 \textsc{ else } e_3$.

Lambda expressions $\lambda(x : t).e$ consisting of a variable x, a type t, and an expression e.

Function applications $e_1 \, e_2$ consisting of two expressions e_1 and e_2.

Let expressions $\textsc{let } x = e_1 \textsc{ in } e_2$ consisting of a variable x and two expressions e_1 and e_2.

Recursive let expressions $\textsc{let rec } f\,(x : t_1) : t_2 = e_1 \textsc{ in } e_2$ consisting of two variables f and x, a parameter type t, and two expressions e_1 and e_2.

We have seen all these expressions in action in Chapter 1. In contrast to OCaml, Mini-OCaml has only local declarations appearing as part of let expressions and requires type annotations for function declarations because we will equip Mini-OCaml only with very limited type inference capabilities to keep things simple.

The expressions of Mini-OCaml listed above can be specified with a formal device called a **grammar**:

$$
\begin{array}{lcll}
e & ::= & c & \textit{constant} \\
  & | & x & \textit{variable} \\
  & | & e_1 \, o \, e_2 & \textit{operator application} \\
  & | & e_1 \; e_2 & \textit{function application} \\
  & | & \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 & \textit{conditional} \\
  & | & \lambda(x : t).e & \textit{lambda expression} \\
  & | & \text{LET } x = e_1 \text{ IN } e_2 & \textit{let expression} \\
  & | & \text{LET REC } f \; (x : t_1) : t_2 = e_1 \text{ IN } e_2 & \textit{recursive let expression}
\end{array}
$$

$$
\begin{array}{lcll}
t & ::= & \textit{bool} & \textit{boolean type} \\
  & | & \textit{int} & \textit{int type} \\
  & | & t_1 \to t_2 & \textit{function type}
\end{array}
$$

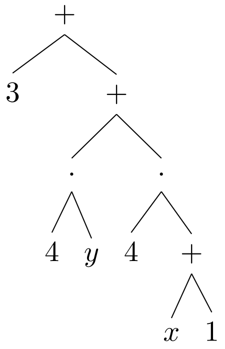
The grammar specifies different forms of expressions, using specific letters for the different kinds of **syntactic objects**: e for expressions, c for constants, and x and f for variables. The grammar clearly specifies the recursive structure of expressions. We speak of the **subexpressions** of an expression. For instance, a conditional IF e_1 THEN e_2 ELSE e_3 has three subexpressions e_1, e_2, and e_3, and a lambda expression $\lambda x.e$ has a single subexpression e. We also speak of the **constituents** of an expression, which are the syntactic objects an expression is combining. For instance, a lambda expression $\lambda x.e$ comes with two constituents x and e, a variable and an expression. Every subexpression is a constituent, of course.

The **abstract syntax tree** of an expression e is defined recursively: It consists of a node that represents e and is labelled by some text that uniquely identifies the alternative of the grammar that e corresponds to (this could be the operator symbol in the case of *operator application* or just LET in the case of *let expression*). The children of that node are the abstract syntax trees of the subexpressions of e. Here are some examples for abstract syntax trees[1]:

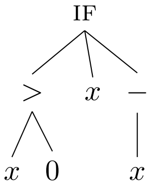
1. `3 + 4 * y + 4 * (x + 1)`
2. `if x > 0 then x else -x`

3.
   ```
   let rec f (n : int) : int =
     if n < 1 then 1 else x * f (n - 1) in f
   ```

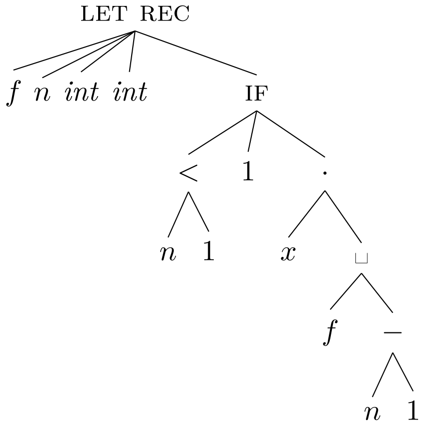
[1] The ␣ symbol in (3) stands for the *function application* branch of the grammar.



(1)



(2)



(3)

The recursive structure of expressions means that expressions can be **nested** into each other. For instance, when we form a function application $e_1\ e_2$, the grammar for expressions allows for all expressions for e_1. One says that **syntactically** the left expression of a function application is unconstrained. Note that a function application $e_1\ e_2$ can only evaluate successfully if e_1 evaluates to a function. Since evaluation is seen as a **semantic issue**, the constraint that the left expression of a function application must describe a function is an element of the **static semantics** (see Section 4.7). For instance, the expression $(1 + 2)\ 3$ is syntactically well-formed but violates the static semantics, because the expression $(1 + 2)$ is not a function.

Exercise 4.2.1 Extend the grammar for abstract expressions with rules for declarations and programs and design corresponding typing rules. Hint: Use judgments $E \vdash D \Rightarrow E'$ and $E \vdash P \Rightarrow E'$ and model programs with the grammar rule $P ::= \emptyset \mid D\ P$ ($\emptyset$ is the empty program).

4.3 Derived Forms

When describing a programming language, it is useful to distinguish between a **kernel language** and **derived forms**. Derived forms are constructs that are explained by translation to the kernel language. The advantage of this approach is that we can keep the static and dynamic semantics small. Fig. 4.3.1 shows several derived forms together with their translations into abstract expressions. Derived forms like these do exist in OCaml.

$\lambda x_1 \dots x_n.e$ cascaded lambda

$\rightsquigarrow \quad \lambda x_1. \cdots .\lambda x_n.e$

$\textsc{let}\ f x_1 \dots x_n = e_1\ \textsc{in}\ e_2$ lifted lambda

$\rightsquigarrow \quad \textsc{let}\ f = \lambda x_1 \dots x_n.\, e_1\ \textsc{in}\ e_2$

$\textsc{let}\ \textsc{rec}\ f\, x\, x_1 \dots x_n = e_1\ \textsc{in}\ e_2$ lifted lambda

$\rightsquigarrow \quad \textsc{let}\ \textsc{rec}\ f x\ = \lambda x_1 \dots x_n.\, e_1\ \textsc{in}\ e_2$

$e1$ && e_2 lazy and

$\rightsquigarrow \quad \textsc{if}\ e_1\ \textsc{then}\ e_2\ \textsc{else}$ false

$e1$ || e_2 lazy or

$\rightsquigarrow \quad \textsc{if}\ e_1\ \textsc{then}$ true $\textsc{else}\ e_2$

(o) lifted operator

$\rightsquigarrow \quad \lambda xy.\, x\, o\, y$

Figure 4.3.1: Derived forms with translation rules

4.4 Lexing

The lexical syntax of a programming languages fixes the characters a program can be written with. There are **white space characters** including space, horizontal tabulation, carriage return, and line feed. The lexical syntax also fixes different classes of words[2]. We have seen the following classes of words in OCaml:

Keywords We have seen the keywords **`let rec in if then else fun`**. The symbols `( ) = : -> * ,` also count as keywords.

Operators We have seen the operators `+ - / mod <> < <= > >=`. There are many more.

Boolean literals `true` and `false`.

Integer literals Examples are `0` and `2456`.

String literals For instance, `"Saarbrücken"`. String literals can contain special letters like the German umlauts.

[2] Other common names for words are: token or lexeme.

Identifiers Examples are `pow`, `pow'`, and `div_checked`. Identifiers start with a lower case letter and can contain letters, digits and the characters "`_`" (underline) and "`'`" (prime). Identifiers exclude words that appear as keywords, operators, or boolean literals. Moreover, identifiers must not contain special letters like the German umlauts. The identifiers `int`, `bool`, and `string` are used as names for the respective types.

Words are usually separated by white space characters. In some cases, white space characters are not needed, as for instance in the character sequence `3*(x/y)` where every character is a word.

The lexical word classes are disjoint. This shows in the symbols "`=`" and "`*`", which are classified as keywords but not as operators. This fits their use as keywords in declarations and types. However, the symbols "`=`" and "`*`" may also serve as infix operators in expressions. The necessary disambiguation will happen at the level of the phrasal syntax.

There are also **comments**, which are character sequences

```
(*  ...  *)
```

counting as white space. Comments are only visible at the lexical level. Comments are useful for remarks the programmer wants to write into a program. Here is a text where the words identified by the lexical syntax are marked:

```
(* Linear Search *)
let rec first f ( k : int ) = int :
if f k then k   (* base case *)
else first f ( k + 1 )   (* recursive case *)
```

Note that the characters of the three comments in the text count as white space and don't contribute to the words.

Consult the OCaml manual if you want to know more about the lexical syntax of OCaml.

Exercise 4.4.1 Mark the words in the following character sequences. For each word give the lexical class.

a) `if x <= 1 then true else f (x+1)`

b) `let rec f x : int = (*TODO*) +x`

c) `let city = "Saarbrücken" in`

d) `int * int -> bool`

e) `if rec then <3`

Why does the character sequence `Saarbrücken` not represent a valid word?

4.5 Parsing

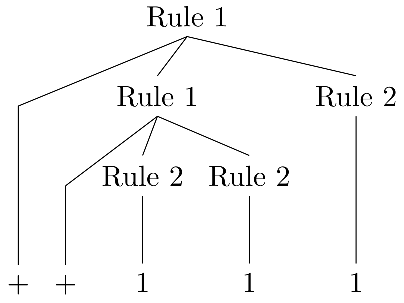
Parsing is the process of analysing the sequence of words produced by the lexer and checking if it adheres to the **concrete syntax** of the programming language. The parser does that by constructing a sequence of **derivation steps** that yield the input text from the **start symbol** of the grammar that defines the **concrete syntax** of the programming language. For example, given a grammar

$$\begin{array}{lcll} e & ::= & + \, e \, e & \text{(Rule 1)} \\ & | & 1 & \text{(Rule 2)} \end{array}$$

a parser for that grammar might find the following sequence of derivations for the input ++111. Deriving means to substitute a **non-terminal** (a symbol on the left side of ::=) with one of its right-hand sides:

$$e \overset{\text{Rule 1}}{\Rightarrow} +ee \overset{\text{Rule 1}}{\Rightarrow} ++eee \overset{\text{Rule 2}}{\Rightarrow} ++1ee \overset{\text{Rule 2}}{\Rightarrow} ++11e \overset{\text{Rule 2}}{\Rightarrow} ++111$$

Note that there are different orders in which Rule 1 and Rule 2 can be applied to achieve the same end result. A derivation sequence gives rise to the syntax tree of the program in the following way: The leaves of the tree are the tokens of the input text. The inner nodes of the tree correspond to the rules in the grammar that the parser used in the derivation. The syntax tree of the above derivation is

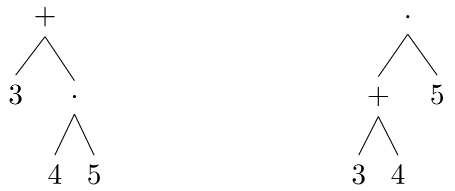


Now, one can show that for the input ++111 and the grammar shown above, every derivation sequence yields the same syntax tree. Even more, one can show that each text that can be derived by this grammar has a unique syntax tree. This is a property of the grammar and we say that such a grammar is **unambiguous**.

The crucial difference between concrete and abstract syntax is that the grammar that defines the abstract syntax usually is **intentionally ambiguous**. Consider the expression

$$3 + 4 \cdot 5$$

The grammar of Mini-OCaml's abstract syntax (see Section 4.2) allows for two derivations that have different syntax trees. Note that it is often common to label the inner nodes with a distinct token that identifies the rule (such as $+$ or $\cdot$):



The reason for this is that there is only one rule, namely

$$e ::= e \; o \; e$$

that describes the structure of arithmetic expressions. This rule allows for the two derivations

$$\begin{array}{lllll} e \Rightarrow e + e & \Rightarrow 3 + e & \Rightarrow 3 + e \cdot e & \Rightarrow 3 + 4 \cdot e & \Rightarrow 3 + 4 \cdot 5 \\ e \Rightarrow e \cdot e & \Rightarrow e \cdot 5 & \Rightarrow e + e \cdot 5 & \Rightarrow 3 + e \cdot 5 & \Rightarrow 3 + 4 \cdot 5 \end{array}$$

which have two different syntax trees. Still, the abstract syntax captures the **structure** of all programs of the language but it loses the correspondence to the (linear) program text. This is intentional because as soon as the parser constructed the syntax tree of a program, the program text is not needed any more and we only care about the abstract syntax of the program.

Here is an **unambiguous** grammar that describes the structure of arithmetic expressions **and** resolves the ambiguities above by implicitly encoding operator precedence and associativity.

$$\begin{array}{rcl} e & ::= & t + e \mid t - e \mid t \\ t & ::= & f * t \mid f/t \mid f \\ f & ::= & (e) \mid x \mid c \end{array} \tag{4.1}$$

To sum up: A parser needs an unambiguous grammar that describes all syntactically correct programs of the programming language. Using such a grammar, it analyses the input text trying to find a derivation sequence for the input program. While determining this sequence, the parser constructs the syntax tree of the program. Usually, the concrete syntax of the language is just a refinement of the abstract syntax so that the parser can directly construct the abstract syntax tree. In practice, one often uses **parser generators** that take a grammar as input and produce a parser for that grammar.[3]

[3] Menhir is a popular parser generator for OCaml.

Exercise 4.5.1 Construct a derivation sequence for the expression $3 + 4 \cdot 5$ using the grammar (4.1) and draw the corresponding syntax tree.

Exercise 4.5.2 Extend grammar (4.1) with the unary minus operator $-$ and a power operator $**$.

Exercise 4.5.3 How can right-associative operators (such as ::) be encoded in a grammar?

Exercise 4.5.4 Read-up on OCaml's operator precedence and associativity and think about how to encode it in an unambiguous grammar.

4.6 Derivation Systems

The semantic layers of a programming language are described with derivation systems. **Derivation systems** are systems of **inference rules** deriving statements called **judgments**. Judgments may be seen as syntactic objects. We explain derivation systems with a system deriving judgments $x \mathrel{\dot{<}} y$ where x and y are integers. The system consists of two inferences rules:

$$\frac{}{x \mathrel{\dot{<}} x+1} \qquad\qquad \frac{x \mathrel{\dot{<}} y \qquad y \mathrel{\dot{<}} z}{x \mathrel{\dot{<}} z}$$

We may verbalize the rules as saying:

1. Every number is smaller than its successor.
2. If x is smaller than y and y is smaller than z, then x is smaller than z.

Derivations of judgments are obtained by combining rules. Here are two different derivations of the judgment $3 \mathrel{\dot{<}} 6$:

$$\cfrac{\cfrac{}{3 \mathrel{\dot{<}} 4} \qquad \cfrac{\cfrac{}{4 \mathrel{\dot{<}} 5} \qquad \cfrac{}{5 \mathrel{\dot{<}} 6}}{4 \mathrel{\dot{<}} 6}}{3 \mathrel{\dot{<}} 6}$$

$$\cfrac{\cfrac{\cfrac{}{3 \mathrel{\dot{<}} 4} \qquad \cfrac{}{4 \mathrel{\dot{<}} 5}}{3 \mathrel{\dot{<}} 5} \qquad \cfrac{}{5 \mathrel{\dot{<}} 6}}{3 \mathrel{\dot{<}} 6}$$

Every line in the derivations represents the application of one of the two inferences rules. Note that the initial judgments of the derivation are justified by the first rule, and that the non-initial judgments of the derivation are justified by the second rule.

In general, an **inference rule** has the format

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the **conclusion** C is a judgment and the **premises** $P_1, \dots, P_n$ are either judgments or side conditions. A rule says that a derivation of the conclusion judgment can be obtained from the derivations of the premise judgments provided the side conditions appearing as premises are satisfied. The inference rules in our example system don't have side conditions.

Returning to our example derivation system, it is clear that whenever a judgment $x \dot{<} y$ is derivable, the comparison $x < y$ is true. This follows from the fact that both inference rules express valid properties of comparisons $x < y$. Moreover, given two integers x and y such $x < y$, we can always construct a derivation of the judgment $x \dot{<} y$ following a recursive algorithm:

1. If $y = x + 1$, the first rule yields a derivation of $x \dot{<} y$.
2. Otherwise, obtain a derivation of $x \dot{<} y - 1$ by recursion. Moreover, obtain a derivation of $y - 1 \dot{<} y$ by the first rule. Now obtain a derivation of $x \dot{<} y$ using the second rule.

The example derivation system considered here was chosen for the purpose of explanation. In practice, derivation systems are used for describing computational relations that cannot be described otherwise. Typing and evaluation of expressions are examples for such relations, and we are going to use derivation systems for this purpose.

Exercise 4.6.1 Give all derivations for $5 \dot{<} 9$.

Exercise 4.6.2 Give two inference rules for judgments $x \dot{<} y$ such that your rules derive the same judgments the rules given above derive, but have the additional property that no judgment has more than one derivation.

Exercise 4.6.3 Give two inference rules for judgments $x \dot{<} y$ such that a judgment $x \dot{<} y$ is derivable if and only if $x + k \cdot 5 = y$ for some $k \geq 1$.

4.7 Static Semantics

We formulate the static semantics of abstract expressions using **typing judgments**

$$E \vdash e : t$$

saying that in the environment E the expression e has type t. An **environment** is a finite collection of **bindings** $x : t$ mapping variables to types. We assume that environments are **functional**, that is, have at most one binding per variable. Here is an example for a typing judgment:

$$x : \mathit{int},\ \ y : \mathit{bool} \vdash \text{IF } y \text{ THEN } x \text{ ELSE } 0 \ \ : \ \ \mathit{int}$$

The inference rules in Fig. 4.7.1 establish a derivation system for typing judgments. We will refer to the rules as **typing rules**. If the judgment $E \vdash e : t$ is derivable, we say that expression e has type t in environment E. The environment is needed so that e can contain variables that are not bound by surrounding lambda expressions or let expressions. The typing rules for constants and for operator applications assume that the types for constants and operators are given in tables (one type per constant or operator).

Note that the rules for constants, variables, and operator applications have **tacit premises** ($c : t$, $(x : t) \in E$, $o : t_1 \to t_2 \to t$) that will not show up in derivations but need to be checked for a derivation to be valid. Tacit premises are better known as **side conditions**.

Here is a derivation for the above example judgement (we write E for the environment $[x : \mathit{int},\ \ y : \mathit{bool}]$):

$$\dfrac{\dfrac{}{E \vdash y : \mathit{bool}} \qquad \dfrac{}{E \vdash x : \mathit{int}} \qquad \dfrac{}{E \vdash 0 : \mathit{int}}}{E \vdash \text{IF } y \text{ THEN } x \text{ ELSE } 0 \ \ : \ \ \mathit{int}}$$

Note that the subderivations are established by the rules for variables and constants.

The rules for lambda expressions and let expressions **update** the existing environment E with a binding $x : t$. If E doesn't contain a binding for x, the binding $x : t$ is added. Otherwise, the existing binding for x is replaced with $x : t$. For instance,

$$\begin{aligned}
[x : t_1,\ \ y : t_2],\ \ z : t_3 \ &= \ [x : t_1,\ \ y : t_2,\ \ z : t_3] \\
[x : t_1,\ \ y : t_2],\ \ x : t_3 \ &= \ [y : t_2,\ \ x : t_3]
\end{aligned}$$

$$\frac{c : t}{E \vdash c \,:\, t} \qquad\qquad \frac{(x : t) \in E}{E \vdash x \,:\, t}$$

$$\frac{o : t_1 \to t_2 \to t \qquad E \vdash e_1 \,:\, t_1 \qquad E \vdash e_2 \,:\, t_2}{E \vdash e_1\, o\, e_2 \,:\, t}$$

$$\frac{E \vdash e_1 \,:\, t_2 \to t \qquad E \vdash e_2 \,:\, t_2}{E \vdash e_1\, e_2 \,:\, t}$$

$$\frac{E \vdash e_1 \,:\, \mathit{bool} \qquad E \vdash e_2 \,:\, t \qquad E \vdash e_3 \,:\, t}{E \vdash \textsc{if}\ e_1\ \textsc{then}\ e_2\ \textsc{else}\ e_3 \,:\, t}$$

$$\frac{E \vdash e_1 \,:\, t_1 \qquad E,\, x : t_1 \vdash e_2 \,:\, t}{E \vdash \textsc{let}\ x = e_1\ \textsc{in}\ e_2 \,:\, t}$$

$$\frac{E,\, x : t_1 \vdash e \,:\, t_2}{E \vdash \lambda(x : t_1).e \,:\, t_1 \to t_2}$$

$$\frac{E,\, f : t_1 \to t_2,\ x : t_1 \vdash e_1 \,:\, t_2 \qquad E,\, f : t_1 \to t_2 \vdash e_2 \,:\, t}{E \vdash \textsc{let rec}\ f\ (x : t_1) : t_2 = e_1\ \textsc{in}\ e_2 \,:\, t}$$

Figure 4.7.1: Typing rules

We say that an expression e is **well typed** in an environment E if the judgment $E \vdash e : t$ is derivable for some type t. Correspondingly, we say that an expressions e is **ill-typed** in an environment E if there is no type t such that the judgment $E \vdash e : t$ is derivable.

The typing rule for non-recursive let expressions

$$\frac{E \vdash e_1 \,:\, t_1 \qquad E,\, x : t_1 \vdash e_2 \,:\, t}{E \vdash \textsc{let}\ x = e_1\ \textsc{in}\ e_2 \,:\, t}$$

accommodates the local variable x since the premise $E \vdash e_1 \,:\, t_1$ determines the type for x. This argument assumes that a judgment $E \vdash e : t$ determines t given E and e. We will expand on this argument in the next section when we discuss a type checking algorithm.

From the typing rules for let expressions one can obtain typing rules for declarations and programs.

A system of typing rules is commonly referred to as a **type system**. One says that the typing rules formulate a **type discipline** for a language, and that type checking ensures that programs obey the type discipline.

Here is an example of a type derivation where t_1 and t_2 are placeholders for arbitrary types:

$$\dfrac{\dfrac{\dfrac{}{x : t_2 \vdash x : t_2}}{x : t_1 \vdash \lambda(x : t_2).x : t_2 \to t_2}}{[] \vdash \lambda(x : t_1).\lambda(x : t_2).x : t_1 \to t_2 \to t_2}$$

The derivation uses the typing rules for variables and lambda expressions. Note how elegantly the typing rules handle the shadowing of the first argument variable in $\lambda(x : t_1).\lambda(x : t_2).x$.

Exercise 4.7.1 For each of the following expressions e find the proper types and construct a derivation $[] \vdash e : t$. Try to give most general types.

a) $\lambda(x :?).x$

b) $\lambda(x :?).\lambda(y :?).yx$

c) $\lambda(x :?).\lambda(y :?).xx$

d) $\lambda(f :?).\lambda(g :?).\lambda(x :?).f\,x\,(g\,x)$

e) $\textsc{let rec}\ f\ (x :?) :? = f\,x\ \textsc{in}\ f$

f) $\lambda(x :?).\lambda(y :?).\lambda(z :?).\textsc{if}\ x\ \textsc{then}\ y\ \textsc{else}\ z$

4.8 Type Checking Algorithm

Given an environment E and an expression e, there is at most one type t such that $E \vdash e : t$. Moreover, there is a **type checking algorithm** that given E and e yields the corresponding type if there is one, and otherwise isolates a subexpression where the type checking fails. We may also say that a type checking algorithm constructs a derivation $E \vdash e : t$ given E and e.

The algorithm exploits that there is exactly one typing rule for every syntactic form. Moreover, the typing rules have the important property that the judgments appearing as premises of a rule contain only subexpressions of the expression appearing in the conclusion of the rule. Consequently, construction of a derivation for a given expression by backward application of the typing rules will always terminate.

The typing rules describe a **syntax-directed** type checking algorithm that given an environment E and an expression e checks whether there is a type t such that the judgment $E \vdash e : t$ is derivable. Given an expression e, the algorithm considers the uniquely determined typing rule for e and recurses on the instantiated premises of the rule. For instance, given an environment E and an expression $\text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3$, the algorithm follows the typing rule for conditionals

$$\frac{E \vdash e_1 : \mathit{bool} \qquad E \vdash e_2 : t \qquad E \vdash e_3 : t}{E \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 : t}$$

and recurses on the subproblems (E, e_1), (E, e_2), and (E, e_3). If the algorithm succeeds for the subproblems with the types t_1, t_2, and t_3, it checks the equations $t_1 = \mathit{bool}$ and $t_2 = t_3$. If the equations are satisfied, the algorithm has constructed a derivation for $E \vdash e : t_2$. If the equations are not satisfied, no such derivation exists and the algorithm fails with one of the following error messages:

1. Expression e_1 does not have type *bool*.
2. Expression e_2 and e_3 have different types.

We now have a type checking algorithm for expressions. The algorithm is given by the inference rules. We will eventually program the algorithm in OCaml. An important observation is that the typing rules give us a concise description of the algorithm. If we start to formulate the algorithmic reading of the rules with words, we end up with a lengthy text that is hard to understand, as we have seen at the example of the conditional rule.

Exercise 4.8.1 Give the typing rule for lambda expressions $\lambda x : t.\, e$ augmented with type specifications. Formulate the algorithmic reading of the rule with words.

Exercise 4.8.2 Give the typing rule for recursive let expressions

$$\text{LET REC } f\ (x : t_1) : t_2 = e_1 \text{ IN } e_2$$

augmented with type specifications. Formulate the algorithmic reading of the rule with words.

Exercise 4.8.3 For each of the following simple expressions e construct a derivation $E \vdash e : t$. Use the placeholders t_1, t_2, t_3, t_4 for types to avoid unnecessary commitments to concrete types. Try to give most general types.

a) fx
b) $fx(gx)y$
c) $\lambda x : t_1.\, \lambda y : t_2.\, x$
d) $\lambda x : t_1.\, \lambda x : t_2.\, x$
e) $\lambda x : t.\, fx(gx)y$
f) $\text{IF } x \text{ THEN } y \text{ ELSE } z$
g) $\text{IF } x \text{ THEN } y \text{ ELSE } x$
h) $\text{IF } x \text{ THEN } y + 0 \text{ ELSE } z$

Exercise 4.8.4 Test your understanding by giving typing rules for judgments $E \vdash e : t$ and expressions $e ::= x \mid \lambda x.e \mid ee$ without looking at Fig. 4.7.1.

4.9 Free and Local Variables

Given an expression e, we call a variable x **free in** e if x has an occurrence in e that is not in the **scope** of a **variable binder**. For instance, the variable x is free in the expression $(\lambda x.x)\,x$ since the 3rd occurrence of x is not in the scope of the lambda expression. We call an expression e **open** if some variable occurs **free** in e, and **closed otherwise**. For instance, the expression

$$\text{LET } f = \lambda x. fx \text{ IN } f$$

is open since f occurs free in it (2nd occurrence of f). On the other hand, the expression

$$\text{LET REC } fx = fx \text{ IN } f$$

is closed since the 2nd occurrence of f is in the scope of the recursive let expression. When we speak of the **free variables** of an expressions, we mean these variables that are free in the expression.

We distinguish between **binding occurrences** and **using occurrences** of a variable in an expression. For instance, in the expression

$$\text{LET } f = \lambda x. fx \text{ IN } f$$

the first occurrences of f and x are binding, and all other occurrences of f and x are using.

The typing rules for expressions observe the binding rules for variables. If $E \vdash e : t$ is derivable, we know that at most the variables that are assigned a type by the environment E are free in e. Hence we know that e is **closed** if $[\,] \vdash e : t$ is derivable for some type t.

Let the letter X denote finite sets of variables. We say that an expression e is **closed in** X if every variable that is free in e is an element of X. It is helpful to consider a derivation system for judgments $X \vdash e$ such that $X \vdash e$ is derivable if and only if e is closed in X. Fig. 4.9.1 gives such a derivation system. We refer to the rules of the system as **binding rules**. The notation X, x stands for the set obtained from X by adding x (i.e., $X \cup \{x\}$), Note that the binding rules can be seen as simplifications of the typing rules, and that the algorithmic reading of the rules is obvious (check whether e is closed in X). Only the rule for variables has a side condition.

$$\frac{}{X \vdash c} \qquad \frac{x \in X}{X \vdash x} \qquad \frac{X \vdash e_1 \qquad X \vdash e_2}{X \vdash e_1 \, o \, e_2} \qquad \frac{X \vdash e_1 \qquad X \vdash e_2}{X \vdash e_1 \, e_2}$$

$$\frac{X \vdash e_1 \qquad X \vdash e_2 \qquad X \vdash e_3}{X \vdash \textsc{if } e_1 \textsc{ then } e_2 \textsc{ else } e_3} \qquad \frac{X \vdash e_1 \qquad X,\, x \vdash e_2}{X \vdash \textsc{let } x = e_1 \textsc{ in } e_2}$$

$$\frac{X,\, x \vdash e}{X \vdash \lambda(x : t).e} \qquad \frac{X,\, f,\, x \vdash e_1 \qquad X,\, f \vdash e_2}{X \vdash \textsc{let rec } f \ (x : t_1) : t_2 = e_1 \textsc{ in } e_2}$$

Figure 4.9.1: Binding rules

Exercise 4.9.1 Give derivations for the judgments $\{y, z\} \vdash (\lambda x.x)\, y$ and $\emptyset \vdash \lambda x.x$. Explain why the judgment $\{x, y\} \vdash z$ is not be derivable.

Exercise 4.9.2 Two expressions are **alpha equivalent** if they are equal up to consistent renaming of local variables. For instance, $\lambda x.x$ and $\lambda y.y$ are alpha equivalent. Consistent renaming of local variables is also known as **alpha renaming**. For each of the following expressions give an alpha equivalent expression such that no variable has more than one binding occurrence.

a) $(\lambda x.x)(\lambda x.x)$

b) $\lambda x.\lambda x.\lambda x.x$

c) $\textsc{let rec } x\, x = x \textsc{ in } x$

4.10 Dynamic Semantics

The dynamic semantics of a programming language defines what effect the execution of a phrase should have. For our language of abstract expressions, the successful execution of an expression should produce a value. One usually refers to the execution of an expression as **evaluation** to emphasize that it will produce a value if it doesn't diverge or lead to an erroneous situation (e.g., division by zero).

In Chapter 1 we explained the evaluation of expressions with a **rewriting model** simplifying expressions by rewriting with defining equations. We now switch to an **environment model** where expressions are evaluated in an environment binding variables to values. The environment model is the standard model for the dynamic semantics of programming languages.

One characteristic aspect of the environment model is the fact that functions appear as values called **closures**. For instance, if we evaluate a lambda expression $\lambda x.e$ in a value environment V, we obtain the closure

$$(x, e, V)$$

consisting of the argument variable x, the expression e, and the value environment V providing the values for the free variables of the abstraction $\lambda x.e$. The name closure is motivated by the fact that a lambda expression becomes a self-contained function once we add an environment providing values for the free variables. Closures don't appear in the rewriting model since there the values for free variables are substituted in as part of rewriting.

Closures for recursive functions take the form

$$(f, x, e, V)$$

where f is the name of the function, x is the argument variable, e is the body of the function, and V is a value environment providing values for the free variables of e other than f and x. Only when the recursive closure is applied to an argument, the bindings for f and x will be added to the environment V.

We describe the evaluation of expressions with a derivation system for **evaluation judgments** of the form

$$V \vdash e \triangleright v$$

saying that in the value environment V the expression e evaluates to the value v. A **value environment** is a finite collection of bindings $x \triangleright v$ mapping variables to values. As for type environments, we assume that value environments are functional.

Values are either values of the base types (numbers, booleans, strings), or closures as described above.

The inference rules for evaluation judgments, commonly called **evaluation rules**, appear in Fig. 4.10.1. The structure of the evaluation rules is similar to the structure of the typing rules. As one would expect, every evaluation rule has an algorithmic reading. For conditionals we have two rules accommodating the two possible branchings. For function applications we also have two rules accommodating separately plain closures and recursive closures. Both rules evaluate the expression of the closure in the environment of the closure updated with a binding of the argument variable to the value serving as actual argument. In the recursive case also a binding of the function variable to the closure is added to enable recursion.

The rule for constants assumes that constants are values, which is fine at the level of abstract syntax. Recall that constants range over

$$\frac{}{V \vdash c \rhd c} \qquad\qquad \frac{(x \rhd v) \in V}{V \vdash x \rhd v}$$

$$\frac{V \vdash e_1 \rhd v_1 \qquad V \vdash e_2 \rhd v_2 \qquad v_1 \; o \; v_2 = v}{V \vdash e_1 \; o \; e_2 \rhd v}$$

$$\frac{V \vdash e_1 \rhd \mathsf{true} \qquad V \vdash e_2 \rhd v}{V \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \rhd v} \qquad \frac{V \vdash e_1 \rhd \mathsf{false} \qquad V \vdash e_3 \rhd v}{V \vdash \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \rhd v}$$

$$\frac{V \vdash e_1 \rhd v_1 \qquad V,\, x \rhd v_1 \vdash e_2 \rhd v}{V \vdash \text{LET } x = e_1 \text{ IN } e_2 \rhd v}$$

$$\frac{}{V \vdash \lambda x.e \rhd (x, e, V)}$$

$$\frac{V \vdash e_1 \rhd (x, e, V') \qquad V \vdash e_2 \rhd v_2 \qquad V',\, x \rhd v_2 \vdash e \rhd v}{V \vdash e_1 \, e_2 \rhd v}$$

$$\frac{V,\, f \rhd (f, x, e_1, V) \vdash e_2 \rhd v}{V \vdash \text{LET REC } f \; x = e_1 \text{ IN } e_2 \rhd v}$$

$$\frac{V \vdash e_1 \rhd v_1 \qquad V \vdash e_2 \rhd v_2 \qquad v_1 = (f, x, e, V') \qquad V',\, f \rhd v_1,\, x \rhd v_2 \vdash e \rhd v}{V \vdash e_1 \, e_2 \rhd v}$$

Figure 4.10.1: Evaluation Rules

numbers, booleans, and strings. For operators we assume that we have a table

$$v_1 \; o \; v_2 = v$$

fixing the results of successful operator applications.

Note the side condition $v_1 = (f, x, e, V')$ appearing in the application rule for recursive closures. We have this side condition for better readability. It can be eliminated by replacing both occurrences of v_1 with the closure (f, x, e, V').

The evaluation rules describe the successful evaluation of expressions at an abstract level omitting details that will be added by an implementation. In particular, the rules don't say what happens when an operator application fails (e.g., division by zero).

We say that an expression e is **evaluable** in an environment V if there is a value v such that the judgment $V \vdash e \rhd v$ is derivable with the evaluation rules.

Here is an example of a derivation of an evaluation judgment:

$$\frac{\overline{[\,] \vdash \lambda x.\lambda y.x \rhd (x,\ \lambda y.x,\ [\,])} \qquad \overline{[\,] \vdash 1 \rhd 1} \qquad \overline{x \rhd 1 \vdash \lambda y.x \rhd (y,\ x,\ [x \rhd 1])}}{[\,] \vdash (\lambda x.\lambda y.x)\,1 \rhd (y,\ x,\ [x \rhd 1])}$$

The above derivation can be written more concisely by omitting material copied from the conclusion to the premises:

$$\frac{\overline{\cdots \rhd (x,\ \lambda y.x,\ [\,])} \qquad \overline{\cdots \rhd 1} \qquad \overline{x \rhd 1 \vdash \lambda y.x \rhd (y,\ x,\ [x \rhd 1])}}{[\,] \vdash (\lambda x.\lambda y.x)\,1 \rhd (y,\ x,\ [x \rhd 1])}$$

Note that the evaluation rules for function applications can yield a value for an application $e_1\,e_2$ only if both subexpressions e_1 and e_2 are evaluable. This is in contrast to conditionals $\text{IF}\ e_1\ \text{THEN}\ e_2\ \text{ELSE}\ e_3$, which may be evaluable although one of the constituents e_2 and e_3 is not evaluable. This relates to the problem underlying exercise 1.13.1.

Type Safety

The evaluation rules do not mention types at all. Thus we can execute ill-typed expressions. This may lead to erroneous situations that cannot appear with well-typed expressions, for instance a function application $e_1\,e_2$ where e_1 evaluates to a number, or an addition of a closure and a string. In contrast, if we evaluate a well-typed expression, this kind of erroneous situations are impossible. This fundamental property is known as **type safety**. Type safety provides for more efficient execution of programming languages, and also makes it easier for the programmer to write correct programs. OCaml is designed as a type-safe language.

Exercise 4.10.1 For the following expressions e find values v such that $[\,] \vdash e \rhd v$ is derivable. Draw the derivations.

a) $(\lambda x.x)1$

b) $(\lambda x.\lambda y.x)1$

c) $(\lambda f.\lambda x.fx)\,(\lambda x.x)$

d) $\text{LET REC}\ fx = x\ \text{IN}\ f$

e) $(\lambda y.\ \text{LET REC}\ fx = x\ \text{IN}\ f)\,5$

Exercise 4.10.2 Give evaluation rules for declarations and programs. Hint: Use judgments $V \vdash D \Rightarrow V'$ and $V \vdash P \Rightarrow V'$ and model programs with the grammar $P ::= \emptyset \mid D\ P$ ($\emptyset$ is the empty program).

5 Mini-OCaml Interpreter

In this chapter we describe a programming project in which you will realize a complete Mini-OCaml interpreter. This is the first time we ask you to write and debug a program that altogether has a few hundred lines of code. As you will find out, this makes a dramatic difference to writing and debugging the few-line programs we have been considering so far.

Debugging larger programs can be very difficult. The method working best in practice is having a modular design of the program where each module can be debugged by itself. We speak of *divide and conquer*.

To find bugs you need to come up with tests, and once you have found a bug, you usually need to refine the test showing the bug in order that you find the place in the program that is responsible for the bug.

For the project, we provide you with a modular design featuring the following components:

- A lexer translating strings into lists of tokens.
- A parser translating lists of tokens into abstract expressions.
- A type checker computing the type of an expression in an environment.
- An evaluator computing the value of an expression in an environment.

Each of the components can be written and debugged by itself, which is the best one can hope for. The glue between the components is provided by constructor types for expressions, types, and tokens. There is also a constructor type for the values of Mini-OCaml covering plain and recursive closures.

We assumes you are familiar with Chapter 4 on syntax and semantics.

5.1 Expressions, Types, Environments

Our starting point is the abstract grammar for the types and expressions of Mini-OCaml: The letter c ranges over *constants* which we choose to be booleans or integers, and the letters x and f range over *variables* which we choose to be strings.

$$
\begin{array}{rcl}
t & ::= & bool \;\mid\; int \;\mid\; t_1 \to t_2 \\
o & ::= & + \;\mid\; - \;\mid\; \cdot \;\mid\; \leq \\
e & ::= & x \;\mid\; c \;\mid\; e_1\, o\, e_2 \;\mid\; e_1\; e_2 \qquad\qquad\qquad (c : \mathbb{B} \mid \mathbb{Z}) \\
 & \mid & \text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3 \\
 & \mid & \lambda x : t.e \\
 & \mid & \text{LET } x = e_1 \text{ IN } e_2 \\
 & \mid & \text{LET REC } f\ (x : t_1) : t_2 = e_1 \text{ IN } e_2
\end{array}
$$

Figure 5.1.1: Grammar for the abstract syntax of Mini-OCaml types and expressions

An important design decision is to have lambda expressions and recursive let expressions both in untyped and typed form.

We realize the abstract grammar in OCaml with a system of constructor types:

```
type ty  = Bool | Int | Arrow of ty ∗ ty
type con = Bcon of bool | Icon of int
type op  = Add | Sub | Mul | Leq
type var = string
type exp = Var of var | Con of con
         | Oapp of op ∗ exp ∗ exp
         | Fapp of exp ∗ exp
         | If of exp ∗ exp ∗exp
         | Lam of var ∗ ty ∗ exp
         | Let of var ∗ exp ∗ exp
         | Letrec of var ∗ var ∗ ty ∗ ty ∗ exp ∗ exp
```

Note that *var* is not a constructor type but simply a second name for *string* that we introduce for readability.

Exercise 5.1.1 Write a tail-recursive faculty function $n!$ in Mini-OCaml. First write the expression in OCaml, then translate it into the abstract syntax of Mini-OCaml in OCaml.

5.2 Type Checker

The type checker needs type environments that map variables to types. To implement an environment, we use a key-value map as discussed in Section 2.16.

```
type 'a env = (var ∗ 'a) list
type 'a tenv = ty env
```

We realize the type checker for Mini-OCaml with a function

$$check : \ tenv \to exp \to ty$$

that checks whether an expression is well-typed in an environment. If this is the case, *check* yields the unique type of the expression, otherwise it throws an exception. Expressions containing untyped lambda or recursive let expressions count as ill-typed. The type checker matches on the given expression and for each variant follows the algorithmic reading of the respective typing rule (Fig. 4.7.1):

```
let rec check env e : ty = match e with
  | Var x -> ?···?
  | Con (Bcon b) -> Bool
  | Con (Icon n) -> Int
  | Oapp (o,e1,e2) -> check_op o (check env e1) (check env e2)
  | Fapp (e1,e2) -> check_fun (check env e1) (check env e2)
  | If (e1,e2,e3) -> ?···?
  | Lam (x,t,e) -> Arrow (t, check (update env x t) e)
  | Let (x,e1,e2) -> check (update env x (check env e1)) e2
  | Letrec (f,x,t1,t2,e1,e2) -> ?···?
```

Note that we are using helper functions for operator and function applications for better readability.

5.3 Evaluator

For the evaluator we first define a constructor type providing the values of Mini-OCaml as OCaml values:

```
type va = Bval of bool | Ival of int
        | Closure of var ∗ exp ∗ va env
        | Rclosure of var ∗ var ∗ exp ∗ va env
```

We realize the evaluator with a function

$$eval : \ venv \to exp \to value$$

that evaluates an expression in an environment. The evaluator matches on the given expression and for each variant follows the algorithmic reading of the respective evaluation rule (Fig. 4.10.1):

```
let rec eval env e : va = match e with
  | Var x -> ?···?
  | Con (Bcon b) -> Bval b
  | Con (Icon n) -> Ival n
  | Oapp (o,e1,e2) -> eval_op o (eval env e1) (eval env e2)
  | Fapp (e1,e2) -> eval_fun (eval env e1) (eval env e2)
  | If (e1,e2,e3) -> ?···?
  | Lam (x,_,e) -> Closure (x,e,env)
  | Let (x,e1,e2) -> eval (update env x (eval env e1)) e2
  | Letrec (f,x,_,_,e1,e2) ->  ?···?
and eval_fun v1 v2 = match v1 with ?···?
```

As with the type checker, we use helper functions for operator and function applications for better readability. This time the helper function for function applications is mutually recursive with the master evaluation function.

5.4 Lexer

The lexer for Mini-OCaml needs work, mainly because it needs to recognize identifiers (i.e., variables) and number constants. The first step consists in declaring a constructor type for tokens:

```
type const   = BCON of bool | ICON of int
type token   = LP | RP | EQ | COL | ARR | ADD | SUB | MUL | LEQ
             | IF | THEN | ELSE | LAM | LET | IN | REC
             | CON of const | VAR of string | BOOL | INT
```

The tokens `LP`, `RP`, `EQ`, `COL`, `ARR`, and `LAM` are realized with the strings `"("`, `")"`, `"="`, `":"`, `"->"`, and `"fun"`.

We realize the lexer with 5 mutually tail-recursive functions:

```
let lex s : token list =
  let get i = String.get s i in
  let getstr i n = String.sub s (i-n) n in
  let exhausted i = i >= String.length s in
  let verify i c = not (exhausted i) && get i = c in
  let rec lex i l =
    if exhausted i then List.rev l
    else match get i with
      | '+' -> lex (i+1) (ADD::l)
      ?···?
      | c when whitespace c -> lex (i+1) l
      | c when digit c -> lex_num (i+1) (num c) l
      | c when lc_letter c -> lex_id (i+1) 1 l
      | c -> failwith "lex: illegal character"
  and lex_num i n l = ?···?
  and lex_num' i n l = lex i (CON (ICON n)::l)
  and lex_id i n l =  ?···?
  and lex_id' i n l = match getstr i n with
    | "if" -> lex i (IF::l)
    ?···?
    | s -> lex i (VAR s::l)
  in lex 0 []
```

There is some cleverness in this design. We avoid considering auxiliary strings by accessing the given string by position using *String.get* and by using *String.sub* to extract an identifier identified by its starting position and length. The constants *true* and *false* and keywords like *if* are first recognized as identifiers and then transformed into the respective constant or keyword tokens.

Digits and letters are recognized by using the function *Char.code* and exploiting the fact that the codes of the respective characters follow each other continuously in a canonical order (there are separate intervals for digits, lower case letters, and upper case letters).

Following OCaml, an identifier must start with a lower case letter and can then continue with digits, lower and upper case letters, and the special characters '_' (underline) and '′' (quote).

Note the use of patterns with **when conditions** in the declaration of the parsing function *lex*. This is a syntactical convenience translating into nested conditionals.

Note that *lex_id′* is the function where you can change the identifier-like keywords of the concrete syntax. For instance, you may replace the keyword *fun* with the keyword *lam* or *lambda*. Such a change will not show at the level of tokens and hence will not be visible in the parser.

We remark that the lexer follows the *longest munch rule* when it reads identifiers and numbers; that is, it reads as many characters as are possible for an identifier or number. Thus `" ifx "` is recognized as a single variable token *Var* `"ifx"` rather than the keyword token *If* followed by the variable token *Var* `"x"`.

Exercise 5.4.1 Rewrite the function *lex* so that no patterns with when conditions are used. Use nested conditionals instead.

Exercise 5.4.2 Extend the lexer with comments.

5.5 Parsing

The process of translating string representations of syntactic objects into tree representations is known as *parsing.* We will focus on a parsing method called *recursive descent* which works well for **prefix** linearizations. For grammars with **infix** expressions[1], recursive descent is less suitable. Therefore we will present another technique that is called **operator precedence parsing**. Let us first discuss recursive descent parsing and then go to operator precedence parsing.

5.5.1 Recursive Descent Parsing

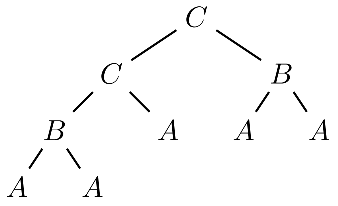
In the discussion in Section 4.5, we have seen that the job of a parser is to reconstruct the tree structure of a syntactic object from its linearization. By linearization we mean the derivation mechanism by which sequences of words are derived from a grammar. A kind of linearization that makes this reconstruction easy is called **prefix** linearization.

A first prerequisite for prefix linearizations is that each rule has a terminal that uniquely identifies the rule. A second prerequisite is that each rule starts with that unique terminal. A prefix linearization linearizes a tree by emitting terminals and linearizing successor trees in the order in which they appear in the rule. For example, consider the grammar:

$$e ::= A \mid B\ e\ e \mid C\ e\ e$$

The prefix linearization of the tree

[1] Infix means that the operator that operates on two expressions is in between the expressions, like in $1 + 2$.



is given by

$$C\ C\ B\ A\ A\ A\ B\ A\ A$$

Now, parsing a prefix linearization is simple. For each non-terminal e, we create a **parse function** `parse` that gets the remaining token sequence as input and returns the pair of the tree and the remaining token sequence after parsing the tree. This function inspects the first token in the input, and because each rule as a unique identifying terminal, it knows which rule to apply. The rule is processed in order as follows: If the current item is a terminal, we check that the current token in the input is equal to that terminal. If the current item is a non-terminal, we simply call the parse function for that non-terminal and obtain a sub-tree and a remaining token sequence. After all items of a rule have been processed, we are ready to return the tree and the remaining token sequence.

Consider the following OCaml implementation of a recursive descent parser for the simple ABC grammar above:

```ocaml
type tree = A | B of tree * tree | C of tree * tree
let rec parse ts = match ts with
  | 'A' :: ts -> A, ts
  | 'B' :: ts -> let l, ts = parse ts in
                 let r, ts = parse ts in
                 B (l, r), ts
  | 'C' :: ts -> let l, ts = parse ts in
                 let r, ts = parse ts in
                 C (l, r), ts
  | _ -> failwith "unexpected character"
```

And we have that

```ocaml
parse (explode "CCBAAABAA")
```

yields the tree above where *explode* is a function that turns a string into a list of characters.

Unfortunately, the grammar of the abstract syntax of Mini-OCaml in Fig. 5.1.1 does not fulfil the second requirement because the rules for binary expression $e_1\, o\, e_2$ and function application $e_1\, e_2$ do not start with a unique terminal.

However, if we remove these two rules, we obtain a grammar that fulfils the requirements. Note that the concrete syntax contains more ornaments than the abstract syntax. We can easily write a recursive descent parser for this sub-grammar:

```
let rec parse_expr ts = match ts with
  | VAR s :: ts -> Var s, ts
  | CON c :: ts -> Con c, ts
  | LP    :: ts -> let e,  ts = parse_expr ts in
                   let     ts = expect RP ts in
                   e, ts
  | IF    :: ts -> let e1, ts = parse_expr ts in
                   let     ts = expect THEN ts in
                   let e2, ts = parse_expr ts in
                   let     ts = expect ELSE ts in
                   let e3, ts = parse_expr ts in
                   If (e1, e2, e3), ts
  | LET   :: ts -> let x,  ts = expect_var ts in
                   let     ts = expect EQ ts in
                   let e1, ts = parse_expr ts in
                   let     ts = expect IN ts in
                   let e2, ts = parse_expr ts in
                   Let (x, e1, e2), ts
  ...
```

This parser uses two functions `expect` and `expect_var` that check whether the next token in the input is the expected token or signal an error.

```
exception ExpectedVar of token list
exception ExpectedToken of token * token list

let expect_var = match ts with
  | VAR s :: ts -> s, ts
  | _ -> raise (ExpectedVar ts)

let expect t = match ts with
  | t' :: ts when t = t' -> ts
  | _ -> raise (ExpectedToken (t, ts))
```

5.5.2 Operator Precedence Parsing

When linearizing expressions using infix notation, there arises the ambiguity which operator binds which operands. Consider the following example:

$$a \quad + \quad b \quad * \quad c \quad + \quad d \quad \leq \quad e$$

Does a belong to the left $+$ or to the $*$? This question is resolved by the **precedence** of the operators. $*$ has a higher precedence than $+$, so a is an operand of $*$.

Operator precedence parsing is a technique to parse binary expressions in infix notation based on the concept of **operator precedence** or **binding power** (both terms are used interchangeably). We assign each operator a **left binding power** and a **right binding power**. This binding power determines how strong an operator "pulls" on its operands. For Mini-OCaml the binding powers are as follows:

Operators	Left Binding Power	Right Binding Power
function application	6	7
*	4	5
+ -	2	3
<=	1	1

The binding powers encode two things: First, higher binding express higher precedence. Second, different left and right binding powers express associativity. The comparison operator <= is not associative. Multiplication, addition, and subtraction are left associative which can be seen from this example:

$$_0 \quad a \quad {}_2{+}_3 \quad b \quad {}_2{+}_3 \quad c$$

The left addition pulls b stronger than the right one because its right binding power is higher than the left binding power of the right addition. Therefore, the expression reads as

$$(2+3)+4$$

When annotating the left and right binding powers to the operators, the example expression from above looks like this:

$$_0 \quad a \quad {}_2{+}_3 \quad b \quad {}_4{*}_5 \quad c \quad {}_2{+}_3 \quad d \quad {}_1{\leq}_1 \quad e$$

The structure of the expression is now clear and can be recovered recursively: First, the multiplication "pulls" its operands 3 and 4 stronger than the two additions. Therefore, the expression reads as

$$_0 \quad a \quad {}_2{+}_3 \quad (b * c) \quad {}_2{+}_3 \quad d \quad {}_1{\leq}_1 \quad e$$

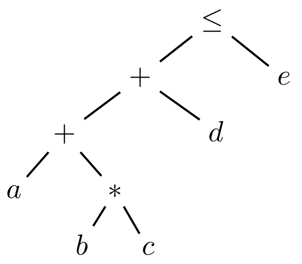
Then, the left addition pulls $(b * c)$ stronger than the right addition and its left operand 2 stronger than the initial precedence 0. Therefore, the expression reads as

$$(a + (b * c)) \quad {}_2{+}_3 \quad d \quad {}_1{\leq}_1 \quad e$$

Finally, the remaining addition pulls its right operand 5 stronger than the comparison operator. Therefore, the expression reads as

$$((a + (b * c)) + d) \leq e$$

which coincides with the following abstract syntax tree:



The operator precedence parser constructs the abstract syntax tree of such an expression from left to right recursing over the precedence level, starting with the lowest precedence level 0, and returning the abstract syntax tree of the expression at the current precedence level.

In the example above, the operands to the binary operators have been variable names. In general, they can be arbitrary non-binary expressions, such as constants, if-then-elses, let expressions, and so on. Let us call such non-binary expressions now **simple expressions**. These simple expressions are parsed by a separate function, typically using recursive descent parsing.

The precedence parser is a recursive function that obtains a left-hand side expression l and a precedence level p as input and returns the abstract syntax tree of of the expression **up to** the first operator with a left binding power less than p. Its only action is to examine the next token t in the input and distinguish the following three cases:

1. The next token is not a binary operator. Then, the expression has ended and we return the expression l.
2. The next token is a binary operator o with left and right binding powers p_l and p_r. There are two sub-cases:
   a) $p_l < p$: the operator has a lower precedence than the current precedence level p. Therefore, the operator pulls its operands weaker than the current precedence level. Hence, we return the expression l.
   b) $p_l > p$: the operator has a higher precedence than the current precedence level p. Therefore, the operator pulls its operands stronger than the current precedence level. We consume the operator token and parse another **simple** expression l'. We recurse

with l' and precedence level p_r eventually obtaining the abstract syntax tree for the right-hand side expression r. Note that by construction, all operators in r have a binding power at least as high as p_r. Then, we continue parsing on level p with the left-hand side expression $l\,o\,r$.

The case $p = p_l$ cannot occur because p is always set to a right binding power and the right binding powers and the left binding powers are disjoint.

Let us write an execution trace of this algorithm (let's call it *pp*) for the example expression above. Tokens are in `typewriter` font. The first argument is the current precedence level and the second one represents the (partial) abstract syntax tree build so far.

$$\begin{aligned}
& pp\ 0\ \ (a)\ \ [\texttt{+}, \texttt{b}, *, \texttt{c}, \texttt{+}, \texttt{d}, \texttt{<=}, \texttt{e}] \\
= \;& pp\ 0\ \ (a + (pp\ 3\ b\ [*, \texttt{c}, \texttt{+}, \texttt{d}, \texttt{<=}, \texttt{e}]))\ \ \ldots \\
= \;& pp\ 0\ \ (a + (pp\ 3\ b * ((pp\ 4\ c\ [\texttt{+}, \texttt{d}, \texttt{<=}, \texttt{e}])))\ \ \ldots \\
= \;& pp\ 0\ \ (a + (pp\ 3\ b * c\ [\texttt{+}, \texttt{d}, \texttt{<=}, \texttt{e}]))\ \ \ldots \\
= \;& pp\ 0\ \ (a + (b * c))\ [\texttt{+}, \texttt{d}, \texttt{<=}, \texttt{e}] \\
= \;& pp\ 0\ \ ((a + (b * c)) + pp\ 3\ d\ [\texttt{<=}, \texttt{e}])\ \ \ldots \\
= \;& pp\ 0\ \ ((a + (b * c)) + d)\ \ [\texttt{<=}, \texttt{e}] \\
= \;& ((a + (b * c)) +) \leq e
\end{aligned}$$

The following OCaml code implements the algorithm outlined above:

```
let parse_binary parse_simple parse_op ts =
  let rec pp p (l, ts) =
    match parse_op ts with
      | None -> (l, ts)
      | Some (op, lp, rp, ts') -> if lp < p
        then (l, ts)
        else
          let r, ts = pp rp (parse_simple ts')
          in pp p (op l r, ts)
in pp 0 (parse_simple ts)
```

The function

```
  parse_simple : token list -> exp * token list
```

parses a simple expression and can for example be implemented using recursive descent parsing. The function

```
  parse_op : token list -> ((exp -> exp -> exp) * int * int *
      token list) option
```

examines the next token, checks if it is a binary operator and if that is the case, returns a function to construct an abstract syntax tree for the operator given a left and right expression, the left and right binding powers of the operator, and the remaining token sequence. Otherwise it returns none. For the arithmetic operators of Mini-OCaml `parse_op` can be implemented as follows:

```
let parse_arithmetic_op ts =
  let create op l r = Oapp (op, l, r) in match ts with
  | LEQ :: ts -> Some (create Leq, 1, 1, ts)
  | ADD :: ts -> Some (create Add, 2, 3, ts)
  | SUB :: ts -> Some (create Sub, 2, 3, ts)
  | MUL :: ts -> Some (create Mul, 4, 5, ts)
  | _ -> None
```

Exercise 5.5.1 Incorporate the binary function application operator into the precedence parser. Hint: you just need to extend `parse_op`.

Exercise 5.5.2 Use `parse_binary` to implement a parser for the type language of Mini-OCaml.

6 Running Time

Often there are several algorithms solving a given computational task, and these algorithm may differ significantly in their running times. A typical example is sorting of lists. Insertion sort, the sorting algorithm we have already seen, requires quadratic worst-case time. We will see a substantially faster algorithm known as merge sort in this chapter whose running time is almost linear.

It turns out that we can define the running time of an expression as the number of function calls needed for its execution. This abstract and technology-independent notion of running time provides an excellent tool for distinguishing between faster and slower algorithms.

6.1 The Idea

Given a specification of a computational task, we can write different functions satisfying the specification. In case the functions are based on different algorithms, the execution times of the functions may differ substantially. There is an abstract notion of asymptotic running time predicting the execution times of functions. Prominent examples of asymptotic running times are

$O(1)$	constant time
$O(\log n)$	logarithmic time
$O(n)$	linear time
$O(n \log n)$	linearithmic time
$O(n^2)$	quadratic time
$O(2^n)$	exponential time

A function that has linear running time has the property that the execution time grows at most linearly with the argument size. Similarly, functions that have logarithmic or quadratic or exponential running time have the property that the execution time grows at most logarithmically or quadratically or exponentially with the argument size. Linearithmic time is almost like linear time and behaves the same in practice. Finally, the execution time of a function that has constant running time is bounded for all arguments by a fixed constant. In practice, constant, linear, and linearithmic running time is what one looks for, quadratic running times signals that there will be performance problems for larger

Size	Running time function			
n	linear (n)	quadratic (n^2)	cubic (n^3)	exponential (2^n)
	Execution time (assuming 10^9 function calls per second)			
10^3	10^{-6} seconds	10^{-3} seconds	1 second	forever
10^4	10^{-5} seconds	10^{-1} seconds	20 minutes	forever
10^5	10^{-4} seconds	10 seconds	10 days	forever
10^6	10^{-3} seconds	20 minutes	30 years	forever
10^7	10^{-2} seconds	1 day	forever	forever

Figure 6.1.1: Execution times depending on argument size.

arguments, and exponential running time signals that there are arguments of reasonable size for which the execution time is too high for practical purposes. Fig. 6.1.1 shows execution times for different running times and argument sizes and Fig. 6.1.2 shows plots of common running time functions.

The running times for list sorting algorithms are interesting. We will see that insertion sort has quadratic running time, and that there is a sorting algorithm (merge sort) with linearithmic running time that is much faster in practice.

Logarithmic running time $O(\log n)$ is much faster than linear time. The standard example for logarithmic running time is binary search, an algorithm for searching ordered sequences we will discuss in this chapter. Logarithmic running time occurs if the input size is halved at each recursion step, and only a constant number of functions calls is needed to prepare a recursion step.

A good measure of the abstract running time of an expression is the number of function calls needed for its execution. Expressions whose execution doesn't involve function calls receive the abstract running time 0. As an example, we consider a list concatenation function:

$$\begin{aligned} [\,] \mathbin{@} l_2 &:= l_2 \\ (x :: l_1) \mathbin{@} l_2 &:= x :: (l_1 \mathbin{@} l_2) \end{aligned}$$

The defining equations tell us that the running time of a concatenation $l_1 \mathbin{@} l_2$ only depends on the length of the list l_1. In fact, from the defining equations of the concatenation function @ we can obtain a recursive running time function

$$\begin{aligned} r(0) &:= 1 \\ r(n+1) &:= 1 + r(n) \end{aligned}$$

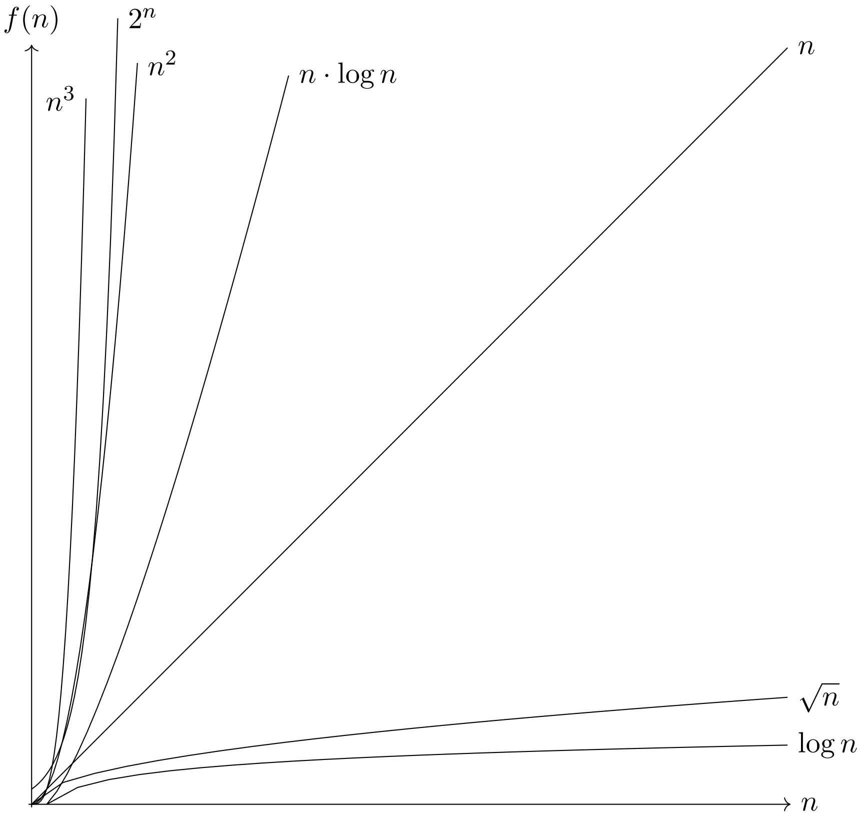


Figure 6.1.2: Plots of common running time functions.

giving us the running time for every call $l_1 @ l_2$ where l_1 has length n. A little bit of discrete mathematics gives us an explicit characterization of the running time function:

$$r(n) \;=\; n+1$$

We can now say that the list concatenation function @ has linear running time in the length of the first list.

We remark that we distinguish between function applications and function calls. A *function application* is a syntactic expression $e\, e_1 \ldots e_n$. A *function call* is a tuple $(v, v_1, \ldots, v_n)$ of values that is obtained by evaluating the expressions of a function application.

6.2 List Reversal

We will now look at two functions for list reversal giving us two interesting examples for abstract running times. We start with a tail-recursive function

$$\begin{aligned} \mathit{rev\_append}\ []\ l_2 &\ :=\ l_2 \\ \mathit{rev\_append}\ (x :: l_1)\ l_2 &\ :=\ \mathit{rev\_append}\ l_1\ (x :: l_2) \end{aligned}$$

that has linear running time in the length of the first argument (following the argumentation we have seen for list concatenation). We can now reverse a list l with a function call ($\mathit{rev\_append}\ l\ []$) taking running time linear in the length of l. We remark at this point that *rev_append* gives us an optimal list reversal function that cannot be further improved.

Another list reversal function we call **naive reversal** is

$$\begin{aligned} \mathit{rev}\ [] &\ :=\ [] \\ \mathit{rev}\ (x :: l) &\ :=\ \mathit{rev}\,(l)\ @\ [x] \end{aligned}$$

Practical experiments with an OCaml interpreter show that the performance of naive list reversal is much worse than the performance of tail-recursive list reversal. While *rev_append* will yield the reversal of a list of length 20000 instantaneously, the naive reversal function *rev* will take considerable time for the same task (go to length 100000 to see a more impressive time difference).

The above experiments require an interpreter that can handle deeply nested function calls. While this is the case for the native OCaml interpreter, the browser-based TryOCaml interpreter severely limits the height of the call stack (about 1000 function calls at the time of writing). Thus the above experiments require a native OCaml interpreter. We remark that function calls in tail position (as is the case with tail recursion) don't require allocation on the call stack. In fact, TryOcaml can easily execute the tail-recursive function *rev_append* for lists of length 100000. On the other hand, TryOcaml aborts the execution of the naive list reversal function *rev* already for lists of length 3000.

We return to the discussion why naive list reversal is so much slower than tail-recursive list reversal. The answer simply is that naive list reversal has quadratic running time while tail-recursive list reversal has linear runtime. For instance, when we reverse a list of length 10^4, naive reversal roughly requires 100 million function calls while tail-recursive reversal only needs ten thousand function calls (think of money to get a feel for the numbers).

Tail-recursive list reversal has running time $n+1$ for lists of length n, which can be shown with the same method we used for list concatenation. For naive reversal we have to keep in mind that list concatenation ist not an operation (as suggested by the symbol @) but a function. This means we have to count the function calls needed for concatenation for every recursive call of *rev*. This gives us the recursive definition of the running time function for *rev*:

$$\begin{aligned} r(0) &:= 1 \\ r(n+1) &:= 1 + r(n) + (n+1) \end{aligned}$$

The term $(n + 1)$ is the number of recursion steps a concatenation $\mathit{rev}\,(l) \mathbin{@} [x]$ takes if l has length n. We make use of the fact that *rev* leaves the length of a list unchanged. We now have a recursive definition of the exact runtime function for *rev*. With standard techniques from discrete mathematics one can obtain an equation characterizing $r(n)$ without recursion:

$$r(n) = (n+1)(n+2)/2$$

One speaks of *recurrence solving*. We will not explain this topic here. For practical purposes it is often convenient to use one of the recurrence solvers in the Internet.[1]

Exercise 6.2.1 Define a linear time function concatenating two lists using only tail recursion. Hint: Exploit the equations $l_1 \mathbin{@_{\mathrm{r}}} l_2 = \mathit{rev}\ l_1 \mathbin{@} l_2$ and $\mathit{rev}\,(\mathit{rev}\ l) = l$.

6.3 Insertion Sort

Recall insertion sort from Section 2.12:

$$\begin{aligned} &\mathit{insert} : \ \mathbb{Z} \to \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z}) \\ &\qquad \mathit{insert}\ x\ [] \ := \ [x] \\ &\mathit{insert}\ x\ (y :: l) \ := \ x :: y :: l \qquad \text{if } x \le y \\ &\mathit{insert}\ x\ (y :: l) \ := \ y :: \mathit{insert}\ x\ l \quad \text{if } x > y \end{aligned}$$

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

The structure of insertion sort is similar to naive list reversal. However, there is the new aspect that an insertion may cost between 1 and $n+1$

[1] We recommend https://www.wolframalpha.com. Entering the two defining equations into the solver will suffice to obtain the closed formula characterization of $r(n)$.

function calls depending on where the insertion in a list of length n takes place. We say that list insertion has constant running time in the **best case** and linear running time in the **worst case**. We also say that list insertion has linear **worst-case running time**.

We now look at the running time of *isort*. We have linear best-case running time $(2n + 1)$ if the input list is in ascending order $(x_1 \leq x_2 \leq \cdots)$ and quadratic worst-case running time $((n+1)(n+2)/2)$ if the input list is in strictly descending order $(x_1 > x_2 > \cdots)$. The running time function for the worst case is identical with the running time function for naive list reversal.

The quadratic worst-case running time shows in practice. While a list of length 20000 is sorted instantaneously if it is in ascending order, sorting takes considerable time if the list is in strictly descending order.

6.4 Merge Sort

Because of its quadratic worst-case running time, insertion sort is not a good sorting function in practice. It turns out that there is a much better sorting algorithm called *merge sort* that in practice has almost linear running time in the length of the input list.

Merge sort first splits the input list into two lists of equal length (plus/minus one). It then sorts the two sublists by recursion and merges the two sorted lists into one sorted list. Both splitting and merging can be carried out in linear time in the length of the input list. The question now is how often merge sort has to recurse. It turns out that the recursion depth (see Section 6.8) of merge sort is logarithmic in the length of the input length. For instance, if we are given an input list of length 2^{20} (about one million), the recursion depth will be 20, since the length of the lists to be sorted decreases exponentially: $2^{20}, 2^{19}, 2^{18}, 2^{17}, 2^{16}, 2^{15}, \dots$.

We formulate merge sort in OCaml as follows:

```
let rec split l l1 l2 = match l with
  | [] -> (l1,l2)
  | [x] -> (x::l1,l2)
  | x::y::l -> split l (x::l1) (y::l2)

let rec merge l1 l2 = match l1, l2 with
  | [], l2 -> l2
  | l1, [] -> l1
  | x::l1, y::l2 when x <= y -> x :: merge l1 (y::l2)
  | x::l1, y::l2  -> y :: merge (x::l1) l2

let rec msort l = match l with
```

```
  | x::y::l -> let (l1,l2) = split l [x] [y] in
               merge (msort l1) (msort l2)
  | l -> l
```

It is clear that the running time of *split* for an input list of length n is at most $n+1$ (for larger n it is at most $1+n/2$). The running time of *merge* is $1+n_1+n_2$ in the worst case where n_1 and n_2 are the lengths of the input lists l_1 and l_2. Thus the worst-case running time of *msort* is linear in the length of the input list (for the merge and split steps) plus twice the running time of *msort* for input lists of half the length:

$$r(n) := 1 + r(\frac{n}{2}) + 2n$$

Solving this recurrence results in linearithmic running time $O(n \log n)$ for *msort* and input lists of length n.

Note that in the above code *split* is tail-recursive but *merge* is not. A tail-recursive variant of *merge* is straightforward:

```
let rec merge l1 l2 l = match l1, l2 with
  | [], l2 -> List.rev_append l l2
  | l1, [] -> List.rev_append l l1
  | x::l1, y::l2 when x <= y -> merge l1 (y::l2) (x::l)
  | x::l1, y::l2 -> merge (x::l1) l2 (y::l)
```

With the tail-recursive merge function only the recursion depth of *msort* remains, which is logarithmic in the length of the input list.

Exercise 6.4.1 Check with an interpreter that the execution time of *msort* is moderate even for lists of length 10^5. Convince yourself that the ordering of the input list doesn't matter.

6.5 Binary Search

Suppose we have a function $f : \mathbb{N} \to \mathbb{N}$ and want to check whether the sequence

$$f(0),\ f(1),\ f(2),\ \ldots$$

takes the value x for some k (that is, $f(k) = x$). Without further information about f, we can perform a linear search for the first k such that $f(k) = x$.

We can search faster if we assume that f is *increasing*:

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

We can then apply a search technique called **binary search**. Binary search modifies the problem such that, given l, r, and x, we look for a position k in the sequence

$$f(l) \leq \cdots \leq f(r)$$

such that $l \leq k \leq r$ and $f(k) = x$. Binary search proceeds as follows:

1. If $r < l$, the number k we are looking for does not exist.
2. If $l \leq r$, we determine $m = (l + r)/2$ and consider three cases:
   a) If $f(m) = x$, m is the number we are looking for.
   b) If $f(m) < x$, we continue the search in the interval $(m + 1, r)$.
   c) If $f(m) > x$, we continue the search in the interval $(l, m - 1)$.

We note that on recursion the size of the interval to be searched is halved. Thus, if we start with an interval of length 2^n, we have at most $n+1$ recursion steps until we terminate. In other words, the worst-case running time of binary search measured in recursion steps is logarithmic in the length of the interval to be searched. This is in contrast to linear search, where the worst-case running time measured in recursion steps is linear in the length of the interval to be searched.

We realize binary search with an OCAML function

$$\mathit{find} : \ \forall\alpha.\ (\mathit{int} \to \alpha) \to \alpha \to \mathit{int} \to \mathit{int} \to \mathcal{O}(\mathit{int})$$

declared as follows:

```
let find f x l r =
  let rec aux l r =
    if r < l then None
    else let m = (l+r)/2 in
      let y = f m in
      if x = y then Some m
      else if x < y then aux l (m-1) else aux (m+1) r
  in aux l r
```

Note that the helper function *aux* is tail recursive.

In contrast to linear search, binary needs an upper bound r. If we work with machine numbers, we may simply use the largest machine integer *max_int* as an upper bound. If we consider the problem mathematically, we fist need to find an upper bound r such that $x \leq f(r)$. If such an r exists, we may find it quickly with an **exponential search**. We start with a large number k and double k until $x \leq f(k)$. If f is strictly increasing $(f(0) < f(1) < f(2) < \cdots)$, we have $x \leq f(x)$ and can hence use x as an upper bound.

Exercise 6.5.1 The binary search function can be simplified if it returns booleans rather than options. Declare a binary search function checking whether f yields a given value for some number in a given interval. Use the lazy boolean connectives.

Exercise 6.5.2 Binary search can be used for inversion of strictly increasing functions.

a) Declare a function *square* deciding whether an integer $x \geq 0$ is a square number (that is, $x = n^2$ for some n). The worst-case running time of *square* x should be logarithmic in x.

b) Declare a function *sqrt* that given an integer $x \geq 0$ computes the largest n such that $n^2 \leq x$. The worst-case running time of *sqrt* x should be logarithmic in x.

c) Declare a function *inv* that given a strictly increasing function $f : \mathbb{N} \to \mathbb{N}$ and an integer $x \geq 0$ computes the largest n such that $f(n) \leq x$. The worst-case running time of *inv* f x should be logarithmic in x if f has constant running time.

6.6 Trees

Consider the definition of a simple binary tree:

$$\mathit{tree} \;::=\; A \;\mid\; B(\mathit{tree}, \mathit{tree})$$

Two essential notions for trees in general and our binary trees in particular are *size* and *depth*:

$$\begin{aligned} \mathit{size}\ A &:= 1 \\ \mathit{size}\ (B\,(t_1\,t_2)) &:= 1 + \mathit{size}\ t_1 + \mathit{size}\ t_2 \end{aligned}$$

$$\begin{aligned} \mathit{depth}\ A &:= 0 \\ \mathit{depth}\ (B\,(t_1\,t_2)) &:= 1 + \max\,(\mathit{depth}\ t_1)\,(\mathit{depth}\ t_2) \end{aligned}$$

Note that the function *size* computes its own running time; that is, the running time of *size* for a tree t is $\mathit{size}\,(t)$.

Here is a function $\mathit{mtree} : \mathbb{N} \to \mathit{tree}$ that yields binary tree of depth n that has maximal size:

$$\begin{aligned} \mathit{mtree}\,(0) &:= A \\ \mathit{mtree}\,(n+1) &:= B\,(\mathit{mtree}\,(n), \mathit{mtree}\,(n)) \end{aligned}$$

The running time of *mtree* is

$$\begin{aligned} r(0) &:= 1 \\ r(n+1) &:= 1 + r(n) + r(n) \end{aligned}$$

A recurrence solver yields the explicit characterization

$$r(n) \;=\; 2^{n+1} - 1$$

which tells us that *mtree* has exponential running time $O(2^n)$.

It is straightforward to rewrite *mtree* such that the running time becomes linear:

$$\begin{aligned} \mathit{mtree}\,(0) \;&:=\; A \\ \mathit{mtree}\,(n+1) \;&:=\; \textsc{let}\ \ t = \mathit{mtree}\,(n)\ \ \textsc{in}\ \ B\,(t,t) \end{aligned}$$

The insight is that a maximal tree for a given depth has two identical subtrees. Thus it suffices to compute the subtree once and use it twice.

For maximal trees the running time of *size* is exponential in the depth of the input tree. Consequently, the running time of $\lambda n.\, \mathit{size}\,(\mathit{mtree}\, n)$ is exponential in n even if the linear-time variant of *mtree* is used.

Exercise 6.6.1 (Minimal Trees) We call a binary tree tree of depth n *minimal* if its size is minimal for all binary trees of depth n.

a) Argue that there are two different minimal binary trees of depth 2.

b) Declare a linear-time function that given n yields a minimal binary tree of depth n.

c) Declare a linear-time function that given n yields the size of minimal binary trees of depth n.

d) Give an explicit formula for the size of minimal binary trees of depth n.

Exercise 6.6.2 (Ternary Trees) Consider *ternary trees* as follows:

$$\mathit{tree} \;::=\; A \;\mid\; B(\mathit{tree}, \mathit{tree}, \mathit{tree})$$

a) Declare a function that yields the size of ternary trees.

b) Declare a function that yields the depth of ternary trees.

c) Declare a linear-time function that yields a ternary tree of depth n that has maximal size.

d) Declare a linear-time function that yields a ternary tree of depth n that has minimal size.

e) Give an explicit formula for the size of maximal ternary trees of depth n.

f) Give an explicit formula for the size of minimal ternary trees of depth n.

6.7 Big O Notation

A key idea in this chapter is taking the number of function calls needed for the execution of an expression as running time of the expression. This definition of running time works amazingly well in practice, in particular as it comes to identifying functions whose execution time may be problematic in practice. As one would expect, in a functional programming language significant running times can only be obtained with recursion.

Another basic idea is looking at the maximal running time for arguments of a given size. For lists, typically their length is taken as size. The idea of taking the maximal running time for all arguments of a given size is known as *worst-case assumption*. A good example for the worst-case assumption is insertion sort, where the best-case running time is linear and the worst-case running time is quadratic in the length of the input list.

Once a numeric argument size is fixed, one can usually define the worst-case running time function for a given function as a recursive function $\mathbb{N} \to \mathbb{N}$ following the defining equations of the given function.

Given a recursive running time function, one can usually obtain an explicit formula for the running time using so-called recurrence solving techniques from discrete mathematics. Once one has an explicit formula for the running time, one can obtain the asymptotic running time in big O notation (e.g., $O(n)$, $O(n \log n)$, $O(n^2)$). Big O notation is also known as *Bachmann–Landau notation* or *asymptotic notation*.

Big O notation is based on an *asymptotic dominance relation* $f \preceq g$ for functions $f, g : \mathbb{N} \to \mathbb{N}$ defined as follows:

$$f \preceq g \;\;:=\;\; \exists\, n_0, k, c.\; \forall\, n \geq n_0.\;\; f(n) \leq k \cdot g(n) + c$$

Speaking informally, we have $f \preceq g$ iff f is upwards bounded by a pumped version of g for all n but finitely many exceptions.

Asymptotic dominance gives us the *asymptotic equivalence relation*

$$f \approx g \;\;:=\;\; f \preceq g \;\wedge\; g \preceq f$$

If $f \approx g$, one says that f and g have the same **growth rate**.

Here are a few examples to help your intuition:

$$\begin{aligned} 2n + 13 \;&\approx\; n \\ 7n^2 + 2n + 13 \;&\approx\; n^2 \\ 2^{n+3} + 7n^2 + 2n + 13 \;&\approx\; 2^n \end{aligned}$$

We also have

$$\underbrace{n^1 \preceq n^2 \preceq n^3 \preceq \cdots}_{\text{polynomial}} \preceq \underbrace{2^n \preceq 3^n \preceq 4^n \preceq \cdots}_{\text{exponential}}$$

where the converse directions do not hold. One speaks of polynomial and exponential functions.

We now say that a function f is $O(g)$ if $f \preceq g$. Big O notation like $O(n)$ and $O(n^2)$ are in fact abbreviations for $O(\lambda n.n)$ and $O(\lambda n.n^2)$. Moreover, $O(n \log n)$ abbreviates $O(\lambda n.\, n \cdot \ulcorner \log_2(n+1) \urcorner)$.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as the order of the function.

In the literature, one uses the notation "f is $\Theta(g)$" for $f \approx g$ (tight upper bound). This means that there is an input to the program that leads to an **asymptotically equivalent** running time to g.

Exercise 6.7.1 Give the order of the following functions using big O notation.

a) $\lambda n.\, 2n + 13$

b) $\lambda n.\, 5n^2 + 2n + 13$

c) $\lambda n.\, 5n^3 + n(n+1) + 13n + 2$

d) $\lambda n.\, 2^n + n^4$

e) $\lambda n.\, 2^{n+5} + n^2$

Exercise 6.7.2 (Characterizations of asymptotic dominance) Show that the following propositions are equivalent for all functions $f, g : \mathbb{N} \to \mathbb{N}$:

1. $\exists\, n_0, k, c.\ \forall\, n \geq n_0.\ \ f(n) \leq k \cdot g(n) + c$
2. $\exists k. \forall n.\ \ f(n) \leq k \cdot (g(n) + 1)$

6.8 Call Depth

The execution of a function call usually requires the execution of further function calls. This is particularly true for calls of recursive functions. The function calls that must be executed for a given initial function call are naturally organized into a *call tree* where the calls needed for a particular call appear as descendants of the particular call. Note that the size of a call tree is the abstract running time of the initial function call appearing as the root of the tree.

When a call is executed, all calls in its call tree must be executed. When an interpreter executes a call that is a descendent of the the initial call, it must store the path from the root to the current call so that it can return from the current call to the parent call. The path from the initial call to the current call is stored in the so-called *call stack*.

The length of the path from the initial call to a lower call is known as *call depth* (often called *recursion depth*). In contrast to functional programming languages, most programming languages severely limit the available call depth, typically to a few thousand calls. At this point tail-recursion and tail-recursive calls turn out to be important since tail-recursive calls don't count for the call depth in some programming systems limiting the call depth. The reason is that a tail-recursive call directly yields the result of the parent call so that it can replace the parent call on the call stack when its execution is initiated.

In contrast to native OCaml interpreters, the browser-based TryOCaml interpreter (realized with JavaScript) limits the recursion depth to a few thousand calls. If you want to attack computationally demanding problems with TryOCaml, it is necessary that you limit the necessary recursion depth by replacing full recursion with tail recursion. A good example for this problem is merge sort, where the helper functions *split* and *merge* can be realized with tail recursion and only the main function requires full recursion. Since the main function halves the input size upon recursion, it requires call depth that is only logarithmic in the size of the initial input (for instance, call depth 20 for a list of length 10^6).

7 Inductive Correctness Proofs

In this chapter we will prove properties of functions defined with equations and terminating recursion. To handle recursive functions, terminating recursive proofs commonly known as inductive proofs will be used.

We assume familiarity with equational reasoning, a basic mathematical proof technique obtaining valid equations by rewriting with valid equations. For instance, $5 + (x - x) = 5$ follows with the equations $x - x = 0$ and $x + 0 = x$. One speaks of *replacement of equals by equals*.

7.1 Propositions and Proofs

Propositions are mathematical statements that may be true or false. Equations are a basic form of propositions. Proofs are mathematical constructions that demonstrate the truth of propositions. Propositions can be combined with three basic connectives:

conjunction	$P \wedge Q$	P and Q
disjunction	$P \vee Q$	P or Q
implication	$P \to Q$	if P then Q

There is a special proposition $\bot$ called falsity that has no proof. Negation $\neg P$ is expressed as $P \to \bot$. Variables occurring in propositions can be qualified with typed quantifications:

universal quantification	$\forall x : t.\, P$	for all x of type t, P
existential quantification	$\exists x : t.\, P$	for some x of type t, P

Propositions are used all the time in mathematical reasoning. Often propositions are formulated informally using natural language.

For every proposition but $\bot$, there is a basic form of proof:

- To prove a conjunction $P \wedge Q$, one proves both P and Q.
- To prove a disjunction $P \vee Q$, one proves either P or Q.
- To prove an implication $P \to Q$, one describes a function that given a proof of P yields a proof of Q.
- To prove a universal quantification $\forall x : t.\, P$, one describes a function that given a value x of type t yields a proof of P.
- To prove an existential quantification $\exists x : t.\, P$, one gives a value x of type t and a proof of P.

Usually propositions and proofs are written in natural language using symbolic notation where convenient. There are typed functional programming languages in which one can write propositions and proofs, and these languages are implemented with proof assistants helping with the construction of proofs.[1]

7.2 List induction

Many properties of the basic list functions defined in Chapter 2 can be expressed with equations. The functions for list concatenation and list reversal, for instance, satisfy the following equations:

$$\begin{aligned} (l_1 \mathbin{@} l_2) \mathbin{@} l_3 &= l_1 \mathbin{@} (l_2 \mathbin{@} l_3) \\ \mathit{rev}\,(l_1 \mathbin{@} l_2) &= \mathit{rev}\ l_2 \mathbin{@} \mathit{rev}\ l_1 \\ \mathit{rev}\,(\mathit{rev}\ l) &= l \end{aligned}$$

The letters l_1, l_2, l_3, and l act as variables ranging over lists over some base type α. With all typed quantifications made explicit, the second equation, for instance, becomes

$$\forall \alpha : \mathbb{T}.\ \forall l_1 : \mathcal{L}(\alpha).\ \forall l_2 : \mathcal{L}(\alpha).\ \ \mathit{rev}\,(l_1 \mathbin{@} l_2) = \mathit{rev}\ l_2 \mathbin{@} \mathit{rev}\ l_1$$

where the symbol $\mathbb{T}$ represents the type of types. We will usually not give the types of the variables, but for the correctness of the mathematical reasoning it is essential that types can be assigned consistently.

The equations can be shown by equational reasoning using the defining equations of the functions and a recursive proof technique called list induction.

Fact 7.2.1 (List induction) To show that a proposition $p(l)$ holds for all lists l, it suffices to show the following propositions:

1. *Nil case:* $p([])$
2. *Cons case:* $\forall x. \forall l.\ p(l) \rightarrow p(x :: l)$

Proof We assume proofs of the nil case and the cons case. Let l be a list. We prove $p(l)$ by case analysis and recursion on l. If $l = []$, the nil case gives us a proof. If $l = x :: l'$, recursion gives us a proof of $p(l')$. Now the function proving the cons case gives us a proof of $p(l)$. ■

The predicate p is known as **induction predicate**,[2] and the proposition $p(l)$ in the cons case is known as **inductive hypothesis**.

[1] A popular proof assistant we use in Saarbrücken is the Coq proof assistant.

[2] A predicate may be thought of as a function that yields a proposition.

We will demonstrate equational reasoning and the use of list induction by proving some properties of list concatenation. We will make extensive use of the defining equations for @ :

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

We say that an equation follows by **simplification** if its two sides can be made equal by applying defining equations (from left to right). For instance,

$$[1,2] \mathbin{@} ([3,5] \mathbin{@} [5,6]) = [1,2,3,5] \mathbin{@} [5,6]$$

follows by simplification (both sides simplify to $[1,2,3,5,5,6]$).

Once variables are involved, simplification is often not enough. For instance, $[\,] \mathbin{@} l = l$ follows by simplification, but $l \mathbin{@} [\,] = l$ does not. To prove $l \mathbin{@} [\,] = l$, we need list induction in addition to simplification.

Fact $l \mathbin{@} [\,] = l$.

Proof By list induction on l.

- Nil case: $[\,] \mathbin{@} [\,] = [\,]$ holds by simplification.
- Cons case: We assume a proof of $l \mathbin{@} [\,] = l$ and show $(x :: l) \mathbin{@} [\,] = x :: l$. By simplification it suffices to prove $x :: (l \mathbin{@} [\,]) = x :: l$. Holds by the inductive hypothesis, which gives us a proof of $l \mathbin{@} [\,] = l$. ■

The above proof is quite wordy for a routine argument. We will adopt a more compact format.

Fact 7.2.2 $l \mathbin{@} [\,] = l$.

Proof Induction on l. The nil case holds by simplification. Cons case:

$$\begin{aligned} (x :: l) \mathbin{@} [\,] &= x :: l && \text{simplification} \\ x :: (l \mathbin{@} [\,]) & && \text{induction} \\ x :: l & && \end{aligned}$$

■

The compact format arranges things such that the inductive hypothesis is identical with the claim that is proven. This way there is no need to write down the inductive hypothesis.

Our second example for an inductive proof concerns the *associativity* of list concatenation.

Fact 7.2.3 $(l_1 @ l_2) @ l_3 = l_1 @ (l_2 @ l_3)$.

Proof Induction on l_1. The nil case holds by simplification. Cons case:

$$\begin{array}{rcll} ((x :: l_1) @ l_2) @ l_3 & = & (x :: l_1) @ (l_2 @ l_3) & \text{simplification} \\ x :: ((l_1 @ l_2) @ l_3) & = & x :: (l_1 @ (l_2 @ l_3)) & \text{induction} \\ x :: (l_1 @ (l_2 @ l_3)) & & & \end{array}$$ ■

Exercise 7.2.4 Prove $\mathit{map}\, f\, (l_1 @ l_2) = \mathit{map}\, f\, l_1 @ \mathit{map}\, f\, l_2$.

7.3 Properties of List Reversal

We now use equational reasoning and list induction to prove equational properties of list reversal. For some cases a quantified inductive hypothesis is needed, an important feature we have not seen before. We start with the definition of naive list reversal:

$$\begin{array}{rcl} \multicolumn{3}{c}{\mathit{rev} : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)} \\ \mathit{rev}\ [] & := & [] \\ \mathit{rev}\ (x :: l) & := & \mathit{rev}\,(l) @ [x] \end{array}$$

We prove a distribution property for list reversal and list concatenation. The proof requires lemmas for list concatenation we have shown before.

Fact 7.3.1 $\mathit{rev}\,(l_1 @ l_2) = \mathit{rev}\ l_2 @ \mathit{rev}\ l_1$.

Proof Induction on l_1. Nil case:

$$\begin{array}{rcll} \mathit{rev}\,([] @ l_2) & = & \mathit{rev}\ l_2 @ \mathit{rev}\ [] & \text{simplification} \\ \mathit{rev}\ l_2 & = & \mathit{rev}\ l_2 @ [] & \text{fact 7.2.2} \end{array}$$

Cons case:

$$\begin{array}{rcll} \mathit{rev}\,((x :: l_1) @ l_2) & = & \mathit{rev}\ l_2 @ \mathit{rev}\,(x :: l_1) & \text{simplification} \\ \mathit{rev}\,(l_1 @ l_2) @ [x] & = & \mathit{rev}\ l_2 @ (\mathit{rev}\ l_1 @ [x]) & \text{induction} \\ (\mathit{rev}\ l_2 @ \mathit{rev}\ l_1) @ [x] & & & \text{fact 7.2.3} \end{array}$$ ■

Next we show that naive lists reversal agrees with tail-recursive list reversal. Recall the definition of reversing concatenation (*rev_append*):

$$\begin{array}{rcl} \multicolumn{3}{c}{@_{\mathrm{r}} \ : \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathcal{L}(\alpha) \to \mathcal{L}(\alpha)} \\ [] @_{\mathrm{r}}\ l_2 & := & l_2 \\ (x :: l_1) @_{\mathrm{r}}\ l_2 & := & l_1 @_{\mathrm{r}} (x :: l_2) \end{array}$$

Fact 7.3.2 $l_1 \mathbin{@_\mathrm{r}} l_2 = \mathit{rev}\ l_1 \mathbin{@} l_2$.

Proof We prove $\forall l_2.\ l_1 \mathbin{@_\mathrm{r}} l_2 = \mathit{rev}\ l_1 \mathbin{@} l_2$ by induction on l_1. The quantification of l_2 is needed so we get a strong enough inductive hypothesis.

The nil case holds by simplification. Cons case:

$$\begin{array}{rcll}
(x :: l_1) \mathbin{@_\mathrm{r}} l_2 & = & \mathit{rev}\,(x :: l_1) \mathbin{@} l_2 & \text{simplification} \\
l_1 \mathbin{@_\mathrm{r}} (x :: l_2) & = & (\mathit{rev}\ l_1 \mathbin{@} [x]) \mathbin{@} l_2 & \text{induction} \\
\mathit{rev}\ l_1 \mathbin{@} (x :: l_2) & = & (\mathit{rev}\ l_1 \mathbin{@} [x]) \mathbin{@} l_2 & \text{fact 7.2.3} \\
\mathit{rev}\ l_1 \mathbin{@} (x :: l_2) & = & \mathit{rev}\ l_1 \mathbin{@} ([x] \mathbin{@} l_2) & \text{simplification}
\end{array}$$

Note that the inductive hypothesis is used with the instance
$l_2 := x :: l_2$. ■

The above proof uses a **quantified inductive hypothesis**, a feature we have not seen before. The induction predicate for the proof is $\lambda l_1.\forall l_2.\ l_1 \mathbin{@_\mathrm{r}} l_2 = \mathit{rev}\ l_1 \mathbin{@} l_2$. The quantification of l_2 is essential since the function $l_1 \mathbin{@_\mathrm{r}} l_2$ changes both arguments upon recursion. While induction on l_1 takes care of the change of l_1, quantification takes care of the change of l_2.

Fact 7.3.3 $\mathit{rev}\ l = l \mathbin{@_\mathrm{r}} []$.

Proof Follows with fact 7.3.2 and fact 7.2.2. ■

Fact 7.3.4 (Self inversion) $\mathit{rev}\,(\mathit{rev}\,l) = l$.

Proof By induction on l using fact 7.3.1. ■

Tail-recursive linear-time list concatenation

With reversing concatenation $l_1 \mathbin{@_\mathrm{r}} l_2$ we can define a function concatenating two lists using only tail-recursion and taking running time linear in the length of the first list:

$$\mathit{con}\ l_1\ l_2 \ := \ (l_1 \mathbin{@_\mathrm{r}} []) \mathbin{@_\mathrm{r}} l_2$$

Fact 7.3.5 $l_1 \mathbin{@} l_2 = (l_1 \mathbin{@_\mathrm{r}} []) \mathbin{@_\mathrm{r}} l_2$.

Proof Follows with facts 7.3.2 to 7.3.4. ■

Exercise 7.3.6 Give the induction predicate for the proof of fact 7.3.2.

Exercise 7.3.7 Prove $\mathit{rev}\,(\mathit{rev}\,l) = l$.

Exercise 7.3.8 Prove $map\, f\, (rev\ l) = rev\, (map\, f\, l)$.

Exercise 7.3.9 In this exercise we will write $|l|$ for $length\, l$ for better mathematical readability. Prove the following properties of the length of lists:

a) $|l_1 @ l_2| = |l_1| + |l_2|$.

b) $|rev\ l| = |l|$.

c) $|map\, f\, l| = |l|$.

Exercise 7.3.10 (Tail-recursive list concatenation)
Define a function *con* concatenating two lists using only tail-recursion whose running time is linear in the length of the first list.

Exercise 7.3.11 Give the running time function for $con\ l_1\ l_2$ and the length of l_1 not using recursion.

Exercise 7.3.12 (Power lists)
Consider a power list function defined as follows:

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

a) Prove $|pow\ l| = 2^{|l|}$ where $|l|$ is notation for $length\, l$. The equation says that a list of length n has 2^n sublists.

b) Give the running time function for function *pow*.

c) Give the running time function for a variant of *pow* which avoids the binary recursion with a let expression.

Exercise 7.3.13 (Tail-recursive length of lists) Consider the tail-recursive function

$$\begin{aligned} len\ []\ a\ &=\ a \\ len\ (x :: l)\ a\ &=\ len\ l\ (a+1) \end{aligned}$$

Prove $len\ l\ a = a + length\ l$ by induction on l. Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

7.4 Natural Number Induction

Inductive proofs are not limited to lists. The most basic induction principle is induction on natural numbers, commonly known as *mathematical induction*.

Fact 7.4.1 (Mathematical induction) To show that a proposition $p(n)$ holds for all natural numbers n, it suffices to show the following propositions:

- *Zero case:* $p(0)$
- *Successor case:* $\forall n.\ p(n) \to p(n+1)$

Proof A proof of the successor case provides us with a function that for every number n and for every proof of $p(n)$ yields a proof of $p(n+1)$. Given proofs of the zero case and the successor case, we can construct for all numbers n a proof of $p(n)$ by case analysis and recursion on n:

- If $n = 0$, the zero case provides a proof of $p(n)$.
- If $n = n' + 1$, we obtain a proof of $p(n')$ by recursion. The function provided by the successor case then gives us a proof of $p(n' + 1)$, which is a proof of $p(n)$. ■

If you are familiar with mathematical induction, you may be wondering why we are using the terms "zero case" and "successor case" in place of the standard terms "base case" and "induction step". The reason is that we want to emphasize the similarity between list induction and mathematical induction. In fact, we can see natural numbers as values that are obtained with the constructors zero and successor, similar to how lists are obtained with the constructors nil and cons. Written with constructors, the number 3 takes the form $succ\,(succ\,(succ\ 0))$, where the successor constructor satisfies $succ\ n = n + 1$.

We demonstrate number induction with the function

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

Fact 7.4.2 $\quad 2 \cdot sum\,n \ = \ n \cdot (n+1)$.

Proof By induction on n. The zero case holds by simplification. We argue the successor case:

$$\begin{array}{rcll} 2 \cdot sum\,(n+1) & = & (n+1)\cdot((n+1)+1) & \text{simplification} \\ 2 \cdot sum\,n + 2\cdot n + 2 & = & (n+1)\cdot(n+2) & \text{induction} \\ n\cdot(n+1) + 2\cdot n + 2 & & & \text{simplification} \end{array}$$ ■

Exercise 7.4.3 (Little Gauss formula) Prove the following equations by induction on n.

a) $0 + 1 + \cdots + n \;=\; n \cdot (n + 1)/2$

b) $0 + 1 + \cdots + n \;=\; \mathit{sum}\ n$

Equations (a) is known as *little Gauss formula* and gives us an efficient method to compute the sum of all numbers up to n by hand. For instance, the sum of all numbers up to 100 is $100 \cdot 101/2 = 5050$. The second equation tells us that the mathematical notation $0 + 1 + \cdots + n$ may be defined with the recursive function *sum*.

Exercise 7.4.4 (Running time functions)

In Chapter 6, we derived the following running time functions:

$$\begin{aligned} r_1(0) &:= 1 & r_2(0) &:= 1 \\ r_1(n+1) &:= 1 + r_1(n) & r_2(n+1) &:= 1 + r_2(n) + (n+1) \end{aligned}$$

$$\begin{aligned} r_3(0) &:= 1 \\ r_3(n+1) &:= 1 + r_3(n) + r_3(n) \end{aligned}$$

Using a recurrence solver, we obtained explicit characterizations of the functions:

$$r_1(n) = n + 1 \qquad r_2(n) = (n+1)(n+2)/2$$
$$r_3(n) = 2^{n+1} - 1$$

Use mathematical induction to verify the correctness of the explicit characterizations. Note that the proof obligations imposed by mathematical induction boil down to checking that the explicit characterizations satisfy the defining equations of the running time functions.

Exercise 7.4.5 (Sum of odd numbers)

Define a function $f : \mathbb{N}^+ \to \mathbb{N}^+$ computing the sum $1 + 3 + \cdots + (2n - 1)$ of the odd numbers from 1 to $2n - 1$. Prove $f(n) = n^2$.

Exercise 7.4.6 (Sum of square numbers)

Define a function $f : \mathbb{N}^+ \to \mathbb{N}^+$ computing the sum $0^2 + 1^2 + \cdots + n^2$ of the square numbers. Prove $f(n) = n \cdot (2n^2 + 3n + 1)/6$.

7.5 Correctness of Tail-Recursive Formulations

Given a recursive function that is not tail-recursive, one can often come up with a tail-recursive version collecting intermediate results in additional accumulator arguments. One then would like to prove that the tail-recursive version agrees with the original version. We already saw such a proof in Section 7.3 for the tail-recursive version of list-reversal. For list reversal we have to prove

$$rev\ l \ = \ l \mathbin{@_{\mathrm{r}}} []$$

using list recursion. The induction will only go through if we generalize the correctness statement to

$$l_1 \mathbin{@_{\mathrm{r}}} l_2 \ = \ rev\ l_1 \mathbin{@} l_2$$

If we think about it, the generalized correctness statement amounts to an equational specification of $l_1 \mathbin{@_{\mathrm{r}}} l_2$. Since the tail-recursive function $l_1 \mathbin{@_{\mathrm{r}}} l_2$ recurses on l_1 and modifies l_2, a proof by list induction on l_1 quantifying the accumulator arguments l_2 works.

Recall the Fibonacci function from Section 1.16:

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

One speaks of a binary recursion since there are two parallel recursive applications in the 3rd defining equation. It is known that the running time of *fib* is exponential in the sense that it is not bounded by any polynomial.

To come up with a tail-recursive version of *fib*, we observe that a Fibonacci number is obtained as the sum of the preceding two Fibonacci numbers starting from the initial Fibonacci numbers 0 and 1. In Section 1.16 this fact is used to obtain the Fibonacci numbers with iteration. We now realize the iteration using two accumulator arguments for the preceding Fibonacci numbers:

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

We would like to show

$$\mathit{fib}\ n \ = \ \mathit{fibo}\ n\ 0\ 1$$

A proof by induction on n will not go through since *fibo* modifies its two accumulator arguments. What we need is a more general equation specifying *fibo* more accurately:

$$\mathit{fibo}\ n\ (\mathit{fib}\ m)\ (\mathit{fib}\,(m+1)) \ = \ fib\,(n+m)$$

We prove the equation by induction on n with m quantified. The zero case holds by simplification, and the successor case goes as follows:

$$\begin{array}{rcll}
\mathit{fibo}\ (n+1)\ (\mathit{fib}\ m)\ (\mathit{fib}\,(m+1)) & = & \mathit{fib}\,(n+1+m) & \text{simplification} \\
\mathit{fibo}\ n\ (\mathit{fib}\,(m+1))\ (\mathit{fib}\ m + \mathit{fib}\,(m+1)) & & & \text{def. eq. } \mathit{fib} \leftarrow \\
\mathit{fibo}\ n\ (\mathit{fib}\,(m+1))\ (\mathit{fib}\,(m+2)) & & & \text{induction} \\
\mathit{fib}\,(n+(m+1)) & & & \text{simplification}
\end{array}$$

Note the right-to-left application of the 3rd defining equation of *fib*.

Exercise 7.5.1 (Tail-recursive factorial function)

Consider the tail-recursive function

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

Prove $\mathit{fac}\ n\ a = a \cdot n!$ by induction on n assuming the equations $0! = 1$ and $(n+1)! = (n+1)\cdot n!$. Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

Exercise 7.5.2 (Tail-recursive power function)

Consider the tail-recursive function

$$\begin{aligned}
&\mathit{pow} : \ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\qquad \mathit{pow}\ x\ 0\ a \ := \ a \\
&\mathit{pow}\ x\ (n+1)\ a \ := \ \mathit{pow}\ x\ n\ (x \cdot a)
\end{aligned}$$

Prove $\mathit{pow}\ x\ n\ a = x^n \cdot a$ by induction on n. Note that a needs to be quantified in the inductive hypothesis. Give the induction predicate.

Exercise 7.5.3 (Tail-recursive sum) Define a tail-recursive variant sum' of *sum* and prove $\mathit{sum}'\ n\ 0 = \mathit{sum}\ n$.

Exercise 7.5.4 There is an alternative proof of the correctness of *fibo* using a lemma for *fibo*.

a) Prove $\mathit{fibo}\ (n+2)\ a\ b \ = \ \mathit{fibo}\ n\ a\ b + \mathit{fibo}\ (n+1)\ a\ b$.

b) Prove $\mathit{fib}\ n \ = \ \mathit{fibo}\ n\ 0\ 1$ by induction on n using (a).

Hint: (a) follows by induction on n with a and b quantified.

7.6 Properties of Iteration

Using number induction, we will prove some properties of iteration Section 1.15.

We start with the iteration function

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

Note that *it* is defined differently from the tail-recursive iteration function *iter* in Section 1.15. We will eventually show that both iteration functions agree.

We first show that factorials

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

can be computed with iteration.

Fact 7.6.1 $(n, n!) \ = \ it\ (\lambda\,(k,a).\,(k+1,\ (k+1)\cdot a))\ n\ (0,1)$.

Proof We define $f(k,a) := (k+1,\ (k+1)\cdot a)$ and prove

$$(n, n!) \ = \ it\ f\ n\ (0,1)$$

by induction on n. The zero case holds by simplification. We argue the successor case:

$$\begin{aligned} (n+1,\ (n+1)!) \ &=\ it\ f\ (n+1)\ (0,1) && \text{simplification} \\ f\,(n, n!) \ &=\ f\,(it\ f\ n\ (0,1)) && \text{induction} \end{aligned}$$ ■

Next we prove an important property of *it* that we need for showing the agreement with *iter*.

Fact 7.6.2 (Shift Law) $f\,(it\ f\ n\ x) \ = \ it\ f\ n\ (f\,x)$.

Proof Induction on n. The zero case holds by simplification.
Successor case:

$$\begin{aligned} f\,(it\ f\ (n+1)\ x) \ &=\ it\ f\ (n+1)\ (f\,x) && \text{simplification} \\ f\,(f\,(it\ f\ n\ x)) \ &=\ f\,(it\ f\ n\ (f\,x)) && \text{induction} \end{aligned}$$ ■

Recall the tail-recursive iteration function *iter* from Section 1.15 :

$$\begin{aligned} \mathit{iter}\ f\ 0\ x &:= x \\ \mathit{iter}\ f\ (n+1)\ x &:= \mathit{iter}\, f\, n\, (f\, x) \end{aligned}$$

Fact 7.6.3 (Agreement) $\mathit{it}\, f\, n\, x = \mathit{iter}\, f\, n\, x$.

Proof We prove $\forall x.\ \mathit{it}\, f\, n\, x = \mathit{iter}\, f\, n\, x$ by induction on n. The quantification of x is needed since *iter* changes this argument upon recursion. The zero case holds by simplification. Successor case:

$$\begin{aligned} \mathit{it}\, f\, (n+1)\, x &= \mathit{iter}\, f\, (n+1)\, x && \text{simplification} \\ f(\mathit{it}\, f\, n\, x) &= \mathit{iter}\, f\, n\, (fx) && \text{shift law} \\ \mathit{it}\, f\, n\, (fx) & && \text{induction} \end{aligned}$$ ■

Exercise 7.6.4 Prove $x^n = \mathit{it}\,(\lambda a.\ a \cdot x)\, n\, 1$.

Exercise 7.6.5 Recall the Fibonacci function *fib* from Section 7.5. Prove $(\mathit{fib}\ n, \mathit{fib}\,(n+1)) = \mathit{it}\, f\, n\ (0,1)$ for $f\ (a,b) := (b, a+b)$.

Exercise 7.6.6 Prove the shift law for *iter* in two ways: (1) by a direct inductive proof, and (2) using the agreement with *it* and the shift law for *it*.

7.7 Induction for Terminating Functions

Given a terminating recursive function, and a property concerning the results of the function, we can prove the property following the defining equations of the function and assuming the property for the recursive calls of the function. Such a proof is an inductive proof following the recursion scheme underlying the defining equations. We speak of **function induction**.

Our first example for function induction concerns the remainder operation:

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

Note that the type of % ensures that $y > 0$ in $x \,\%\, y$, a property that in turn ensures termination of the function. From the definition it seems that we have $x \,\%\, y < y$. A proof of this property needs induction to account for the recursion of the second defining equation.

Fact $x \,\%\, y < y$.

Proof By induction on $x \,\%\, y$. If $x \,\%\, y$ is obtained with the first defining equation, we have $x < y$ and $x \,\%\, y = x$, which yields the claim. If $x \,\%\, y$ is obtained with the second defining equation, the claim follows by induction. More precisely, we have $x \,\%\, y = (x-y) \,\%\, y$ and $(x-y) \,\%\, y < y$ by induction for the recursive call. ■

The induction used in the proof is function induction. A proof by function induction follows the case analysis established by the defining equations and assumes that the claim holds for the recursive applications. The use of function induction is admissible for functions with terminating defining equations. What happens is that function induction constructs a recursive proof following the terminating recursion of the defining equations.

We impose the restriction that the proposition to be shown by function induction contains exactly one application of the function governing the induction. A proof by function induction now case analyses the function application following the defining equations of the function. By simplifying with the defining equations recursive calls of the function are introduced, and for these calls we do have the inductive hypothesis.

Using the above example, we introduce a more formalized way to write down proofs by function induction.

Fact 7.7.1 $x \,\%\, y < y$.

Proof By induction on $x \,\%\, y$.

1. $x \,\%\, y = x$ and $x < y$. The claim follows.
2. $x \,\%\, y = (x - y) \,\%\, y$ and $x \geq y$. By induction we have $(x - y) \,\%\, y < y$. The claim follows. ■

Exercise 7.7.2 Let $y > 0$. Prove $x = (x/y) \cdot y + x \,\%\, y$ by function induction on x/y. First write down the type and the defining equations for x/y and $x \,\%\, y$.

7.8 Euclid's Algorithm

There is an elegant algorithm known as *Euclid's algorithm* computing the greatest common divisor of two integers $x, y \geq 0$. The algorithm is based on two straightforward rules:

1. If one of the numbers is 0, the other number is the greatest common divisor of the two numbers.

2. If $0 < y \leq x$, replace x with $x - y$.

Note that either the first or the second rule is applicable, and that application of the second rule terminates since the second rule decreases the sum of the two numbers.

We now recall that $x \,\%\, y$ is obtained by subtracting y from x as long as $y \leq x$. This gives us the following function for computing greatest common divisors:

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

The function terminates since recursion decreases the second argument (ensured by fact 7.7.1). Here is a trace:

$$gcd\ 91\ 35 \ = \ gcd\ 35\ 21 \ = \ gcd\ 21\ 14 \ = \ gcd\ 14\ 7 \ = \ gcd\ 7\ 0 \ = \ 7$$

Our goal here is to carefully explain how the algorithm can be derived from basic principles so that the correctness of the algorithm is obvious.

In this section, all numbers will be nonnegative integers. We first define the **divides relation** for numbers:

$$k \mid x \ := \ \exists n.\ x = n \cdot k \qquad\qquad k \textbf{ divides } x$$

By our convention, k, x, and n are all nonnegative integers. When we have $k \mid x$, we say that k is a **divisor** or a **factor** of x. Moreover, when we say that k is a **common divisor** of x and y we mean that k is a divisor of both x and y.

We observe that every number x has 1 and x as divisors. Moreover, a positive number x has only finitely many divisors, which are all between 1 and x. In contrast, 0 has infinitely many divisors since it is divided by every number. It follows that two numbers have a greatest common divisor if and only if not both numbers are zero.

We call a number z the **GCD** of two numbers x and y if the divisors of z are exactly the common divisors of x and y. Here are examples:

- The GCD of 15 and 63 is 3.
- The GCD of 0 and 0 is 0.

More generally, we observe the following facts.

Fact 7.8.1 (GCD)

1. The GCD of two numbers uniquely exists.

2. The GCD of x and x is x.
3. *Zero rule:* The GCD of x and 0 is x.
4. *Symmetry:* The GCD of x and y is the GCD of y and x.
5. The GCD of two numbers that are not both 0 is the greatest common divisor of the numbers.

Our goal is to find an algorithm that computes the GCD of two numbers. If one of the two numbers is 0, we are done since the other number is the GCD of the two numbers. It remains to find a rule that given two positive numbers x and y yields two numbers x' and y' that have the same GCD and whose sum is smaller (i.e., $x+y > x'+y'$). We then can use recursion to reduce the initial numbers to the trivial case where one of the numbers is 0.

It turns out that the GCD of two number $x \geq y$ stays unchanged if x is replaced with the difference $x - y$.

Fact 7.8.2 (Subtraction Rule) If $x \geq y$, then the common divisors of x and y are exactly the common divisors of $x - y$ and y.

Proof We need to show two directions.

Suppose $x = n \cdot a$ and $y = n \cdot b$. We have $x - y = n \cdot a - n \cdot b = n \cdot (a - b)$.

Suppose $x - y = n \cdot a$ and $y = n \cdot b$. We have $x = x - y + y = n \cdot a + n \cdot b = n \cdot (a + b)$. ■

Fact 7.8.3 (Modulo Rule) If $y > 0$, then the common divisors of x and y are exactly the common divisors of $x \,\%\, y$ and y.

Proof Follows with the subtraction rule (fact 7.8.2) since $x \,\%\, y$ is obtained from x by repeated subtraction of y. More formally, we argue the claim by function induction on $x \,\%\, y$.

1. $x \,\%\, y = x$ and $x < y$. The claim follows.
2. $x \,\%\, y = (x - y) \,\%\, y$ and $x \geq y$. By induction we have that the common divisors of $x - y$ and y are the common divisors of $(x - y) \,\%\, y$ and y. The claim follows with the subtraction rule (fact 7.8.2). ■

Euclid's algorithm now simply applies the modulo rule until the zero rule applies, using the symmetry rule when needed. Recall our formulation of Euclid's algorithm with the function

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

Recall that *gcd* terminates since recursion decreases the second argument (ensured by fact 7.7.1).

Fact 7.8.4 (Correctness) $gcd\,x\,y$ is the GCD of x and y.

Proof By function induction on $gcd\ x\ y$.

1. $gcd\ x\ 0 = x$. Claim follows by zero rule.
2. $gcd\ x\ y = gcd\ y\ (x \,\%\, y)$ and $y > 0$. By induction we know that $gcd\ y\ (x \,\%\, y)$ is the GCD of y and $x \,\%\, y$. Hence $gcd\ x\ y$ is the GCD of y and $x \,\%\, y$. It follows with the modulo rule and the symmetry rule that $gcd\ x\ y$ is the GCD of x and y. ■

Running Time

The running time of $gcd\,x\,y$ is at most y since each recursion step decreases y. In fact, one can show that the asymptotic running time of $gcd\,x\,y$ is logarithmic in $\max\{x, y\}$ assuming that $x \,\%\, y$ requires constant time, i. e. its running time is independent of x and y. This assumption is reasonable because modern computers can compute the remainder of a division of two integers of a given size (typically 64 bits) within a few clock cycles, mostly independently of the values of the numbers.

To see that the running time is logarithmic, let us consider two recursive calls of $gcd\ x\ y$ and assume that $0 < y < x$. This assumption is reasonable because for $y = 0$, *gcd* is not defined and for $y \geq x$ we have that $x \,\%\, y = x$, so in this case $gcd\ x\ y = gcd\ y\ x$. Hence, if $y \geq x$, the first recursive call just puts the smaller argument in the second position.

The two recursive calls then look like this:

$$gcd\ x\ y = gcd\ y\ (x \,\%\, y) = gcd\ (x \,\%\, y)\ \ldots$$

We claim that for $0 < y < x$, $x \,\%\, y \leq x/2$ which means that after two recursive calls *gcd* at least halves its first argument. To see this, let us consider the following two cases:

1. $y \leq x/2$. With $x \,\%\, y < y$ (fact 7.7.1) we have $x \,\%\, y < y \leq x/2$.
2. $y > x/2$. Because $y > x/2$, we have $x/y = 1$: y does not fit into x more than once. Hence, $x \,\%\, y = x - y$. Because $x/2 < y < x$ by assumption, $x - y < x/2$ and therefore $x \,\%\, y < x/2$.

Hence, *gcd* at least halves its first argument every two recursive calls and the worst-case running time is $O(\log_2(\max\{x, y\}))$.

Exercise 7.8.5 Define a function $g : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ computing the GCD of two numbers whose recursion decreases the first argument.

Exercise 7.8.6 Define a function $g : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ computing the GCD of two numbers using subtraction and not using the remainder operation. Argue the correctness of your function. Hint: Define the function such that recursion decreases the sum of the two arguments.

Exercise 7.8.7 Define a function $g : \mathbb{N}^+ \to \mathbb{N}^+ \to \mathbb{N}^+$ computing the GCD of two positive numbers using the remainder function. Argue the correctness of your function. Use only one conditional.

Exercise 7.8.8 Define a function $g : \mathcal{L}(\mathbb{N}) \to \mathbb{N}$ such that the common divisors of the elements of a list l are exactly the divisors of $g(l)$.

Exercise 7.8.9 Argue the correctness of the claims of fact 7.8.1.

7.9 Complete Induction

Complete induction is a generalization of mathematical induction making it possible to assume in a proof of $p(n)$ a proof of $p(k)$ for all $k < n$. Complete induction may be seen as a natural scheme for recursive proofs where the recursion is on natural numbers. Using complete induction, we can justify the function inductions we have seen so far. While complete induction appears to be more powerful than mathematical induction, it can be obtained from mathematical induction with an elegant proof.

Fact 7.9.1 (Complete induction) To show that a proposition $p(n)$ holds for all natural numbers n, it suffices to show the following proposition:

$$\forall n.\ (\forall k < n.\ p(k)) \ \to\ p(n)$$

Proof We assume $H_1 : \forall n.\ (\forall k < n.\ p(k)) \ \to\ p(n)$. To show $\forall n. p(n)$, we show more generally

$$\forall n.\ \forall k < n.\ p(k)$$

by induction on n. So the trick is to do mathematical induction on the upper bound introduced by the generalization.

The zero case $\forall k < 0.\ p(k)$ is obvious since there is no natural number $k < 0$.

In the successor case, we have the inductive hypothesis

$$H_2 : \forall k < n.\ p(k)$$

and show $\forall k < n+1.\ p(k)$. We assume $H_3 : k < n+1$ and prove $p(k)$. By H_1 it suffices to prove

$$\forall k' < k.\ p(k')$$

We assume $k' < k$ and prove $p(k')$. By H_3 we have $k' < n$. Now the claim follows by H_2. ■

Informally, we may say that complete induction allows us to assume proofs of all propositions $p(k)$ with $k < n$ when we prove $p(n)$. In a certain sense, we get this inductive hypothesis for free, in the same way we get recursion for free when we define a function. In contrast to mathematical induction, complete induction does not impose a case analysis.

We can now explain the function induction on $x \,\%\, y$ used for fact 7.7.1, exercise 7.7.2, and fact 7.8.3 with complete induction on x. Similarly, the function induction on $gcd\, x\, y$ used for fact 7.8.4 can be explained with complete induction on y. In each case, the complete induction agrees with the termination argument for the underlying function ($x \,\%\, y$ decreases x, $gcd\, x\, y$ decreases y). The induction predicates for the complete inductions are as follows:

- fact 7.7.1: $p(x) := \ x \,\%\, y < y$.
- exercise 7.7.2: $p(x) := \ x = (x/y) \cdot y + x \,\%\, y$.
- fact 7.8.3: $p(x) :=$ the cds[3] of x, y are the cds of y, $x \,\%\, y$.
- fact 7.8.4: $p(y) := \ \forall x.\ gcd\ x\ y$ is the GCD of x, y.

That x is quantified in the induction predicate for fact 7.8.4 is necessary since gcd changes both arguments. We will not go further into logical details here but instead rely on our mathematical intuition.

We give three examples for the use of complete induction.

Fact 7.9.2 A list of length n has 2^n sublists.

Proof For $n = 0$, the claim is obvious. For $n > 0$, the list has the form $x :: l$. Every sublist of $x :: l$ now either keeps x or deletes x. A sublist of $x :: l$ that deletes x is a sublist of l. By complete induction we know that l has 2^{n-1} sublists. We now observe that the sublists of $x :: l$ that keep x are exactly the sublists of l with x added in front. This gives us that l has $2^{n-1} + 2^{n-1} = 2^n$ sublists. ■

Note that the proof suggests a function that given a list obtains a list of all sublists.

[3] common divisors

Recall that a prime number is an integer greater 1 that cannot be obtained as the product of two integers greater 1 (Section 1.17).

Fact 7.9.3 Every integer greater than 1 has a prime factorization.

Proof Let $x > 1$. If x is prime, then $[x]$ is a prime factorization of x. Otherwise, $x = x_1 \cdot x_2$ with $1 < x_1, x_2 < x$. By complete induction we have prime factorizations l_1 and l_2 for x_1 and x_2. We observe that $l_1 \mathbin{@} l_2$ is a prime factorization of x. ■

Note that the proof suggest a function that given a number greater 1 obtains a prime factorization of the number. The function relies on a helper function that given $x > 1$ decides whether there are $x_1, x_2 > 1$ such that $x = x_1 \cdot x_2$.

Fact 7.9.4 (Euclidean division) Let $x \geq 0$ and $y > 0$. Then there exist $a \geq 0$ and $0 \leq b < y$ such that $x = a \cdot y + b$.

Proof By complete induction on x. If $x < y$, we satisfy the claim with $a = 0$ and $b = x$. Otherwise $x \geq y$. By the inductive hypothesis we have $x - y = a \cdot y + b$ with $a \geq 0$ and $0 \leq b < y$. We have $x = (a+1) \cdot y + b$, which yields the claim. ■

Exercise 7.9.5 Declare a prime factorization function following the proof of fact 7.9.3.

Exercise 7.9.6 Declare a division function following the proof of fact 7.9.4. Given x and y as required, the function should return a quotient-remainder pair (a, b) for x and y.

Exercise 7.9.7 Prove $\mathit{fib}(n) < 2^n$ by complete induction.

Exercise 7.9.8 Derive mathematical induction from complete induction.

7.10 Prime Factorization Revisited

We have discussed prime factorization algorithms in Section 2.15. We now review the correctness arguments in the light of the formal foundations we have laid out in this chapter. The verification of the optimized prime factorization algorithm will require invariants, an important feature we have not yet seen in this chapter.

We recall two important definitions:

$$\begin{aligned} k \mid x &:= \exists n.\ x = n \cdot k && k \text{ divides } x \\ \mathit{prime}\ x &:= x \geq 2 \land \neg\exists\, k, n \geq 2.\ x = k \cdot n \end{aligned}$$

The definitions assume that k and x are natural numbers.

We start with a straightforward prime factorization algorithm:

$$\begin{aligned} &\mathit{pfac} :\ \mathbb{N} \to \mathcal{L}(\mathbb{N}) \\ &\mathit{pfac}\ x\ := \begin{cases} [] & \text{if } x < 2 \\ \text{LET } k = \mathit{first}\,(\lambda k.\ k \,|\, x)\, 2 \text{ IN } k :: \mathit{pfac}\,(x/k) & \text{if } x \geq 2 \end{cases} \end{aligned}$$

We first argue the termination of *pfac*. The linear search realized with *first* terminates since $x \geq 2$, the search starts with 2, and x divides x. The linear search yields a k such that $2 \leq k \leq x$ and thus $x/k < x$. Thus *pfac* terminates since it decreases its argument.

We now show

$$(x < 2 \ \land\ \mathit{pfac}\, x = []) \ \lor\ (x \geq 2 \ \land\ \mathit{pfac}\, x \text{ is prime fact. of } x)$$

by functional induction on *pfac* x. The interesting case is $x \geq 2$. The linear search gives us the least prime factor k of x satisfying $2 \leq k \leq x$. We have either $k < x$ or $k = x$. Thus either $x/k \geq 2$ or $x/k = 1$. With the inductive hypothesis it now follows that $k :: \mathit{pfac}\,(x/k)$ is the prime factorization of x.

Next we consider an optimized prime factorization algorithm:

$$\begin{aligned} &\mathit{pfac}' :\ \mathbb{N} \to \mathbb{N} \to \mathcal{L}(\mathbb{N}) \\ &\mathit{pfac}'\ k\ x\ := \begin{cases} [x] & \text{if } k^2 > x \\ k :: \mathit{pfac}'\ k\ (x/k) & \text{if } k^2 \leq x \text{ and } k \mid x \\ \mathit{pfac}'\,(k+1)\ x & \text{if } k^2 \leq x \text{ and } \neg(k \mid x) \end{cases} \end{aligned}$$

We would like to show that $\mathit{pfac}'\, 2\ x$ yields a prime factorization of x if $x \geq 2$. Since k (the counter) is modified upon recursion, it is clear that $\mathit{pfac}'\, 2\ x$ cannot be verified by itself and we have to look more generally at $\mathit{pfac}'\, k\ x$ where the arguments satisfy some safety condition. The safety condition must be satisfied by the initial arguments and must be preserved for the recursive calls identified by the defining equations of *pfac′*. Safety conditions that are preserved for the recursive calls of a function are called **invariants**. Safety conditions act as preconditions for a function specifying the admissible arguments of the function.

So far, all functions we did correctness proofs for did terminate for all arguments admitted by the types of the functions. This is not the case for $pfac'$, which diverges for $x = 1$ and $k = 1$. Thus we need an invariant for $pfac'$ ensuring termination. We choose the condition

$$safe_1\, k\ x \ := \ 2 \leq k \leq x$$

which is satisfied by the initial arguments and is preserved for both recursive calls. Thus $safe_1$ is an invariant for $pfac'$, as required by the termination argument. Given $safe_1\, k\ x$, termination follows from the fact that the difference $x - k \geq 0$ is decreased upon recursion.

With the invariant $safe_1$ in place, we can use function induction on $pfac'\ k\ x$ to show that $pfac'\ k\ x$ always yields a factorization of x (that is, x is the product of the numbers in the list $pfac'\ k\ x$). Using function induction, we can also verify that the numbers in the list $pfac'\ k\ x$ appear in ascending order starting with the number 2.

It remains to show that all numbers in the list $pfac'\ k\ x$ are prime. For this to be true, the invariant $safe_1$ is not strong enough. To have this property, we need to ensure that k is a lower bound for all numbers $m \geq 2$ dividing x:

$$safe_2\, k\ x \ := \ 2 \leq k \leq x \ \wedge \ \forall m \geq 2.\ \ m \mid x \to m \geq k$$

It is not difficult to verify that $safe_2\, k\ x$ is an invariant for $pfac'\ k\ x$. With function induction we can now verify that every number in $pfac'\ k\ x$ is prime if $safe_2\, k\ x$ is satisfied.

To summarize, the essential new concept we need for the verification of the optimized prime factorization algorithm are invariants. An invariant is a precondition for the arguments of a function that is preserved under recursion. The idea then is that a function is only verified for arguments satisfying the invariant. This way we can exclude arguments for which the function does not terminate or operations are not defined (e.g., division or reminder for 0). We may see an invariant for a function as a refinement of the type of the function. Like the type of a function, an invariant for a function must be checked for every (recursive) application of the function. We may see invariant checking as a refined form of type checking that in contrast to ordinary type checking often cannot be done algorithmically.

Running times

Let us look at the running times of $pfac\, x$ and $pfac'\, 2\, x$ for the case where x is a prime number. Then $pfac'\, 2\, x$ will increment k starting from 2 until $k^2 > x$ and then finish. This give us a running time $O(\lceil \sqrt{x}\, \rceil)$. On the other hand, $pfac\, x$ will increment k starting from 2 until x in the recursion realizing *first*. This give us a running time $O(x)$. We conclude that the running time of $pfac\,x$ is quadratic in the running time of $pfac'\,2\,x$.

Exercise 7.10.1 For each of the recursive applications of $pfac'$ give the condition one has to verify to establish $safe_2$ as an invariant.

Exercise 7.10.2 Convince yourself that the proposition $safe_2\,k\;x$ is equivalent to $k \geq 2 \;\wedge\; x \geq 2 \;\wedge\; \forall m \geq 2.\;\; m \mid x \to m \geq k$.

Exercise 7.10.3 Declare the functions $pfac$ and $pfac'$ in OCaml.

Exercise 7.10.4 Consider the function $f : \mathbb{Z} \to \mathbb{Z}$ defined as

$$f(x) \;:=\; \text{IF } x = 7 \text{ THEN } x \text{ ELSE } f(x-1)$$

Give a predicate $p(x)$ holding exactly for the numbers f terminates on. Convince yourself that p is an invariant for f.

Exercise 7.10.5 Consider the function $f : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ defined as

$$f\;x\;y \;:=\; \text{IF } x = 0 \wedge y = 27 \text{ THEN } x + y \text{ ELSE } f(x-1)(y+1)$$

Give an invariant pxy for f holding exactly for the numbers f terminates on.

Exercise 7.10.6 Consider the function $f : \mathbb{N}^+ \to \mathbb{N} \to \mathbb{N}$ defined as

$$f\;n\;k \;:=\; \text{IF } k/n = 2 \text{ THEN } k \text{ ELSE } f\,n\,(k+1)$$

a) Give an invariant $p\,nk$ for f holding exactly for the numbers f terminates on.

b) Give an invariant $p\,nk$ for f that is strong enough to show $f n\,0 = 2{\cdot}n$.

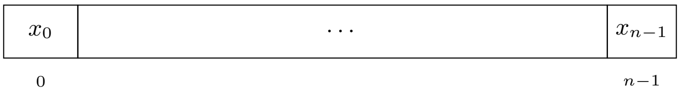
8 Arrays

Arrays are mutable data structures representing computer memory within programming languages. The mutability of arrays is in contrast to the immutability of values (e.g., numbers, lists). Mutability is an intrinsic aspect of physical objects while immutability is a characteristic feature of mathematical objects (a clock changes its state, but a number has no state that could change).

Arrays are made available through special values acting as names and through operations acting on arrays identified by array values. Due to their mutability, arrays don't have a direct mathematical representation, making functions using arrays more difficult to reason about.

8.1 Basic Array Operations

Arrays may be described as fixed-length mutable sequences with constant-time field access and update.



The length and the element type of an array are fixed at *allocation time* (the time an array comes into existence). The *fields* or *cells* of an array are boxes holding values known as *elements* of the array. The fields are numbered starting from 0. One speaks of the *indices* or *positions* of an array. The key difference between lists and arrays is that arrays are mutable in that the values in their fields can be overwritten by an update operation. Arrays are represented by special *array values* acting as immutable names for the mutable array objects in existence. Each time an array is allocated in the computer's memory, a fresh name identifying the new array is chosen.

OCaml provides arrays through the standard structure *Array*. The operation

$$\mathit{Array.make} : \ \forall\alpha.\ \mathit{int} \to \alpha \to \mathit{array}(\alpha)$$

allocates a fresh array of a given length where all fields are initialized with a given value. An *array type* has the form $\mathit{array}(t)$ and covers all arrays whose elements have type t, no matter what the length of the array is. There is also a special constructor type *unit*

$$\mathit{unit} \ ::= \ ()$$

with a single constructor () serving as target type of the array update operation. There are access and update operations for the fields of an array:

$$\begin{aligned} \mathit{Array.get} &: \ \forall\alpha.\ \mathit{array}(\alpha) \to \mathit{int} \to \alpha \\ \mathit{Array.set} &: \ \forall\alpha.\ \mathit{array}(\alpha) \to \mathit{int} \to \alpha \to \mathit{unit} \end{aligned}$$

An operation $\mathit{Array.get}\,a\,i$ yields the ith element of a (i.e., the value in the ith field of the array a), and an operation $\mathit{Array.set}\,a\,i\,x$ updates the ith element of a to x (i.e., the value in the ith field of the array a is replaced with the value x). Both operations raise an invalid argument exception

```
Exception: Invalid_argument "index out of bounds"
```

if the given index is not admissible for the given array.

OCaml supports convenient notations for field access and field update:

$$\begin{aligned} e_1.(e_2) \quad &\rightsquigarrow \quad \mathit{Array.get}\ e_1\, e_2 \\ e_1.(e_2) \leftarrow e_3 \quad &\rightsquigarrow \quad \mathit{Array.set}\ e_1\, e_2\, e_3 \end{aligned}$$

Let's step through a first demo:

```
let a = Array.make 3 7
let v_before = a.(1)
let _ = a.(1) <- 5
let v_after = a.(1)
```

The 1st declaration allocates an array a whose initial state is $[7, 7, 7]$. The 2nd declaration binds the identifier *v_before* to 7 (the 1st element of a at this point). The 3rd declaration updates a to the state $[7, 5, 7]$. The 4th declaration binds the identifier *v_after* to 5 (the 1st element of a at this point). Note that the identifier *v_before* is still bound to 7.

There is also a basic array operation that yields the length of an array:

$$\mathit{Array.length} : \ \forall\alpha.\ \mathit{array}\,(\alpha) \to \mathit{int}$$

As a convenience, OCaml offers the possibility to allocate an array with a given initial state:

```
let a = [|1;2;3|]
```

This declaration binds a to a new array of length 3 whose fields hold the values $1, 2, 3$. The declaration may be seen as derived syntax for

```
let a = Array.make 3 1
let _ = a.(1) <- 2
let _ = a.(2) <- 3
```

Sequentializations

OCaml offers expressions of the form

$$e_1; e_2$$

called **sequentializations**, which execute e_1 before e_2 and yield the value of e_2. The value of e_1 is ignored. The reason e_1 is executed is that it may update an array, a so-called **side effect**. OCaml issues a warning in case the type of e_1 is not *unit*. We can now write the expression

```
let a = Array.make 3 1 in a.(1) <- 2; a.(2) <- 3; a
```

to explain the expression $[|1; 2; 3|]$. We remark that the sequentialization operator $e_1; e_2$ is treated as a right-associative operator so that iterated sequentializations $e_1; \dots; e_n$ can be written without parentheses.

Sequentializations may be considered as a derived form that can be expressed with a let expression $\text{LET}\ \_ = e_1\ \text{IN}\ e_2$.

8.2 Conversion between Arrays and Lists

The state of an array can be represented as a list, and given a list, one can allocate a fresh array whose state is given by the list. OCaml provides the conversions with two predefined functions:

$$\begin{aligned} \mathit{Array.to\_list} &: \ \forall\alpha.\ \mathit{array}(\alpha) \to \mathcal{L}(\alpha) \\ \mathit{Array.of\_list} &: \ \forall\alpha.\ \mathcal{L}(\alpha) \to \mathit{array}(\alpha) \end{aligned}$$

At this point we need to say that OCaml provides an immutable empty array for every element type (i.e., an array of length 0 with no fields).

The conversion from arrays to lists can be defined with the operations *Array.length* and *Array.get*:

```
let to_list a =
  let rec loop i l =
    if i < 0 then l else loop (i-1) (a.(i)::l)
  in loop (Array.length a - 1) []
```

The loop function shifts a pointer i from right to left through the array and collects the values in the scanned field in a list l.

The conversion from lists to arrays can be defined with the operations *Array.make* and *Array.set*:

```
let of_list l =
  match l with
  | [] -> [||]
  | x::l ->
    let a = Array.make (List.length l + 1) x in
    let rec loop l i =
       match l with
         | [] -> a
         | x::l -> (a.(i) <- x; loop l (i + 1))
    in loop l 1
```

Here the loop function recurses on the list and shifts a pointer from left to right through the array to copy the values from the list into the array. There is the subtlety that given an empty list we can obtain the corresponding empty array only with the expression [| |] since *Array.make* requires a default value even if we ask for an empty array. Thus we consider [| |] as a native constant rather than a derived form. In fact, the empty array [| |] is an immutable value.

Note that the worker functions in both conversion functions are tail-recursive. We reserve the identifier *loop* for tail-recursive worker functions.

Exercise 8.2.1 (Cloning)
Declare a function $\forall\alpha.\ array(\alpha) \to array(\alpha)$ cloning arrays. Make sure your function also works for empty arrays. OCaml offers a cloning function as *Array.copy*.

Exercise 8.2.2 OCaml offers a predefined function *Array.init* such that *Array.init* n f allocates an array with initial state *List.init* n f. Declare such an init function for arrays using neither *List.init* nor *Array.of_list*. Use only tail recursion.

Exercise 8.2.3 Declare a function that yields the sum of the elements of an array over *int*.

8.3 Binary Search

Given a sorted array a and a value x, one check whether x occurs in a with a running time that is at most logarithmic in the length of the array. The trick is to check whether the value in the middle of the array is x, and in case it is not, continue with either the left half or the right half of the array. The calculation of the running time assumes that field access is constant time, which is the case for arrays. One speaks of binary search to distinguish the algorithm from linear search, which has linear running time.[1] We have seen binary search before in generalized form (Section 6.5).

Binary search is usually realized such that in the positive case it returns an index where the element searched for occurs.

To ease the presentation of binary search and related algorithms, we work with a three-valued constructor type

```
type comparison = LE | EQ | GR
```

and a polymorphic comparison function

```
let comp x y : comparison =
  if x < y then LE
  else if x = y then EQ else GR
```

The binary search function

$$bsearch : \ \forall \alpha.\ array(\alpha) \to \alpha \to \mathcal{O}(\mathbb{N})$$

can now be declared as follows:

```
let bsearch a x : int option =
  let rec loop l r =
    if l > r then None
    else let m = (l+r)/2 in
      match comp x a.(m) with
      | LE -> loop l (m-1)
      | EQ -> Some m
      | GR -> loop (m+1) r
  in loop 0 (Array.length a - 1)
```

Note that the worker function *loop* is tail-recursive and that the segment of the array to be considered is at least halved upon recursion.

Binary search satisfies two invariants:

1. The state of the array agrees with the initial state (that is, remains unchanged).
2. All elements to the left of l are smaller than x and all elements to the right of r are greater than x.

Given the invariants, the correctness of *bsearch* is easily verified.

[1] Linear search generalizes binary search in that it doesn't assume the array is sorted.

Exercise 8.3.1 (Linear search) Declare a function

$$lsearch : \ \forall\alpha.\ array(\alpha) \to \alpha \to \mathcal{O}(\mathbb{N})$$

that checks whether a value occurs in an array and in the positive case yields an index where the value occurs. The running time of *lsearch* should be $O(n)$ where n is the length of the array.

Exercise 8.3.2 (Binary search for increasing functions) Declare a function

$$search : \ \forall\alpha.\ (\mathbb{N} \to \alpha) \to \mathbb{N} \to \alpha \to \mathcal{O}(\alpha)$$

that given an increasing function f, a number n, and a value x, checks whether there is some $i \leq n$ such that $f(i) = x$ using at most $O(\log n)$ calls of f. A function f is increasing if $f(i) \leq f(j)$ whenever $i \leq j$.

8.4 Reversal

There is an elegant algorithm reversing the elements of an array in time linear in the length of the array. The algorithm is based on a function

$$swap : \ \forall\alpha.\ array(\alpha) \to \mathbb{N} \to \mathbb{N} \to unit$$

swapping the values at two given positions of a given array:

```
let swap a i j : unit =
  let x = a.(i) in a.(i) <- a.(j); a.(j) <- x
```

The algorithm now reverses a segment $i < j$ in an array by swapping the values at i and j and continuing with the smaller segment $(i+1, j-1)$:

```
let reverse a : unit =
  let rec loop i j =
    if i > j then ()
    else (swap a i j; loop (i+1) (j-1))
  in loop 0 (Array.length a - 1)
```

One may see i and j as pointers to fields of the array that are moved inside from the leftmost and the rightmost position.

Function *reverse* satisfies two invariants:

1. The state of the array agrees with the initial state up to repeated swapping.
2. The segments $l \leq i$ and $j \leq r$ agree with the reversed initial state.

Exercise 8.4.1 Declare a function $rotate : \ \forall\alpha.\ array(\alpha) \to unit$ rotating the elements of an array by shifting each element by one position to the right except for the last element, which becomes the new first element. For instance, rotation changes the state $[1, 2, 3]$ of an array to $[3, 1, 2]$.

8.5 Sorting

A naive sorting algorithm for arrays is **selection sort**: To sort a segment $i < j$, one swaps the least element of the segment into the first position of the segment and then continues sorting the smaller segment $i+1 \leq j$.

```
let ssort a : unit =
  let r = Array.length a - 1 in
  let rec min k j : int =
    if k >= r then j
    else if a.(k+1) < a.(j) then min (k+1) (k+1)
    else min (k+1) j
  in let rec loop i : unit =
       if i >= r then ()
       else (swap a (min i i) i; loop (i+1))
  in loop 0
```

Function *ssort* satisfies the following invariants:

1. The state of the array agrees with the initial state up to repeated swapping.
2. The segment $0 < l$ is sorted.
3. Every element of the segment $0 < l$ is less or equal than every element of the segment $l \leq r$.
4. The element at j is the least element of the segment $i \leq k$.

The running time of selection sort is quadratic in the length of the array. Note that the two worker functions *min* and *loop* use only tail recursion.

A prominent sorting algorithm for arrays is **quick sort**. The top level of quick sort is reminiscent of merge sort: Quick sort partitions an array into two segments using swapping, where all elements of the left segment are smaller than all elements of the right segment. It then sorts the two segments recursively.

```
let qsort a =
  let partition l r = ?···? in
  let rec qsort' l r =
    if l >= r then ()
    else let m = partition l r in
      qsort' l (m-1); qsort' (m+1) r
  in qsort' 0 (Array.length a - 1)
```

Given $l < r$, *partition* yields m such that $l < m < r$ and

1. $a.(i) < a.(m)$ for all i such that $l \leq i \leq m - 1$
2. $a.(m) \leq a.(i)$ for all i such that $m + 1 \leq i \leq r$

It is easy to verify that *qsort* terminates and sorts the given array if *partition* has the properties specified. Moreover, if *partition* splits a segment in two segments of the same size (plus/minus 1) and runs in linear time in the length of the given segment, the running time of *qsort* is $O(n \log n)$ (following the argument for merge sort).

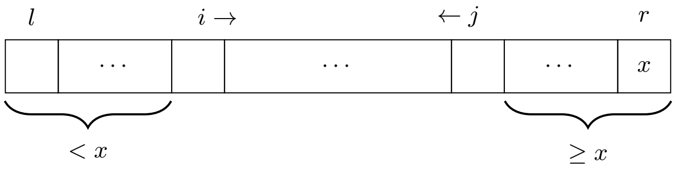
We give a simple partitioning algorithm that chooses the rightmost element of the segment as **pivot element** that will be swapped to the dividing position m once it has been determined:

```
let partition l r =
  let x = a.(r) in
  let rec loop i j =
    if j < i
    then (swap a i r; i)
    else if a.(i) < x then loop (i+1) j
    else if a.(j) >= x then loop i (j-1)
    else (swap a i j; loop (i+1) (j-1))
  in loop l (r-1)
```

The function expects a segment $l < r$ in a and works with two pointers i and j initialized as l and $r - 1$ and satisfying the following invariants for the pivot element $x = a.(r)$:

1. $l \leq i \leq r$ and $l - 1 \leq j < r$
2. $a.(k) < x$ for all positions k left of i (meaning $l \leq k < i$)
3. $x \leq a.(k)$ for all postions k right of j (meaning $j < k \leq r$)

The situation may be described with the following diagram:



The pointers are moved further inside the segment as long as the invariants are preserved. When $j < i$, the dividing position is chosen as i and the elements at i and r are swapped. If $i \leq j$, we have $a.(j) < x$ and $x \leq a.(i)$. Now the elements at i and j are swapped and both pointers are moved inside by one position.

The partitioning algorithm is subtle in that it cannot be understood without the invariants. The algorithm moves the pointers i and j inside where the moves must respect the invariants and the goal is $j < i$. There is one situation where a swap preserves the invariants and enables a move of both i and j. As long as $i \leq j$, a move is always possible.

The given partitioning algorithm may yield an empty segment and a segment only by one position shorter than the given segment, which means that the worst case running time of *qsort* will be quadratic. Better partitioning algorithms can be obtained by first swapping a cleverly determined pivot element into the rightmost position of the given segment.

Exercise 8.5.1 Declare a function $\mathit{sorted} : \forall\alpha.\ \mathit{array}(\alpha) \to \mathbb{B}$ that checks whether an array is sorted. Do not convert to lists. Also write a function that checks whether an array is strictly sorted (i.e., sorted and no element occurring twice).

Exercise 8.5.2 Declare functions that for a nonempty array yield an index such that the value at the index is minimal (or maximal) for the array.

Exercise 8.5.3 Declare a variant of selection sort swapping the greatest element to the right.

Exercise 8.5.4 (Quick sort with middle element as pivot) The performance of quick sort for sorted arrays can be dramatically improved by swapping the middle and the rightmost element before partitioning (thus making the middle element the pivot). Realize the improved quick sort algorithm in OCaml and check the performance claim for the array $\mathit{Array.init}\ 100000\ (\lambda i.i)$.

Exercise 8.5.5 (Quick sort with median-of-three pivots) Some implementations of quick sort choose the pivot as the median of the leftmost, middle, and rightmost element of the segment (the median of three numbers is the number that is in the middle after sorting). This can be realized by swapping the median into the rightmost position before the actual partition starts.

a) A segment in an array has the *median property* if the rightmost element of the segment is the median of the leftmost, middle, and rightmost element of the segment. Declare a function *ensure_median* that given an array and a nonempty segment establishes the median property by possibly swapping two elements.
   Hint: The necessary case analysis as amazingly tricky. Do a naive case analysis whether the leftmost or the rightmost or the middle element is the median using lazy boolean connectives.

b) Declare a quick sort function where partition establishes the median property before it partitions.

8.6 Equality for Arrays

OCaml's polymorphic equality test

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

considers two arrays as equal if their states are equal. However, different arrays may have the same state. There is a second polymorphic equality test

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

testing the equality of two array values without looking at their state:

```
let a = Array.make 2 1
let b = Array.make 2 1
let test2 = (a = b)               (* true *)
let test1 = (a == b)              (* false *)
let test3 = (a.(1) <- 2; a=b)     (* false *)
let test4 = (a.(1) <- 1; a=b)     (* true *)
```

We will refer to $=$ as **structural equality** and to $==$ as **physical equality**. The name physical equality refers to the fact that the operator $==$ when applied to arrays tests whether they are the same physical object (i.e., are stored at the same address in the computer memory). For arrays and other mutable objects, physical equality implies structural equality. We will consider physical equality only for arrays.

8.7 Execution Order

As soon as mutable objects are involved, the order in which expressions and declarations are executed plays an important role. Consider for instance a get and a set operation on the field of an array. If the get operation is executed before the set operation, one gets the old value of the field, while if the set operation is executed before the get operation, one gets the new value of the field.

Like most programming languages, OCaml fixes the execution order only for certain expressions:

- Given $e_1; e_2$, subexpression e_1 is executed before e_2.
- Given LET e_1 IN e_2, subexpression e_1 is executed before e_2.
- Given IF e_1 THEN e_2 ELSE e_2, subexpression e_1 is executed before e_2 or e_3, and only one of the two is executed.

In contrast, the execution order of operator applications $e_1\, o\, e_2$, function applications $e_0\, e_1 \dots e_n$, and tuple expressions $(e_1, \dots, e_n)$ is not fixed and the interpreter can choose which order it realizes. This design decision gives compilers[2] more freedom in generating efficient machine code.

You can find out more about the execution order your interpreter realizes by executing the following declarations:

```
let test = invalid_arg "1"  + invalid_arg "2"
let test = (invalid_arg "1", invalid_arg "2")
let test = (invalid_arg "1"; invalid_arg "2")
```

Depending on the execution order, the declarations raise different exceptions.

We remark that execution order as we discuss it here is only observable through side effects (i.e., raised exceptions and array updates).

One distinguishes between **pure** and **impure functional languages**. Exceptions, mutable objects, and physical equality make functional languages impure. Pure functional languages are very close to mathematical constructions, while impure functional languages integrate mutable objects, exceptions, and aspects concerning their implementation on computers.

8.8 Cells

OCaml provides single field arrays called **cells** using a type family $\mathit{ref}(t)$ and three operations:

$$\begin{array}{rcll} \mathit{ref} & : & \forall\alpha.\ \alpha \to \mathit{ref}(\alpha) & \qquad\qquad \textbf{allocation} \\ ! & : & \forall\alpha.\ \mathit{ref}(\alpha) \to \alpha & \qquad\qquad \textbf{dereference} \\ := & : & \forall\alpha.\ \mathit{ref}(\alpha) \to \alpha \to \mathit{unit} & \qquad\qquad \textbf{assignment} \end{array}$$

The allocation operation ref creates a new cell initialized with a value given; the dereference operation ! yields the value stored in a cell; and the assignment operation := updates a cell with a given value (i.e., the value in the cell is replaced with a given value). Cell values (i.e., the names of cells) are called **references**.

With cells we can declare a function

$$\mathit{next} : \ \mathit{unit} \to \mathit{int}$$

that counts how often it was called:

[2] A compiler is a tool translating programs in a high-level language (e.g., OCaml) into machine language.

```
let c = ref 0
let next () = (c := !c + 1; !c)
```

Now the declarations

```
let test1 = next ()                                 ?1?
let test2 = (next (), next ())                      ?(3, 2)?
let test3 = next () + next () + next ()             ?15?
```

bind their identifiers to the values given at the right. The second declaration assumes that the tuple expressions is executed from right to left.

We will see the function *next* as a stateful enumerator of the sequence $1, 2, 3, \ldots$. We can change the declaration of *next* such that the function encapsulates the cell storing its state:

```
let next = let c = ref 0 in
  fun () -> (c := !c + 1; !c)
```

This version is preferable for situations where only *next* is supposed to manipulate its state.

Exercise 8.8.1 Declare a function $newEnum : unit \to unit \to int$ such that $newEnum$ () yields a function enumerating $1, 2, 3, \ldots$. Make sure that each call $newEnum$ () yields a fresh enumerator function starting from 1.

Exercise 8.8.2 Declare a function $unit \to int$ that enumerates the square numbers $0, 1, 4, 9, 16, \ldots$.

Exercise 8.8.3 Declare a function $unit \to int$ that enumerates the factorials $1, 1, 2, 6, 24, 120, \ldots$.

Exercise 8.8.4 Declare a function $unit \to int$ that enumerates the Fibonacci numbers $0, 1, 1, 2, 3, 5, \ldots$.

Exercise 8.8.5 Complete the declaration of *next* such that

```
let next = ?···?
let x = next ()
let y = next () + next () + next ()
```

binds x to 2 and y to 28.

Exercise 8.8.6 (Enumerator with reset function)
Declare a function

$$unit \to (unit \to int) \times (unit \to unit)$$

that on each application yields a pair consisting of an enumerator for $1, 2, 3 \ldots$ and a function resetting the enumerator to 1.

Exercise 8.8.7 Implement cells with arrays using the identifiers *cell*, *alloc*, *get*, and *set*.

8.9 Mutable Objects Mathematically

Mutable objects don't have a direct mathematical representation. What we can do is represent the state of a mutable object as an immutable value. The other aspect of an immutable object we can represent mathematically is a name fixing its identity. We may then describe the allocation of a mutable object as choosing a fresh name (a name that has not been used so far) and an initial state associated with this name. We thus represent the state of an object as a pair consisting of the name of the object and the current state of the object. As names we may use natural numbers.

9 Data Structures

In this chapter we consider array-based data structures known as stacks, queues, and heaps that are essential for machine-oriented programming. We explain how values of constructor types are stored in heaps using linked blocks, thus providing a first model for the space consumption of OCaml functions. We implement the data structures studied in this chapter using structure and signature declarations, two OCaml features we have not seen so far.

9.1 Structures and Signatures

OCaml offers a facility to bundle declarations into a **structure** and to restrict the visibility of the declarations with a **signature**. As a straightforward example we consider a signature and a structure for cells. We declare the signature as follows:

```
module type CELL = sig
  type 'a cell
  val make : 'a -> 'a cell
  val get  : 'a cell -> 'a
  val set  : 'a cell -> 'a -> unit
end
```

The keywords in the first line are explained by the fact that OCaml refers to structures also as **modules**, and to signatures also as **module types**. We also note that the signatures *CELL* keeps the type constructor *cell* abstract, making it possible to give different implementations for cells.

An obvious implementation of the signature *CELL* would just mirror OCaml's native cells. For the purpose of demonstration, we choose an implementation of *CELL* with arrays of length 1. We realize the implementation with a declaration of a structure *Cell* realizing the signature *CELL*.

```
module Cell : CELL = struct
  type 'a cell = 'a array
  let make x = Array.make 1 x
  let get c = c.(0)
  let set c x = c.(0) <- x
end
```

There is type checking that ensures that the structure implements the fields the signature requires with the required types. Before checking compliance with the signature, the declarations of the structure are checked as if they would appear at the top level.

Once the structure *Cell* is declared, we can use its fields with the dot notation:

```
let enum = let c = Cell.make 0 in
  fun () -> let x = Cell.get c in Cell.set c (x+1); x
```

The declaration gives us an enumerator function for $0, 1, 2, \ldots$.

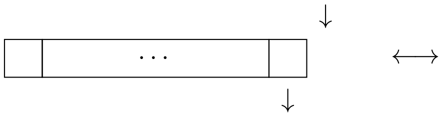
Exercise 9.1.1 Implement the signature *CELL* with OCaml's predefined reference cells.

Exercise 9.1.2 Arrays can be implemented with cells and lists. Declare a signature with a type family $array(\alpha)$ and the operations *make*, *get*, *set*, and *length*, and implement the signature with a structure not using arrays. We remark that expressing arrays with cells and lists comes at the expense that *get* and *set* cannot be realized as constant-time operations.

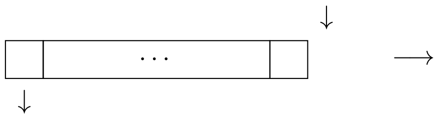
9.2 Stacks and Queues

An **agenda** is a mutable data structure holding a sequence of values called **entries**. The entries are entered into the agenda one after the other, and the sequence reflects the order in which the entries were entered. Depending on the kind of agenda, there are specific operations for inspecting and deleting entries.

Two prominent kinds of agendas are stacks and queues. A **stack** realizes a last-in-first-out (LIFO) policy, where only the most recent entry can be removed.



A **queue** realizes a first-in-first-out (FIFO) policy, where only the oldest entry can be removed.



We may think of a stack as oscillating at the right getting larger and smaller with insertion and removal, and of a queue as moving to the right with insertion and as shrinking at the left with removal.

For stacks we may have the signature

```
module type STACK = sig
  type 'a stack
  val make   : 'a -> 'a stack
  val push   : 'a stack -> 'a -> unit
  val pop    : 'a stack -> unit
  val top    : 'a stack -> 'a
  val height : 'a stack -> int
end
```

and the implementation

```
module Stack : STACK = struct
  type 'a stack = 'a list ref
  exception Empty
  let make x = ref [x]
  let push s x = s:= x :: !s
  let pop s = match !s with
    | [] -> raise Empty
    | _::l -> s:= l
  let top s = match !s with
    | [] -> raise Empty
    | x::_ -> x
  let height s = List.length (!s)
end
```

The operation *make* allocates a stack with a given single element. One says that *push* puts an additional element on the stack, *pop* removes the topmost element of the stack, *top* yields the topmost element of the stack, and *height* yields the height of the stack.

A stack may become empty after sufficiently many pop operations. Note that the structure *Stack* declares an exception *Empty* that is raised by *pop* if applied to an empty stack.

We require that *make* is given an initial element so that the element type of the stack is fixed. Alternatively one can have a make operation that throws away the given element.

For queues we may have the signature

```
module type QUEUE = sig
  type 'a queue
  val make   : 'a -> 'a queue
  val insert : 'a queue -> 'a -> unit
  val remove : 'a queue -> unit
  val first  : 'a queue -> 'a
  val length : 'a queue -> int
end
```

and the implementation

```
module Queue : QUEUE = struct
  type 'a queue = 'a list ref
  exception Empty
  let make x = ref [x]
  let insert q x = q:= !q @ [x]
  let remove q = match !q with
    | [] -> raise Empty
    | _::l -> q:= l
  let first q = match !q with
    | [] -> raise Empty
    | x::_ -> x
  let length q = List.length (!q)
end
```

The operation *make* allocates a queue with a given single element. One says that *insert* appends a new element to the queue, *remove* removes the first element of the queue, *first* yields the first element of the queue, and *length* yields the length of the queue.

As it comes to efficiency, our naive implementations of stacks and queues can be improved. For queues, one would like that *insert* takes constant time rather than linear time in the length of the queue.

9.3 Array-Based Stacks

We will now realize a stack with an array. The elements of the stack are written into the array, and the height of stack is bounded by the length of the array. This approach has the advantage that a push operation does not require new memory but reuses space in the existing array. Such memory-aware techniques are important when data structures are realized close to the machine level.

We will realize a **bounded stack** with a structure and a single array. To make this possible, we fix the element type of the stack to machine integers. We declare the signature

```
module type BSTACK = sig
  val empty : unit -> bool
  val full  : unit -> bool
  val push  : int -> unit
  val pop   : unit -> unit
  val top   : unit -> int
end
```

and realize the stack with the following structure:

```
module S : BSTACK = struct
  let size = 100
  let a = Array.make size 0
  let h = ref 0  (* height of stack *)
  exception Empty
  exception Full
  let empty () = !h = 0
  let full () = !h = size
  let push x = if full() then raise Full else (a.(!h) <- x; h:= !h
       + 1)
  let pop () = if empty() then raise Empty else h:= !h - 1
  let top () = if empty() then raise Empty else a.(!h -1)
end
```

The height of the stack will oscillate in the array depending on the push and pop operations executed. If the size of the array is exhausted, the exception *Full* will be raised.

Note that with the structure approach stacks are realized as structures. Since structures are not values, structures cannot be passed as arguments to functions nor can they be the result of functions.

Note that all operations of bounded stacks are constant time.

Exercise 9.3.1 (Array-based stacks as values)
Array-based stacks can also be represented as values combining an array with a cell holding the current height of the stack. Modify the signature *Stack* from Section 9.2 such that *make* takes the maximal height of the stack object to be allocated as argument, and implement the signature with a structure such that all operations but *make* are constant time. Note that this approach provides a family of stack types where the element type can be freely chosen.

Exercise 9.3.2 Assume a structure $S : BSTACK$.

a) Declare a function $toList : unit \to \mathcal{L}(int)$ that yields the state of the stack S as a list with the most recent entry appearing first.

b) Declare a function $ofList : \mathcal{L}(int) \to unit$ that updates the state of the stack S to the sequence given by the list.

Use only operations declared in the signature *BSTACK*. OCaml's type checking will ensure this if S is declared with signature *BSTACK*.

As an interesting variant, extend the signature *BSTACK* with the operations *toList* and *ofList* so that they can be implemented within the structure with constant running time.

Exercise 9.3.3 Extend signature *BSTACK* and structure S with an operation $height : unit \to int$ that yields the current height of the stack.

9.4 Circular Queues

Similar to what we have seen for stacks, length-bounded queues can be realized within arrays. This time we use the signature

```
module type BQUEUE = sig
  val empty  : unit -> bool
  val full   : unit -> bool
  val insert : int -> unit
  val remove : unit -> unit
  val first  : unit -> int
end
```

and implement it with the structure

```
module Q : BQUEUE = struct
  let size = 100
  let a = Array.make size 0
  let s = ref 0  (* position of first entry *)
  let l = ref 0  (* number of entries *)
  exception Empty
  exception Full
  let empty () = !l = 0
  let full () = !l = size
  let pos x = x mod size
  let insert x = if full() then raise Full
    else (a.(pos(!s + !l)) <- x; l:= !l + 1)
  let remove () = if empty() then raise Empty
    else (s:= pos (!s + 1); l:= !l - 1)
  let first () = if empty() then raise Empty else a.(!s)
end
```

As with stacks, there is no recursion and all operations are constant time. This time we represent the queue with an array and two cells holding the position of the first entry and the number of entries in the queue. To fully exploit the capacity of the array, the sequence of entries may start at some position of the array and continue at position 0 of the array if necessary. This way the sequence of entries can move through the array as if the array was a ring. Technically, this is realized with the constant-time remainder operation.

The ring implementation of queues described above is known as a **circular buffer** or as a **ring buffer**.

Exercise 9.4.1 (Array-based queues as values)

Similar to what we have seen for stacks in exercise 9.3.1, array-based queues can be represented as values combining an array with two cells holding the position of the first entry and the number of entries in the queue. Modify the signature *Queue* from Section 9.2 such that *make* takes the maximal length of the queue object to be allocated as argument, and implement the signature with a structure such that all operations but *make* are constant time.

Exercise 9.4.2 Assume a structure $Q : BQUEUE$.

a) Declare a function $toList : unit \to \mathcal{L}(int)$ that yields the state of the queue Q as a list with the most recent entry appearing first.

b) Declare a function $ofList : \mathcal{L}(int) \to unit$ that updates the state of the queue Q to the sequence given by the list.

Use only operations declared in the signature *BQUEUE*. OCaml's type checking will ensure this if Q is declared with signature *BQUEUE*.

As an interesting variant, extend the signature *BQUEUE* with the operations *toList* and *ofList* so that they can be implemented within the structure with constant running time.

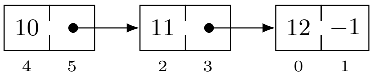
9.5 Block Representation of Lists in Heaps

Values of constructor types like lists, AB-trees, and expressions must be stored in computer memory. We demonstrate the underlying technique by showing how values of constructor types can be represented in an array-based data structure called a **heap**.

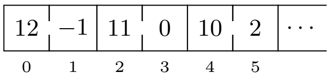
A heap is an array where mutable fixed-length sequences of machine integers called **blocks** are stored. For instance, the list

$$[10, 11, 12]$$

may be stored with three blocks connected through links:

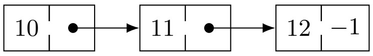


Each of the 3 blocks has two fields, where the first field represents a number in the list and the second field holds a **link**, which is either a link to the next block or −1 representing the empty list. The numbers below the fields are the indices in the array where the fields are allocated. The fields of a block are allocated consecutively in the heap array. The **address** of a block is the index of its first field in the heap array. Links always point to blocks and are represented through the addresses of the blocks. Thus the above linked block representation of the list $[10, 11, 12]$ is represented with the following segment in the heap:

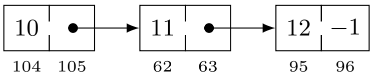


Note that the representation of the empty list as −1 is safe since the addresses of blocks are nonnegative numbers (since they are indices of the heap array).

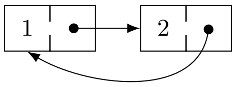
Given a linked block representation of a list



we can freely choose where the blocks are allocated in the heap. There is no particular order to observe, except that the fields of a block must be allocated consecutively. We might for instance choose the following addresses:



It is also perfectly possible to represent **circular lists** like



We emphasize that mathematical lists cannot be circular. It is a consequence of the physical block representation that circular lists come into existence. Lists as they appear with the block representation are commonly called **linked lists**.

Exercise 9.5.1 How many blocks and how many fields are required to store the list $[1, 2, 3, 4]$ in a heap using the linked block representation?

9.6 Realization of a Heap

We will work with a heap implementing the signature

```
module type HEAP = sig
  type address = int
  type index = int
  val alloc   : int -> address
  val get     : address -> index -> int
  val set     : address -> index -> int -> unit
  val release : address -> unit
end
```

The function *alloc* allocates a new block of the given length in the heap and returns the address of the block. Blocks must have at least length 1 (that is, at least one field). The function *get* yields the value in the field of a block, where the field is given by its index in the block, and the block is given by its address in the heap. The function *set* replaces the value in the field of a block. Thus we may think of blocks as mutable objects. The function *release* is given the address of an allocated block and deallocates this block and all blocks allocated after this block. We implement the heap with the following structure:

```
module H : HEAP = struct
  let maxSize = 1000
  let h = Array.make maxSize (-1)
  let s = ref 0  (* current size of heap *)
  exception Address
  exception Full
  type address = int
  type index = int
  let alloc n = if n < 1 then raise Address
    else if !s + n > maxSize then raise Full
    else let a = !s in s:= !s + n; a
  let check a = if a < 0 || a >= !s then raise Address else a
  let get a i = h.(check(a+i))
  let set a i x = h.(check(a+i)) <-x
  let release a = s:= check a
end
```

Note that we use the helper function *check* to ensure that only allocated addresses are used by *set*, *get*, and *release*. The exception *address* signals an address error that has occurred while allocating or accessing a block.

Given the heap H, we declare a high-level allocation function

$$alloc' : \ \mathcal{L}(int) \to address$$

allocating a block for a list of integers:

```
let alloc' l =
  let a = H.alloc (List.length l) in
  let rec loop l i = match l with
    | [] -> a
    | x::l -> H.set a i x; loop l (i+1)
  in loop l 0
```

Note that the low level allocation function ensures that an allocation attempt for the empty list fails with an address exception.

It is now straightforward to declare a function

$$putlist : \ \mathcal{L}(int) \to address$$

allocating a linked block representation of a given list in the heap:

```
let rec putlist l = match l with
  | [] -> -1
  | x::l -> alloc' [x; putlist l]
```

Next we declare a function

$$getlist : \ address \to \mathcal{L}(int)$$

that given the address of the first block of a list stored in the heap recovers the list as a value:

```
let rec getlist a =
  if a = -1 then []
  else  H.get a 0 :: getlist (H.get a 1)
```

Note that *putlist* and *getlist* also work for the empty list. We have

$$getlist\,(putlist\ l) \ = \ l$$

for every list l whose block representation fits into the heap.

Exercise 9.6.1 Given an expression allocating the list $[1,2,3]$ in the heap H.

Exercise 9.6.2 Given an expression allocating a circular list $[1,2,\dots]$ as shown in Section 9.5 in the heap H.

Exercise 9.6.3 Declare a function that given a number $n>0$ allocates a cyclic list $[1,2,\dots,n,\dots]$ in the heap H.

Exercise 9.6.4 Declare a function that checks whether a linked list in the heap H is circular.

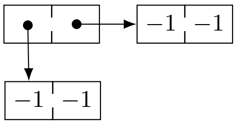
9.7 Block Representation of AB-Trees

Block representation works for constructor types in general. The idea is that a constructor application translates into a block holding the arguments of the constructor application. If there are no arguments, we don't need a block and use a negative number to identify the nullary constructor. If there are arguments, we use a block having a field for every argument. If there are several constructors taking arguments, the blocks will also have a **tag field** holding a number identifying the constructor. This will work for constructor types where the arguments of the constructors are either numbers, booleans, or values of constructor types.

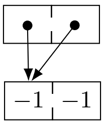
As example we consider AB- and ABC-trees (**??**). For AB-trees we don't need a tag field since there is only one constructor taking arguments. Thus the block representation of AB-trees is as follows:

1. A tree A is represented as -1.
2. A tree $B(t_1, t_2)$ is represented as a block with 2 fields carrying the addresses of t_1 and t_2.

For instance, the block representation of the tree $B(B(A, A), B(A, A))$ looks as follows:



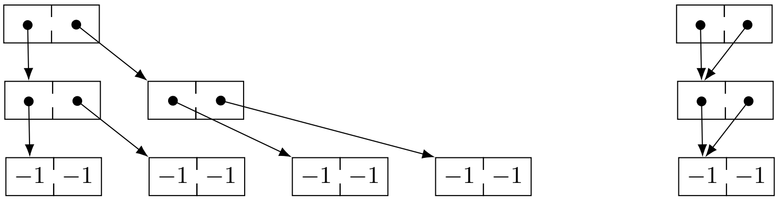
At this point we may ask why in the tree $B(A, A)$ is represented with two boxes rather than just one:



In fact, if the boxes represent immutable values, there is no need for this double allocation. One says that the more compact representation realizes **structure sharing**. For the maximal tree of depth 3

$$B(\ B(B(A, A), B(A, A))\ ,\ B(B(A, A), B(A, A)))$$

the naive representation and the representation with maximal structure sharing look as follows:



We now see that the block representation with no structure sharing is exponentially larger than the block representation with maximal structure sharing.

Exercise 9.7.1 Consider the ABC-tree

$$B(\ C(A, B(A, A)),\ B(A, A))$$

with a block representation employing the tags 0 and 1 for the constructors B and C.

a) Draw a block representation without structure sharing.

b) Draw a block representation with structure sharing.

c) Give an expression allocating the tree with structure sharing.

Exercise 9.7.2 Declare functions such that $getTree\,(putTree\ t) = t$ holds for all AB-trees that fit into the heap.

Exercise 9.7.3 ABC-trees can be stored in a heap by a representing trees of the forms $B(t_1, t_2)$ and $C(t_1, t_2)$ with blocks with three fields, where two fields hold the subtrees and the additional tag field says whether B or C is used. Declare functions such that $getTree\,(putTree\ t) = t$ holds for all ABC-trees t that fit into the heap.

9.8 Structure Sharing and Physical Equality

An OCaml interpreter stores the values of constructor types it computes with in a heap using a block representation, and physical equality (Section 8.6) of values of constructor types is equality of heap addresses. If we use the function (Section 6.6)

```
let rec mtree n =
  if n < 1 then A
  else let t = mtree (n-1) in B(t,t)
```

to obtain the maximal AB-tree of depth n, we are guaranteed to obtain a heap representation with maximal structure sharing whose size $O(n)$. On the other hand, the function

```
let rec mtree' n =
  if n < 1 then A else B(mtree (n-1), mtree (n-1))
```

will construct a heap representation without structure sharing whose size is $O(2^n)$. Thus $mtree'$ is naive in that it takes exponential space and exponential time where linear space and linear time would suffice.

Exercise 9.8.1 Consider the block representation of AB-trees in the heap H.

a) Declare a function that writes the maximal AB-tree of depth n into the heap such that running time and space consumption in the heap are linear in the depth.

b) Declare a function that reads AB-trees from a heap preserving structure sharing between subtrees that are siblings (as in $B(t, t)$). On the heap representations of the trees obtained with the function from (a) your function should only use linear time in the depth of the tree.

c) Using physical equality, you can find out in constant time whether two AB-trees in OCaml have the same heap representation. Declare a function writing AB-trees into the heap such that structure sharing between sibling subtrees in OCaml is preserved. Test your function with $mtree'$.