Lecture Notes

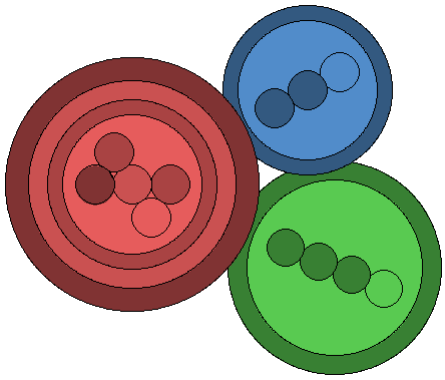
Functional Programming

(in OCaml)

8th Edition

Christian Sternagel and Harald Zankl

October 7, 2016



$$(\lambda mnfx.m\ f\ (n\ f\ x))\ (\lambda fx.f\ (f\ x))\ (\lambda fx.f\ (f\ (f\ x)))$$

```
(fun m n f x -> m f (n f x))
  (fun f x -> f (f x))
  (fun f x -> f (f (f x)))
```

Preface

These course notes accompany the functional programming course 703.024 in winter term 2016/2017 at the University of Innsbruck. They are based on the course notes of the functional programming course 703.024 from previous years, some small corrections have been performed.

Acknowledgments (by Christian Sternagel). Thanks to the strange anonymous referee character who had the main effort of reading the pre-versions of this document.

I am also grateful to all the other people that hunted down deficiencies in my English and errors in the presented code (which are—due to the fact that a functional language is used—rather rare). These are in alphabetical order: Simon Bailey, Christian Bitschnau, Andreas Grill, Daniel Luttinger, Friedrich Neurauter, Thomas Sternagel, and Sarah Winkler.

Acknowledgments (by Harald Zankl). I am grateful to Christian Sternagel for providing his course notes and happy that years after reading pre-versions of this document I can build on it. The graphic on the cover page is inspired by a visualization of the lambda calculus and has been generated using a tool by Viktor Massalõgin.

Contents

Functional programs are readable like a book.

Nao Hirokawa

1. Introduction

This introductory chapter consists of the following parts: a short *historical overview*, some *typographical conventions* to make reading easier, and an *overview* of the remaining chapters.

1.1. Historical Overview

Unlike most historical overviews this one starts today and continues back into the past. 2016
(As you already noticed, dates are indicated in the margin.)

1.1.1. The Origin of OCaml

The starting point is OCaml—an abbreviation for Objective Caml—which was first released in 1996. On the Caml homepage it is stated that: 1996

> Objective Caml was the first language (and is still the only language) that combines the full power of object-oriented programming with ML-style static typing and type inference.

Nowadays OCaml is the most popular variant of Caml that has been developed since 1985 at INRIA. The first implementation appeared in 1987. Originally Caml arose when 1985
the french Formel team became interested in ML (*metalanguage*) in 1980. Where does 1987
the name stem from? CAM was the *Categorical Abstract Machine* which could be seen 1980
as a 'compiler' for ML. Hence the name *categorical abstract machine language*. The first
ML compiler was built in 1974. ML itself was created in the 1970s by Robin Milner at 1974
the University of Edinburgh. Eventually ML developed into several dialects, the most 1970
common of which are now Objective Caml and Standard ML.

1.1.2. Underlying Theory

The main contributions to the underlying theory of functional programming are lambda calculus, combinatory logic, and term rewriting.

Term rewriting is a model of computation which traces its origins back to combinatory logic and lambda calculus, but also (in a separate development) to the study of *word problems* in universal algebra in a landmark paper by Donald Ervin Knuth and his student Peter Bendix, published in 1970. 1970

The lambda calculus—also λ-calculus—is a formal system designed to model computations of any kind, which is *Turing complete*. This means that in principle any algorithm could be described by solely using the λ-calculus. It was introduced by Alonzo Church in the 1930s. 1930

Combinatory logic is an equivalent theoretical foundation, and was further improved by Haskell Brooks Curry in the late 1920s but already invented by Moses Schönfinkel 1929
in 1924. It was originally developed to achieve a clearer approach to the foundations of 1924
mathematics.

1.2. Typographical Conventions

There are mainly two special formats used in this document: One for shell commands (like instructions on how to compile a source file) and the other for OCaml source code. Both are set in `typewriter font`. OCaml source code is `green` where keywords are **`bold`**.

OCaml source listings—representing the contents of a file—are separated from the floating text by leading and trailing horizontal lines.

For convenience everything that is 'clickable' (like page numbers in the index, or entries of the table of contents) is blue (of course you can click anywhere you like, but do not expect anything special unless you happened to click on some blue text).

1.3. Overview

In Chapter 2 (Lists) one of the most commonly used data structures of functional programming languages is introduced. The implementation details of many useful list functions are given and concepts like polymorphism, higher-order functions, and modules are introduced.

An alternative implementation of strings is discussed in Chapter 3. It is not used in the OCaml library, but—as it is purely functional—is nevertheless very interesting in its own right.

Chapter 4 (Trees) introduces another widely used data structure and gives a concrete example—namely the Huffman encoding—of its usage. Additionally some more information on the module system of OCaml is given.

The first theoretical part, Chapter 5 (λ-Calculus) gives an overview of part of the theory underlying functional programming. After giving a short but concise introduction, issues like recursion and the usage of λ-calculus in the implementation of functional languages are briefly discussed.

One of the big advantages of functional programs is that it is much simpler to mathematically prove properties of them than it is for imperative languages. In Chapter 6 (Reasoning About Functional Programs) a basic proof technique for proving properties of functional programs is presented: structural induction.

In Chapter 7 (Efficiency) some methods to improve the time and space complexity of typical functions are presented.

Divide and conquer techniques and dynamic programming are the topic of Chapter 11.

Then Chapter 8 introduces the concept of parser combinators. It is shown how to define functional parsers. The advantages are that no new language has to be learned as it is the case for parser generator tools like Lex and Yacc, and that parsers are just functions, i.e., normal values for functional programs.

Another theoretical part (Chapter 9) is concerned with types and methods to check and even compute (*infer* is more commonly used in this regard) types for given programs, concepts that are ubiquitous in most modern programming languages.

A way to compute with (potentially) infinite data structures is presented in Chapter 10. The key observation is that expressions should only be evaluated if they are really needed to compute the result.

Appendix A is intended as a short crash course in OCaml programming. Readers that do not know (or have to refresh their knowledge about) OCaml, should read this chapter first.

Appendix B gives some hints how to use `ocamlbuild`, a relatively new tool for automating the compilation of most OCaml projects with minimal user input.

2. Lists

One of the basic data structures in almost every functional language is the *list*—sometimes also referred to as *(finite) sequence*. A list is a collection containing arbitrary (but finitely) many elements of the same type. In OCaml the type of lists could be user-defined by

```
type 'a list = Nil | Cons of ('a * 'a list)
```

Then for example the list of all integers from 1 to 4 would be written as

```
Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
```

Since the list type is so useful it is already predefined in OCaml and the above list can be written more succinctly as

```
[1;2;3;4]
```

Unlike arrays (in programming languages like C, Java, etc.), lists do not support fast random access. Rather elements of a list can only be accessed by traversing the list starting at the first element (also called the *head* of the list). This means that accessing the last element of a list is strictly more costly than accessing the head of the same list (only if the list has at least two elements, of course). Thus, lists are predestined for all kinds of computation that are done sequentially. Following the 'head is trump' convention of lists, new elements can only be added at the front of a list ('added' is not quite the correct term since functional data types are never modified, indeed a new list is built containing all elements of the old one plus a new head in addition). The used constructor is '`::`'—pronounced "cons"[1]—where e `::` l is equivalent to `Cons(`e`,`l`)`. A value of the form e `::` l is also referred to as a *cons-cell*. E.g., to add `0` to the above list one would have to write

```
0::[1;2;3;4]
```

resulting in the new list

```
[0;1;2;3;4]
```

The empty list is denoted by '`[]`'—pronounced "nil"[2] (as you will have guessed the counterpart of the constructor '`Nil`' from above). Consequently every list can be constructed by combining its elements just using '`[]`' and '`::`'. The list `[0;1;2;3;4]` for instance can be written as

```
0::(1::(2::(3::(4::[]))))
```

which is equivalent to `0::1::2::3::4::[]` since '`::`' associates to the right. Indeed, internally every list is constructed in this way and the notation using '`[`' and '`]`' together with '`;`' is just *syntactic sugar*.[3] The structure of lists facilitates the use of *recursive functions*. In the examples so far, only integers (type `int` in OCaml) have been used as

[1] Because lists are *constructed* using this operator. The name *cons* derives from LISP, one of the first programming languages with functional spirit.

[2] A contraction of the Latin *nihil*, meaning nothing. If used as an adjective, one speaks about *null* lists, i.e., a list containing no (Latin *nullus*) elements.

[3] Landin invented this term to refer to abbreviations and conventions adopted by languages to make programming in the λ-calculus (see Chapter 5) more convenient. "A little bit of syntactic sugar helps you swallow the lambda calculus."

elements of lists. However, the type variable `'a` (usually pronounced "alpha" since very often Greek letters are used to denote type variables) indicates that there is some choice in the type of list elements (with the restriction that all elements of a given list are of the same type). Indeed the abstract type declaration from the beginning of the chapter gives rise to many concrete types, e.g., lists of characters, lists of integers, lists of lists of floats, etc. Such types are called *polymorphic*.[4] When fixing `'a` to some concrete type (like `int` above) an *instance* of the polymorphic type `'a list` is created. Attached to polymorphic types are polymorphic functions, i.e., functions that work on any instance of some polymorphic type (which, in the case of lists, is only possible if the function does not need any knowledge of the concrete type of a list element). Until now it has only been shown how to construct lists. This alone is not very useful. Hence it will be shown how to access elements of a list. Since accessing elements has to work on every type instance of `'a list` the so called *selectors* are examples of polymorphic functions.

2.1. Selectors

The two selectors for lists are `hd` ("head") and `tl` ("tail"). The *head* of a list is the first of its elements whereas its *tail* is the remaining list (i.e., everything except the first element). E.g., for the list `[0;1;2;3;4]` the head is `0` and the tail is `[1;2;3;4]`. For an empty list the head and the tail are both undefined.

These functions could be implemented as

```
let hd (x::_) = x
```

(i.e., if the given list consists of a cons-cell then return its first argument, in any other case the result is undefined) and

```
let tl (_::xs) = xs
```

(i.e., if the given list consists of a cons-cell then return its second argument, in any other case the result is undefined). The special variable '`_`' can be used whenever one does not need a name for the matching expression. These definitions meet the condition stated above that `hd` and `tl` are undefined on empty lists (since the patterns in the definitions of the selectors do not match an empty list).

A longer—but nevertheless preferable—implementation of the selectors would explicitly issue an error on empty lists. This can be done via the standard library function `failwith : string -> 'a` as follows:

```
let hd = function x::_ -> x
               | _     -> failwith "empty␣list"

let tl = function _::xs -> xs
               | _      -> failwith "empty␣list"
```

Using these definitions, compiler warnings like

```
Warning P: this pattern-matching is not exhaustive.
```

can be avoided.

In the rest of the lecture, computations of functional programs will be modeled by *rewriting* of expressions. Consider for instance the calculation of the result of `hd [1;2;3;4]`, which is handled as an atomic step:

$$\texttt{hd [1;2;3;4]} = \texttt{1}$$

[4] The type has many (from the Greek *poly*) forms (from the Greek *morphe*).

Since there is a certain direction in this *simplification step* (namely replacing a function call by the body of its definition), the equality sign ('=') is replaced by an arrow ('→') in the sequel. Then the above is written as

$$\texttt{hd [1;2;3;4]} \rightarrow \texttt{1}$$

2.2. Some Other Polymorphic List Functions

Here are some examples of polymorphic list functions that will be used in the rest of the document.

2.2.1. Append

A useful function on lists is `append : 'a list -> 'a list -> 'a list` which takes two lists (of same type) and returns a list consisting of the elements of the first followed by the elements of the second list. A possible implementation is

```
let rec append xs ys = if xs = [] then ys
                                  else hd xs::append (tl xs) ys
```

The function works as follows: if the list `xs` is empty then just return the list `ys`, otherwise take the head of `xs` and put it in front of the list resulting from calling `append` (recursively) on the tail of `xs` and the list `ys`. Consider for example the computation steps necessary to append `[3;4]` to `[1;2]`:

```
append [1;2] [3;4]
 → if [1;2] = [] then [3;4]
                 else hd [1;2]::append (tl [1;2]) [3;4]
 → if false then [3;4]
            else hd [1;2]::append (tl [1;2]) [3;4]
 → hd [1;2]::append (tl [1;2]) [3;4]
 → 1::append (tl [1;2]) [3;4]
 → 1::append [2] [3;4]
 → 1::if [2] = [] then [3;4]
                  else hd [2]::append (tl [2]) [3;4]
 → 1::if false then [3;4]
               else hd [2]::append (tl [2]) [3;4]
 → 1::hd [2]::append (tl [2]) [3;4]
 → 1::2::append (tl [2]) [3;4]

 → 1::2::append [] [3;4]
 → 1::2::if [] = [] then [3;4]
                    else hd []::append (tl []) [3;4]
 → 1::2::if true then [3;4]
                 else hd []::append (tl []) [3;4]
 → 1::2::[3;4] = [1;2;3;4]
```

Notice that the green equality sign ('=') is part of OCaml whereas '=' denotes mathematical equivalence. (We are sorry for those who read this in black on white, however, the two 'equalities' should be distinguishable by context.)

In the above *reduction sequence* it is assumed that in addition to the rewrite rules obtained from the function definitions of `hd`, `tl`, and `append`, there are the rules:

$$\texttt{[] = []} \rightarrow \texttt{true}$$

$$x \texttt{ :: } xs \texttt{ = []} \rightarrow \texttt{false}$$

$$\texttt{if false then } t \texttt{ else } e \rightarrow e$$

$$\texttt{if true then } t \texttt{ else } e \rightarrow t$$

Those are examples of built-in rewrite rules that stem from OCaml's implementation of equality and conditional branching.

In future examples reduction sequences as the one above will not be given in full detail, however, if more than one step is done at once, this will be indicated by using '$\rightarrow^+$' instead of '$\rightarrow$' (where '$\rightarrow^+$' denotes the transitive closure of the relation $\rightarrow$ and can be read as "one or more steps of '$\rightarrow$'").

Notice that an equivalent definition of `append` would have been '`@`', which is defined as

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

As you can see the usage of pattern matching obsoletes calls to the selectors `hd` and `tl`. It is mostly a matter of taste whether to use the selectors or pattern matching—notice however that `hd` and `tl` should only be used with special care, to avoid exceptions. Since appending lists is commonly used, we introduced the infix operator '`@`'. (Note: *infix notation* means that the operator is written between its arguments, as opposed to *prefix notation*, where the function is written first.)

2.2.2. Replicate

The function computing a list consisting of `n` copies of the given value `x` is called `replicate` and of type `int -> 'a -> 'a list`. The implementation can be done as follows:

```
let rec replicate n x =
 if n < 1 then [] else x::replicate (n-1) x
```

If `n` (the desired length of the resulting list) is `0` then the only list of length 0 is returned, the empty list. Otherwise `x` is put in front of a list of `n-1 x`'s, in order to obtain a total length of `n`. As an example consider the evaluation of `replicate 2 0`:

```
replicate 2 0
 →+ if 2 < 1 then [] else 0::replicate 1 0
 →+ 0::replicate 1 0
 →+ 0::if 1 < 1 then [] else 0::replicate 0 0
 →+ 0::0::replicate 0 0
 →+ 0::0::if 0 < 1 then [] else 0::replicate (-1) 0
 →+ 0::0::[] = [0;0]
```

2.2.3. Take, Drop, and Split At

Sometimes you need to split a list at a certain position obtaining the two resulting lists. This operation can be achieved by combining two functions. The first function is used to get an initial segment of a list up to a certain number of elements (`take : int -> 'a list -> 'a list`) and the second to get the rest of a list after dropping a certain number of elements (`drop`) resulting in the desired function `split_at : int -> 'a list -> ('a list * 'a list)`. The function `take` is implemented as follows

```
let rec take n xs = if n < 1 then [] else match xs with
 | []    -> []
 | x::xs -> x::take (n-1) xs
```

i.e., nothing is taken from nil, hence nil is returned. Similarly, if `0` elements have to be taken nil is returned. Otherwise the head of the list is added to the list resulting from taking `n-1` elements from its tail. The function `drop : int -> 'a list -> 'a list` can be defined similarly

```
let rec drop n xs = if n < 1 then xs else match xs with
 | []    -> []
 | _::xs -> drop (n-1) xs
```

i.e., it does not drop anything from nil, hence returning nil. If no element has to be dropped the whole list is returned. If there remain `n` elements to be dropped, `n-1` elements from the tail of the list are dropped (which is exactly one element shorter than the full list). Combining the above two functions results in

```
let split_at n xs = (take n xs,drop n xs)
```

where the resulting pair contains the first `n` elements of the given list in its first component and the rest of the list in its second component. In Chapter 7 we will study methods that allow to implement `split_at` slightly more efficient (the current implementation iterates twice over the list `xs`, for sufficiently large `n`).

2.3. Functions on Integer Lists

There are also functions that do only make sense if the elements of a list have a certain type. What follows are examples of such functions for lists of integers.

2.3.1. Range

The function `range : int -> int -> int list` computes—given integers `m` and `n`—the list starting at `m` and consisting of all consecutive integers until (not including) `n`. It can be defined by

```
let rec range m n = if m >= n then []
                               else m::range (m+1) n
```

If `m` is greater than or equal to `n` then there is no list containing elements inbetween and hence nil is returned. Otherwise `m` is put in front of the list resulting from constructing the list of integers between (including) `m + 1` and (not including) `n`.

You may want to verify that

```
range 3 6 →  [3;4;5]
```

2.3.2. Sum

The function `sum : int list -> int`, summing up all elements in a list of integers, can be defined by

```
let rec sum = function []    -> 0
                     | x::xs -> x + sum xs
```

This can be read as: if the list is empty then the sum of its elements is zero, otherwise it consists of a head and a tail and the sum can be computed by adding the head-element to the sum of the tail-elements. The constituent steps of the computation for calculating `sum [1;2;3;4]` are as follows

```
sum [1;2;3;4]
 →   1 + sum [2;3;4]
 →   1 + (2 + sum [3;4])
 →   1 + (2 + (3 + sum [4]))
 →   1 + (2 + (3 + (4 + sum [])))
 →   1 + (2 + (3 + (4 + 0)))
 →+  10
```

2.3.3. Prod

The function `prod : int list -> int`, multiplying all elements in a list of integers, can be defined by:

```
let rec prod = function []    -> 1
                      | x::xs -> x * prod xs
```

As can be seen `prod` is very similar to `sum`, the only differences being that '`*`' is used instead of '`+`' and '`1`' instead of '`0`'.

Remarks. The function call `sum(range 1 n)` corresponds to the mathematical notation

$$\sum_{i=1}^{n-1} i \tag{2.1}$$

and `prod(range 1 n)` to

$$\prod_{i=1}^{n-1} i$$

2.4. Higher-Order Functions

Much of the power of functional programming stems from the fact that functions are first class citizens, meaning that a function is just a value that can be returned from or given to another function. Consider for example the formula

$$\sum_{i=0}^{n-1} 2^i \tag{2.2}$$

computing the greatest number that can be encoded in binary using n bits, i.e., n ones. This formula looks very similar to (2.1) just that before adding, another function is applied to i (namely $f(x) = 2^x$). The function f can be defined as below:

```
let pow2 n = prod(replicate n 2)
```

Another function is needed that takes `pow2` as argument and applies it to a list.

2.4.1. Map

There is indeed a more general function commonly called `map`, which takes a function and a list and applies the function to every element of the given list, returning a new list containing the corresponding results. Its implementation is

```
let rec map f = function []    -> []
                       | x::xs -> f x::map f xs
```

and it has the type `('a -> 'b) -> 'a list -> 'b list`.

Calling `map` f `[`x_1`;` $\ldots$ `;`x_n`]` results in the new list `[`$f\ x_1$`;` $\ldots$ `;`$f\ x_n$`]`. Now (2.2) can be defined by

```
sum(map pow2 (range 0 n))
```

2.4.2. Fold

As already mentioned above, the two functions `sum` and `prod` are almost identical. Intuitively the first inserts a '`+`' whereas the second inserts a '`*`' between every two elements of a list, i.e.,

$$\texttt{sum}\ [x_1; \ldots; x_n] \stackrel{\text{def}}{=} x_1 \texttt{ + } (x_2 \texttt{ + } (\cdots \texttt{ + } (x_{n-1} \texttt{ + } (x_n \texttt{ + } 0)) \cdots))$$

and

$$\texttt{prod}\ [x_1; \ldots; x_n] \stackrel{\text{def}}{=} x_1 \texttt{ * } (x_2 \texttt{ * } (\cdots \texttt{ * } (x_{n-1} \texttt{ * } (x_n \texttt{ * } 1)) \cdots))$$

There is a corresponding higher-order function commonly known as *foldr* (denoting *fold right* for reasons that are immediate from its definition), defined by

$$\texttt{foldr}\ \oplus\ b\ [x_1; \ldots; x_n] \stackrel{\text{def}}{=} x_1 \oplus (x_2 \oplus (\cdots \oplus (x_{n-1} \oplus (x_n \oplus b)) \cdots))$$

where $\oplus$ is a binary function (i.e., taking two arguments) and b the base value (since it would be the result of `foldr` $\oplus$ b `[]`). In OCaml `foldr` can be implemented as

```
let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)
```

Indeed this one is a hard nut to crack but it is not necessary to understand it fully on first sight. To convince yourself that the above definition has the intended behavior, have a close look at the following reduction sequence:

```
foldr (+) 0 [1;2;3]
 →   (+) 1 (foldr (+) 0 [2;3]) = 1 + (foldr (+) 0 [2;3])
 →   1 + (2 + (foldr (+) 0 [3]))
 →   1 + (2 + (3 + (foldr (+) 0 [])))
 →   1 + (2 + (3 + 0))
 →+ 6
```

Notice the usage of '`(+)`' to be able to use '`+`' with prefix notation (see Appendix A).

Using `foldr` there are shorter implementations of `sum` and `prod`, namely:

```
let sum xs = Lst.foldr ( + ) 0 xs

let prod xs = Lst.foldr ( * ) 1 xs
```

Symmetrically to `foldr` (which starts to insert a binary function at the end of a list) there is a function `foldl` which does almost the same (only starting from the beginning of the list):

$$\texttt{foldl}\ \oplus\ b\ [x_1; \ldots; x_n] = ((\cdots((b \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_{n-1}) \oplus x_n$$

It can be implemented as follows:

```
let rec foldl f b = function []     -> b
                           | x::xs -> foldl f (f b x) xs
```

2.4.3. Filter

Sometimes it is useful to remove all elements from a list that do not satisfy a certain property. Therefore the function

```
let rec filter p = function
 | []    -> []
 | x::xs -> if p x then x::filter p xs else filter p xs
```

```
1 (* main : unit -> unit *)
2 let main() =
3  let n = read_int() in
4  let r =
5   IntLst.sum (Lst.map Int.pow2 (IntLst.range 0 n)) in
6  Printf.printf "%i\n" r
7 in main()
8 (*  *)
```

Listing 2.1: `Maxbin.ml`

can be used. For instance, to remove all odd numbers from the four element list `[1;2;3;4]` one could use

```
let even x = x mod 2 = 0 in
filter even [1;2;3;4]
```

resulting in the computation steps:

```
filter even [1;2;3;4]
 →   if even 1 then 1::filter even [2;3;4]
               else filter even [2;3;4]
 →+ filter even [2;3;4]
 →   if even 2 then 2::filter even [3;4]
               else filter even [3;4]
 →+ 2::filter even [3;4]
 →   2::if even 3 then 3::filter even [4]
                  else filter even [4]
 →+ 2::filter even [4]
 →   2::if even 4 then 4::filter even []
                  else filter even []
 →+ 2::4::filter even []
 →   2::4::[] = [2;4]
```

2.5. Introduction to Modules

In most functional languages it is courteous to structure program code using *modules*. A module can be understood as a collection of functions accessible via a common prefix (the module name). Since structuring programs into modules is a good idea we will outright start to do so. At the same time consider the following as an example of how to build your own modules.

So far some functions on lists have been defined (where one can separate functions on arbitrary lists and functions on integer lists) as well as one function (`pow2`) that works directly on integers. For the former the modules called `Lst` (since the name `List` is already occupied by the standard library) and `IntLst` are built, and for the latter a module called `Int`. Therefore all definitions are copied into the respective files `Lst.ml`, `IntLst.ml`, and `Int.ml`.

These modules can be used to write a program computing (2.2). The program can be seen in Listing 2.1. Here `readint : unit -> int` is a function from the module `Pervasives` and reads an integer value from standard input. In lines $4--5$, `r` is bound to the result of the computation. Finally the `printf` function of the `Printf` module is used to write an integer (`r`) followed by a newline onto the standard output channel.

It can be seen that there are dependencies between the different files. For instance `Int.ml` and `IntLst.ml` contain calls to functions from `Lst.ml` and of course `maxbin.ml`

depends on all other modules. To accommodate these dependencies one either links the files in the correct order, resulting in something similar to

```
> ocamlc -o maxbin Lst.ml IntLst.ml Int.ml maxbin.ml
```

or uses `ocamlbuild` described in Appendix B. (The tool `ocamlbuild` is used for automatic compilation of OCaml projects and is included in the OCaml distribution since version 3.10.)

2.6. Exercises

Exercise 2.1. Evaluate the function calls `take 2 [1;2;3]` and `take 3 [1;3]` by giving all computation steps on paper.

Hint: Evaluate the **`match`** statement immediately.

Exercise 2.2. Evaluate the function calls `range 0 3` and `range 3 0` by giving all computation steps on paper.

Exercise 2.3. Evaluate the following function calls by giving all computation steps on paper.

- `foldl (-) 0 [1;2;3]`
- `foldr (-) 0 [1;2;3]`

Hint: Evaluate the **`function`** statement immediately.

Exercise 2.4. Write a function `mem: 'a -> 'a list -> bool`, which tests if an element is contained in a list.

Exercise 2.5. Write a function `length : 'a list -> int`, which computes the number of elements in a list.

Exercise 2.6. Write a function `last : 'a list -> 'a` to return the last element of a list.

Exercise 2.7. Write a function `nth : 'a list -> int -> 'a` to get the n-th element of a list (where the head is indexed with 0).

Exercise 2.8. Write a function `reverse : 'a list -> 'a list` that reverses a list by (recursively) appending the head element at the tail. Then `reverse` $[x_1; \ldots; x_n]$ evaluates to $[x_n; \ldots; x_1]$.

Exercise 2.9. Write a function `concat : 'a list list -> 'a list` that takes a list of lists as input and computes the list resulting from appending those lists one after the other, e.g.,

$$\texttt{concat [[1;2];[3;4];[5;6]]} = \texttt{[1;2;3;4;5;6]}$$

Exercise 2.10. Write `foldl1 : ('a -> 'a -> 'a) -> 'a list -> 'a` as a variant of `foldl` that takes no base value and thus only works on nonempty lists (the function should fail if applied to an empty list).

$$\texttt{foldl1} \oplus [x_1; \ldots; x_n] = ((\cdots(x_1 \oplus x_2)\cdots) \oplus x_{n-1}) \oplus x_n$$

Exercise 2.11. Write `foldr1 : ('a -> 'a -> 'a) -> 'a list -> 'a` as a variant of `foldr` that takes no base value and thus only works on nonempty lists (the function should fail if applied to an empty list).

$$\texttt{foldr1} \oplus [x_1; \ldots; x_n] = x_1 \oplus (x_2 \oplus \cdots(x_{n-1} \oplus x_n)\cdots)$$

Exercise 2.12. Write a function `zip` which takes two lists and returns a list of pairs with the same length as the shorter of both as in the following example:

```
zip [1;2;3] [4;5;6;7;8] = [(1,4);(2,5);(3,6)]
```

Exercise 2.13. Write a function `zip_with` that, given a binary function and two lists, constructs a new list as in:

$$\texttt{zip\_with}\ f\ [x_1;\ldots;x_k]\ [y_1;\ldots;y_\ell] = [f\ x_1\ y_1;\ldots;f\ x_m\ y_m]$$

where m is the smaller of k and ℓ.

Exercise 2.14. Write a function `unzip` on lists (i.e., a list of pairs is 'unzipped' into a pair of lists) using one of the fold functions you already know. E.g.,

$$\texttt{unzip [('a',1);('b',2)]} = \texttt{(['a';'b'],[1;2])}$$

Further give the computation steps to compute `unzip [('a',1);('b',2)]`.

Exercise 2.15. Implement a function

```
sublists : int -> 'a list -> 'a list list
```

such that `sublists k xs` returns all `k`-element sublists of the list `xs`, ignoring the order of list elements. For example,

$$\begin{aligned}\texttt{sublists 3 [1;2]} &= \texttt{[]}\\ \texttt{sublists 3 [1;2;3]} &= \texttt{[[1;2;3]]}\\ \texttt{sublists 2 [1;2;3;4]} &= \texttt{[[1;2];[1;3];[1;4];[2;3];[2;4];[3;4]]}\end{aligned}$$

Exercise 2.16. Given a list $[x_1;\ldots;x_n]$, define a function that computes all its permutations.

$$\texttt{perm [1;2;3]} = \texttt{[[1;2;3];[1;3;2];[2;1;3];[2;3;1];[3;1;2];[3;2;1]]}$$

Hint: Suppose you already computed `perm` $[x_1;\ldots;x_{n-1}]$. How can you use this in order to compute `perm` $[x_1;\ldots;x_n]$?

Exercise 2.17. Given a list $[x_1;\ldots;x_n]$, define a function that computes its 'powerset', i.e.,

$$\texttt{powerset [1;2;3]} = \texttt{[[];[1];[2];[3];[1;2];[1;3];[2;3];[1;2;3]]}$$

Hint: The order of the subsets can be arbitrary; duplicates allowed.

Exercise 2.18. Define functions that compute the maximum/minimum of an arbitrary list.

Hint: Use the functions `max : 'a -> 'a -> 'a` and `min : 'a -> 'a -> 'a` from the module `Pervasives`.

Exercise 2.19. Define functions that compute the arithmetic/geometric mean of a list of `int`s.

Exercise 2.20. Write a recursive function `gsum : int -> float` that computes the following sum:

$$\texttt{gsum}(n) = 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}}$$

Exercise 2.21. Write a recursive function `euler : int -> float` that computes an approximation of Euler's number:

$$\texttt{euler}(n) = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(n-1)!}$$

Exercise 2.22. Define `map` in terms of `foldr` as follows

```
let map f xs = foldr (fun ... -> ...) [] xs
```

where you only have to provide the missing definition of the anonymous function.

Exercise 2.23. Write a console application that does the following:

a) Read a decimal integer from the console (use `read_int`).

b) Compute the binary representation of that integer as a list of binary digits, e.g., `[1;0;1;1]` for 11.

c) Output the binary representation on the console.

Exercise 2.24. Consider *Leibniz' formula*

$$4 \cdot \sum_{i=0}^{\infty} \frac{(-1)^i}{2 \cdot i + 1} = \pi \tag{2.3}$$

for computing π. Restrict to a finite case (i.e., changing ∞ to n). Write a program `leibniz.ml` (having the same structure as the program from Listing 2.1) approximating the number π given n—the number of steps to use. Instead of `"%i\n"` (for printing an integer) you will have to use `"%.10f\n"` (for printing a floating point number with 10 digits after the decimal point). You will also have to use the function `Pervasives.float_of_int` and have to define a function `fsum` computing the sum of all elements contained in a list of floats. Further notice that addition, division, exponentiation, and multiplication of floats are done via the functions

```
( +. ) : float -> float -> float
( /. ) : float -> float -> float
( ** ) : float -> float -> float
( *. ) : float -> float -> float
```

in OCaml.

Exercise 2.25. You have 100 doors (#1 to #100) in a row that are all initially closed. You make 100 passes by the doors. The first time through, you visit every door and toggle the door (if the door is closed, you open it; if it is open, you close it). The second time you only visit every 2nd door (door #2, #4, #6, ...). The third time, every 3rd door (door #3, #6, #9, ...), etc, until you only visit the 100th door.

a) Define a type `t` for doors (open/closed).

b) Write a function `toggle : t -> t` that toggles the door.

c) Generate a list of 100 closed doors.

d) Write a function `pass : int -> t Lst.t -> t Lst.t` that executes the i-th pass over the doors.

e) Write a function `passes : int -> t Lst.t -> t Lst.t` that executes n passes over the doors.

3. Strings

Strings are commonly understood as sequences of characters. The implementation of strings differs between functional languages. OCaml (dedicated to efficiency) implements strings as arrays of characters whereas Haskell for instance implements them as lists of characters. From an educational point of view (especially if the topic is *functional* programming) lists of characters are preferable.

Since OCaml strings are arrays (which are no functional data type at all) the module `Strng` (omitting vowels seems to come into vogue) will be implemented using character lists as representation of strings.

3.1. The Module Strng

The basic thing to state is the type of our new string implementation.

```
type t = char list
```

Instances of that type (i.e., `Strng.t`) will be called l-strings (where 'l' stands for *list*) in the following, whereas the term "string" will refer to the standard type `string` of OCaml (sometimes also "OCaml string"). Further, four functions on l-strings are provided:

(`of_string : string -> t`) builds l-strings from strings

(`of_int : int -> t`) transforms integers into l-strings

(`to_string : t -> string`) transforms l-strings into OCaml strings

(`print : t -> unit`) prints l-strings

The implementations of those functions are given for completeness, however, since they deal with non-functional data types, they will not be explained in detail.

```
let of_string s =
 let rec of_string i acc =
  if i < 0 then acc else of_string (i-1) (s.[i]::acc)
 in
 of_string (String.length s - 1) []

let of_int i = of_string(string_of_int i)

let to_string xs =
 let buffer = Buffer.create 128 in
 List.iter (Buffer.add_char buffer) xs;
 Buffer.contents buffer

let print s = Printf.printf "%s" (to_string s)
```

Whenever necessary, new functions will be added to the `Lst` module.

3.2. Example: Printing Strings

Strings in general are often used to print information on a terminal. Printing is done line by line. In order to provide nicely formatted output, vertical *and* horizontal alignment is important. To facilitate viewing terminal output as a picture, consider a module `Picture` with the following data type specifications

```
type width = int
type height = int
type t = (width * height * Strng.t list)
```

stating that a picture has a width (given as an integer), a height (also given as an integer), and some content (given as a list of l-strings representing the rows of the picture). Two basic operations on pictures are putting two pictures above each other (`above: t -> t -> t`) and putting them next to each other (`beside: t -> t -> t`). This can be implemented as

```
let above (w1,h1,p1) (w2,h2,p2) =
 if w1 = w2 then (w1,h1+h2,p1@p2)
            else failwith "different␣widths"

let beside (w1,h1,p1) (w2,h2,p2) =
 if h1 = h2 then (w1+w2,h1,Lst.zip_with (@) p1 p2)
            else failwith "different␣heights"
```

where the definition of `Lst.zip_with` is left as an exercise (cf. Exercise 2.13). It might be handy to recall the definition of '`@`' from page 5.

Before a picture can be printed on the terminal, it has to be transformed into an l-string. This is done via the function

```
let to_strng (_,_,p) = Lst.join ['\n'] p
```

where `Lst.join : 'a list -> 'a list list -> 'a list` takes a separating list d and a list of lists $[s_1; \ldots; s_n]$ and returns the list

$$\texttt{join}\ d\ [s_1; \ldots; s_n] = s_1\ \texttt{@}\ d\ \texttt{@}\ (s_2\ \texttt{@}\ d\ \texttt{@}\ \cdots (s_{n-1}\ \texttt{@}\ d\ \texttt{@}\ s_n)\cdots)$$

One possible implementation is

```
let join delim = foldr1(fun xs ys -> xs@delim@ys)
```

where `Lst.foldr1` is a specialized variant of `Lst.foldr` that only works on nonempty lists whose implementation is left as an exercise (cf. Exercise 2.11).

Given `above` and `beside`, a nonempty list of pictures can be stacked above each other, or spread beside one another. The implementations of those two functions are almost trivial:

```
let stack ps = Lst.foldr1 above ps

let spread ps = Lst.foldr1 beside ps
```

There are already some means to manipulate pictures but how to construct them? The simplest picture consists of just one pixel (i.e., character). The corresponding function is:

```
let pixel c = (1,1,[[c]])
```

Also (nonempty) l-strings should be representable as pictures. This is done via the function `row : Strng.t -> t`

```
let row s = spread(Lst.map pixel s)
```

i.e., first every character of the l-string is transformed into a pixel using the `pixel` function and then the resulting pictures are spread next one another via `spread` (where the resulting picture will have height 1 and the same width as the length of the given l-string). Using `row`, empty pictures (i.e., all pixels are `' '`) can be specified by:

```
let empty w h =
 let line = Lst.replicate w ' ' in
 let rows = Lst.replicate h line in
 stack(Lst.map row rows)
```

Two useful variants of `stack` and `spread` insert empty pictures—of given width and height, respectively—between two pictures before combining them:

```
let stack_with h ps = Lst.foldr1 (fun p q ->
 above (above p (empty (width q) h)) q) ps

let spread_with w ps = Lst.foldr1 (fun p q ->
 beside (beside p (empty w (height q))) q) ps
```

So these functions stack (spread) a list of pictures vertically (horizontally) with an inter-picture gap of height h (width w). Here the implementation of `width` and `height`—extracting the width and height of a given picture, respectively—is left as an exercise (cf. Exercise 3.1).

Given a list of lists of pictures, a large picture can be constructed by tiling them. There are again two variants: one that just tiles the given pictures and the other that additionally inserts empty pictures in between.

```
let tile pss = stack(Lst.map spread pss)

let tile_with w h pss =
 stack_with h (Lst.map (spread_with w) pss)
```

3.3. Chapter Notes

The examples of this chapter are a very slightly modified version of the examples from Chapter 5 in [3].

3.4. Exercises

Exercise 3.1. Write the function `width : Picture.t -> int` as well as the function `height : Picture.t -> int` returning the width and height of a given picture, respectively.

Exercise 3.2. Write functions for `Strng` to align l-strings within a box of given width. Three functions are needed:

```
left_justify : int -> t -> t
right_justify : int -> t -> t
center : int -> t -> t
```

In each case the integer argument denotes the width of the box. To illustrate the requirements consider the examples:

```
left_justify 5 ['H'; 'A'; 'L'] = ['H'; 'A'; 'L'; ' '; ' ']
right_justify 5 ['H'; 'A'; 'L'] = [' '; ' '; 'H'; 'A'; 'L']
center 5 ['H'; 'A'; 'L'] = [' '; 'H'; 'A'; 'L'; ' ']
```

Hint: You can ignore the case if a string is too long.

Exercise 3.3. Write a function `paragraph : int -> string -> Picture.t` that—given the desired text width w and some text t—constructs a picture of width w such that the content of t is split into as many lines of text as needed to fit into a paragraph of w columns.

Hint: You may break lines at any character.

Exercise 3.4. Modify `paragraph` from Exercise 3.3 in a way that it is possible to left-justify, right-justify, or center each line of text. Break lines at white space only (you may assume that no word is longer than the width of the paragraph. E.g., the function should output (approximately):

```
# Picture.paragraph2 12 Strng.right_justify "Functional programs \
  are readable like a book.";;
-␣:␣Picture.t␣=
␣␣Functional
programs␣are
␣␣␣␣readable
like␣a␣book.
```

Hint: Split the task into the following parts:

a) Separate words by white space

b) Combine words (such that they fit into a single line)

c) Align words (using Exercise 3.2)

d) Make the picture

Exercise 3.5. Implement a function `upper : t -> t` for the `Strng` module that converts all lowercase letters in the given l-string to uppercase ones, e.g.,

`upper ['a'; '1'; 'b'; '_'] = ['A'; '1'; 'B'; '_']`.

Hint: The functions `char_of_int` and `int_of_char` from the Ocaml standard library (module `Pervasives`) might be useful.

Exercise 3.6. Implement a function `substring: t -> t -> bool` for `Strng` that checks whether the first argument is a substring of the second one.

Exercise 3.7. Implement a function `to_strng : int -> int -> Strng.t` taking a number and a base as arguments and returning the l-string representation of the given number to the given base, e.g.,

`to_strng 4 10 = ['4']`
`to_strng 4 2 = ['1'; '0'; '0']`.

The function is only supposed to work for bases between 2 and 10.

Hint: Use the predefined functions `int_of_char` and `char_of_int` from the OCaml standard library (module `Pervasives`).

Exercise 3.8. Given the module `Calendar` containing the functions from Listing 3.1, write a calendar program. The function `first_days`—given a year—returns a list of day names as integers (0 = Sunday, 1 = Monday, ...), the result list does contain an integer for each month of the year, denoting with which day of the week the corresponding month starts (in the given year). E.g., `[0; 1; 2; 3; 4; 5; 6; 0; 1; 2; 3; 4]` would be a year, where January starts with a Monday, February with a Tuesday, and so on, and

```
type day = int
type month = int
type year = int
type date = (day * month * year)
type dayname = int

val first_days : year -> dayname list
val month_lengths : year -> day list
```

Listing 3.1: `Calendar.mli`

finally December starts with a Friday. The function `month_lengths`—given a year—returns a list containing the number of days each month has in that year. Use the `Picture` module to write a program that prints a calendar for a year (provided as argument) where each month is in the format:

```
       January 2007
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Therefore consider each month as consisting of three pictures: the title which should be centered (`January 2007` in the example), the heading (corresponding to the weekdays, i.e., `Su Mo Tu We Th Fr Sa` in the example), and the entries; which can be put above each other using the `Picture` functions. The output for one month thus is a picture of size 21×8 (since if the first of a month is a Friday or a Saturday 6 lines are needed for the entries).

Exercise 3.9. Caesar's cipher encodes a text by replacing each letter by another letter some fixed number (the *key*) of positions down the alphabet. E.g., encoding `HELLO` with a key of 2 yields `JGNNQ`.

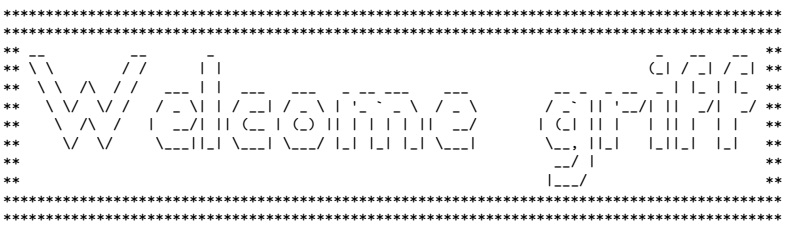
a) Write a function `pos_of_char : char -> int` that returns the position in the alphabet associated with an (uppercase) letter, as well as an inverse function `char_of_pos : int -> char`. E.g., `pos_of_char 'A'` yields `0` and the expression `char_of_pos 25` evaluates to `'Z'`.

   Hint: The predefined functions `char_of_int` and `int_of_char` may be useful.

b) Implement the function `encode : int -> Strng.t -> Strng.t` and the function `decode : int -> Strng.t -> Strng.t` which encode and decode an l-string using a given key.

c) Write a function `crack : Strng.t -> Strng.t list` that returns the list of all possible decodings for a given ciphertext. Use this function to decipher the text `RHNVKTVDXWMAXVHWX`.

Exercise 3.10. Consider the module `AsciiArt`. It contains only a single function. The function `char : char -> Picture.t`. For a given character `c`, `char c` yields an ASCII art letter representing `c` (this only works for the so-called *visible characters*, i.e., the characters 32 `' '` up to 126 `'~'`). Use this module together with the `Picture` module to implement the very useful function `welcome : string -> unit`. That prints a welcome banner for a given user name. E.g., `welcome "griff"` results in



Exercise 3.11. Write a function

a) `mirror_v: Picture.t -> Picture.t` which mirrors a picture vertically, e.g.,

```
***********    ***********
Hello World -> dlroW olleH
***********    ***********
```

b) `mirror_h: Picture.t -> Picture.t` which mirrors a picture horizontally,

c) `mirror: Picture.t -> Picture.t` which mirrors a picture horizontally and vertically.

Hint: The function `Lst.reverse` from Exercise 2.8 might be handy.

Exercise 3.12. Write a function `rotate: Picture.t -> Picturet` which rotates a picture 90° clockwise, e.g.,

```
***    %h*
hi  -> %i*
%%%    % *
```

Exercise 3.13. Write a function `frame: char -> Picture.t -> Picture.t` such that `frame c p` adds a frame (consisting of characters `c`) around picture `p`.

4. Trees

Another famous data structure in functional programming is the tree. A tree consists of some set of nodes plus a relation between those nodes. If a tree is not empty, there is exactly one node (the *root* of the tree) without an ancestor—also called parent—and every other node has exactly one ancestor. That implies that all nodes have some (meaning none or one or two or etc.) successors—also called children. A node without successors is called a *leaf*. A tree could consist of the following data type declaration

```
type 'a tree = Empty | Node of ('a * 'a tree list)
```

in OCaml. Hence a tree is either empty or consists of at least one node containing a value (`'a`) and a list of children (`'a tree list`) that could of course be empty as well. Examples of trees are `Empty`, `Node(1,[])`, `Node(1,[Empty])`, `Node(1,[Empty;Empty])`, `Node(1,[Node(2,[]);Node(3,[])])`, which show that trees do not have a unique representation. Instead of improving the type `'a tree` above we focus on a special kind of trees in the sequel.

4.1. Binary Trees

A common restriction on trees is that the number of child nodes is set to a certain maximum, say n. In case $n = 2$, the resulting trees are called *binary trees* (note that setting $n = 1$ would result in a list). In OCaml a type for binary trees can be defined by

```
type 'a btree = Empty | Node of ('a btree * 'a * 'a btree)
```

stating that a binary tree is either empty or it consists of at least one node having a left child (again a binary tree), containing some value (of type `'a`), and having a right child (also a binary tree). Then the empty tree is represented by `Empty`, the tree consisting of exactly one node having value 0 is represented by `Node(Empty,0,Empty)`, and so on.

4.1.1. The Module BinTree

In the following a module for binary trees is developed. Its name is `BinTree`. For convenience the type from above is given a shorter name (since from outside of the module it is prefixed by the module name anyway).

```
type 'a t = Empty | Node of ('a t * 'a * 'a t)
```

The *size* of a binary tree is the number of its nodes and can be defined by

```
let rec size = function
  | Empty        -> 0
  | Node(l,_,r) -> size l + size r + 1
```

stating that an empty tree has size 0 and the size of any other tree is the size of its left subtree plus the size of its right subtree plus 1.

The *height* of a binary tree is the length of the longest path from its root to some leaf. The implementation uses the `max : 'a -> 'a -> 'a` function of the module `Pervasives` that returns the greater of two given values.

```
let rec height = function
  | Empty        -> 0
  | Node(l,_,r) -> max (height l) (height r) + 1
```

You may already have noticed that writing down an instance of a binary tree is very annoying. Therefor a function to construct a binary tree from a given list would be convenient. A straightforward implementation would be

```
let rec of_list = function []    -> Empty
                         | x::xs -> Node(Empty,x,of_list xs)
```

which will result in binary trees where the left child is always `Empty` and hence having exactly the same structure as the given list but just wasting more memory. This is of course not desirable. It can be seen that when processing a list to build a tree, there is always a choice in which subtree to put an element of the list. An idea of a *fair* construction is to put the first half of the list into the left subtree and the second half into the right subtree. This can be realized as follows:

```
let rec make = function
 | [] -> Empty
 | xs ->
  let m = Lst.length xs / 2 in
  let (ys,zs) = Lst.split_at m xs in
  Node (make ys,Lst.hd zs,make(Lst.tl zs))
```

For an empty list an empty tree is returned. If the list has at least one element then `m` is set to the half of the number of elements of the list, the list is split at index `m` into two parts (you should be able to convince yourself that the second part has at least one element). The first part is used to build the left subtree whereas the tail of the second part is used to build the right subtree. The head of the second part is set as the value of the current node.

Another possibility is to insert each element of a list in a tree depending on whether it is smaller than or equal to the value of the root node or greater than the same value. Therefor consider the function `insert`:

```
let rec insert c v = function
 | Empty       -> Node(Empty,v,Empty)
 | Node(l,w,r) -> if c v w < 1 then Node(insert c v l,w,r)
                               else Node(l,w,insert c v r)
```

which can be used to build an ordered binary tree (also known as binary search tree, since searching elements is very fast in such trees). The argument `c` is a comparison function where the OCaml convention is that such functions have a type that is an instance of `'a -> 'a -> int` and the result of `c x y` denotes that $x < y$ if it is smaller than 0, $x = y$ if it is equal to 0, and $x > y$ if it is greater than 0.

To build a binary search tree the function `search_tree` is used

```
let search_tree c = Lst.foldl (fun t v -> insert c v t) Empty
```

The opposite of building a tree out of a list is to transform a tree into a list. One way to do that is the function `flatten`:

```
let rec flatten = function
 | Empty       -> []
 | Node(l,v,r) -> (flatten l)@(v::flatten r)
```

which processes the nodes of the given tree in a leftmost bottommost fashion. By combining the functions `search_tree` and `flatten` an (admittedly suboptimal) sorting algorithm for lists can be obtained.

```
let sort c xs = BinTree.flatten(BinTree.search_tree c xs)
```

4.2. A Little Bit More on Modules

In OCaml for every implementation file (i.e., ending in `.ml`) you may specify an interface file (ending in `.mli`) by the same name. This file just contains the signature of the module defined. This signature consists of the names of all types and all functions that should be visible outside of the corresponding `.ml` file. Furthermore for each function listed in the signature also the type must be specified. Refer to Listing 4.1 to see the interface file for the module `Lst`. (Note that some of the functions have not been implemented yet.)

```
(* W01 *)
(* W02 *)
type 'a t = 'a list

val hd : 'a t -> 'a
val tl : 'a t -> 'a t
val (@): 'a t -> 'a t -> 'a t
val replicate : int -> 'a -> 'a t
val take : int -> 'a t -> 'a t
val drop : int -> 'a t -> 'a t
val split_at : int -> 'a t -> 'a t * 'a t
val map : ('a -> 'b) -> 'a t -> 'b t
val foldr : ('a -> 'b -> 'b) -> 'b -> 'a t -> 'b
val foldl : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
val filter : ('a -> bool) -> 'a t -> 'a t
(* E02 *)
val mem : 'a -> 'a t -> bool
val length : 'a t -> int
val last : 'a t -> 'a
val nth : 'a t -> int -> 'a
val rev_append : 'a t -> 'a t -> 'a t
val concat : 'a t t -> 'a t
val foldl1 : ('a -> 'a -> 'a) -> 'a t -> 'a
val foldr1 : ('a -> 'a -> 'a) -> 'a t -> 'a
val zip : 'a t -> 'b t -> ('a * 'b) t
val zip_with : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
(* W03 *)
val join : 'a t -> 'a t t -> 'a t
```

Listing 4.1: `Lst.mli`

4.3. Example: Huffman Trees

An example where binary trees are useful is the Huffman coding (named after its inventor, David Huffman) which is used to compress files. When using `ASCII` encoding, every single letter in a text file needs exactly 8 bits. Hence for example the text "text" needs $(4 \cdot 8 =)$ 32 bits in `ASCII`. The idea to compress the needed number of bits is now to assign shorter bit sequences to symbols that occur very often in some given data. Consider for example the assignment

$$\begin{array}{lcl} t & \mapsto & 0 \\ e & \mapsto & 10 \\ x & \mapsto & 11 \end{array}$$

with which the text "text" would only need 6 bits to store. There are two important things to consider. Firstly the frequency of symbols in the given text has to be analyzed in order to be able to assign the shortest codewords to the most frequent symbols. Secondly every assigned codeword has to be unique, i.e., it has to be possible to reconstruct the original text given a compressed bit sequence and a code table (assigning codewords to characters). For instance when scanning the codeword 010110 from left to right (given the above code table) then there is no other possibility than "text" for the original data. The nice property used here is that no codeword is a prefix of another codeword (i.e., there are no two codes c_1 and c_2 such that $c_1 = c_2 b$ for some nonempty bit sequence b) which would for example be the case for the code table:

$$\begin{array}{lcl} t & \mapsto & 0 \\ e & \mapsto & 10 \\ x & \mapsto & 1 \end{array}$$

The result of compressing "text" would be 01010. But this bit sequence cannot be uniquely decoded since also "txtxt" or "tee" have the same encoding.

In order to use the Huffman coding the first thing to do is to analyze the frequency of all symbols in the input (also called the sample).

4.3.1. Analyzing the Sample

Counting Occurrences of Symbols

The idea for counting the number of occurrences of symbols in the sample is to first sort the input (hence equal symbols are one after another), then count the lengths of segments of equal symbols, and finally return a list of symbol-weight pairs (sorted by weights) where the weight of a symbol is the number of its occurrences in the sample.

Additional Functions for Lst

To count the number of occurrences of every symbol in the sample, some additional list functions are convenient. Namely

```
let concat xs = foldr (@) [] xs

let rec take_while p = function
  | []    -> []
  | x::xs -> if p x then x :: take_while p xs else []

let rec drop_while p = function
  | []            -> []
  | x::xs as list -> if p x then drop_while p xs else list

let span p xs = (take_while p xs, drop_while p xs)

let rec until p f x = if p x then x else until p f (f x)
```

where `take_while p xs` results in the longest prefix of `xs` such that all elements satisfy `p`, `drop_while p xs` results in the list obtained from `xs` by removing elements at the front until `p` is no longer satisfied, and `span` is a combination of both. The function `rev` returns the given list in reversed order, `until` repeatedly applies a given function until some condition becomes true, and `concat` takes a list of lists and appends them one after the other.

Counting the lengths of segments consisting of equal symbols is done via the function `collate`:

```
let rec collate = function
 | []          -> []
 | w::ws as xs ->
  let (ys,zs) = Lst.span ((=)w) xs in
  (Lst.length ys,w) :: collate zs
```

resulting in a list of pairs where the second component is a symbol and the first one the number of occurrences of this symbol in the sample. The last part for the sample analysis is the function `sample`

```
let sample xs = sort compare (collate(sort compare xs))
```

that uses the function `Pervasives.compare : 'a -> 'a -> int` (which is the default compare function of OCaml) to return a symbol-frequency list (ordered by increasing frequency) of the sample.

4.3.2. Building the Huffman Tree

In a Huffman tree leaves and all other nodes carry different kinds of information. The value of a leaf is a certain character plus the weight for this character whereas each non-leaf node only needs to store the sum of the weights of its two subtrees. To model that, the type

```
type node = (int * char option)
```

is used where `option` is a standard type constructor of OCaml and is defined by

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

Thus leaf nodes will have values of the form $(w, \texttt{Some}\ c)$ and non-leaf nodes values of the form $(w, \texttt{None})$. The type of Huffman trees is therefore defined by

```
type t = node btree
```

In outline, the procedure for building a Huffman tree is first to convert the given frequency list into a list of trees, and then repeatedly to combine two trees with lightest weights until just one tree remains. Combining two trees is done by the function `combine` and the two helper functions `weight` and `insert`

```
let combine = function
 | xt::yt::xts -> let w = weight xt + weight yt in
  insert (Node(xt,(w,None),yt)) xts
 | _           -> failwith "length␣has␣to␣be␣greater␣than␣1"
let is_singleton x = Lst.length x = 1

let weight = function
 | Node(_,(w,_),_) -> w
 | _               -> failwith "empty␣tree"

let insert vt wts =
 let (xts,yts) =
  Lst.span (fun x -> weight x <= weight vt) wts in
 xts@(vt::yts)
```

The Huffman tree is finally built using the function `tree`

```
let tree xs =
 let ts = Lst.map mknode (sample xs) in
 Lst.hd (Lst.until is_singleton combine ts)


let mknode (w,c) = Node(Empty,(w,Some c),Empty)
```

4.3.3. Encoding and Decoding

In a Huffman tree it is easy (and fast) to find a character given a bit sequence, however, for compressing (i.e., encoding) data, the opposite operation is needed, namely finding a bit sequence for a given character. For this purpose a *code table* is generated out of the Huffman tree. (An example of such a code table was

$$\begin{array}{lcl} t & \mapsto & 0 \\ e & \mapsto & 10 \\ x & \mapsto & 11 \end{array}$$

from the beginning of this section.) Given the following types

```
type bits = int list

type table = (char * bits)Lst.t
```

the above code table amounts to `[('t',[0]);('e',[1;0]);('x',[1;1])]` in Ocaml. The function `table` takes a Huffman tree as input and generates a code table. The helper function performs a top to bottom construction, i.e., it builds the code when descending in the tree and when arriving at a leaf the code (for a single character) is already computed. (In Exercise 4.4 you are asked to compute the table bottom to top.)

```
let table t =
 let rec tab code = function
  | Node(Empty,(_,Some v),Empty) -> [(v,code)]
  | Node(l,_,r) -> tab (code@[0]) l @ tab (code@[1]) r
  | _ -> failwith "the␣Huffman␣tree␣is␣empty"
 in tab [] t
```

To consult the table the function `lookup` is used.

```
let rec lookup tab c = match tab with
 | ((v,code)::tab) -> if v = c then code else lookup tab c
 | _               -> failwith "not␣found"
```

The process of transforming a given text (an l-string) into a sequence of bits (`int list`) is called *encoding*. Of course the encoding relies on a code table which has to be generated from the Huffman tree before. The function `encode: t -> Strng.t -> bits` is used for this purpose. first computes the code table (based on the tree it gets as input) and then encodes the text based on the code table.

```
let encode t text =
 let tab = table t in
 Lst.concat(Lst.map (lookup tab) text)
```

Given some compressed data (bit sequences are encoded as lists of integers where, e.g., 010110 would be encoded by `[0;1;0;1;1;0]`) and a Huffman tree, *decoding* is the process of reconstructing the original (uncompressed) text. The function `decode` does the job and has type `t -> bits -> Strng.t`.

```
let rec decode_char = function
 | (Node(Empty,(_,Some c),Empty),cs) -> (c,cs)
 | (Node(xt,_,_),0::cs)              -> decode_char (xt,cs)
 | (Node(_,_,xt),1::cs)              -> decode_char (xt,cs)
 | _                                 -> failwith "empty␣tree"

let rec decode t = function
 | [] -> []
 | xs -> let (c,xs) = decode_char (t,xs) in c::decode t xs
```

4.4. Chapter Notes

The Huffman example is a modified version of the one for Haskell in [3] (Chapter 6).

4.5. Exercises

Exercise 4.1. Compute (on paper) the Huffman tree for the text:

```
This is a useless message.
```

How many bits does the compressed text need (ignoring the tree itself).

Exercise 4.2. Compute (on paper) the Huffman encoding for the text:

```
DON'T PANIC
```

Exercise 4.3. Show that for a given text the Huffman tree is not unique. Demonstrate the effect on the text `abcdcd`. Does this affect the number of bits needed to encode a message?

Hint: How does an alternative implementation of `insert` affect the resulting tree?

Exercise 4.4. Consider the module `Huffman`. The function `table` (see Section 4.3.3) performs a top to bottom construction of the code table, i.e., it builds the code when descending in the tree and when arriving at a leaf the code (for a single character) is already computed.

A bottom to top function first descends in the tree and then generates the codes from right to left by adding `0`/`1` in the code table for the left/right subtree at the front of the codes.

a) Extend the module `Huffman` by a function `table2 : t -> table`, which generates the code table bottom to top.

b) Which version is more efficient?

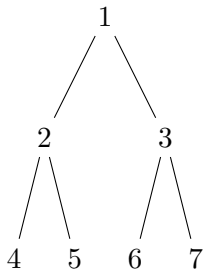
Exercise 4.5. Consider `make2 : 'a list -> 'a btree`

```
let rec make2 = function
  | [] -> Empty
  | x::xs ->
   let m = Lst.length xs / 2 in
   let (ys,zs) = Lst.split_at m xs in
   Node (make2 ys, x, make2 zs)
```

as an alternative implementation of the function `make` from page 21. Compare the result of `make [1;2;3;4;5;6]` and `make2 [1;2;3;4;5;6]`. Which function is more robust, i.e., depends less on the implementation/result of `Lst.split_at m xs`?

Exercise 4.6. Follow the computation of `sample ['h';'e';'l';'l';'o']` by evaluating (on paper) the results of all function calls starting at `sample ['h';'e';'l';'l';'o']`.

Exercise 4.7. Implement in-order, pre-order and post-order tree traversal for binary trees and run your functions on the example tree below. To be more precise, write functions that take a binary tree as sole argument and return a list that contains all tree nodes ordered according to the corresponding method of tree traversal. E.g., for post-order the tree



would result in the list `[4;5;2;6;7;3;1]`.

- In-order traversal visits the left subtree first, then the root, and finally the right subtree.
- Pre-order traversal visits the root first, then the left subtree, and finally the right subtree.
- Post-order traversal visits the left subtree first, then the right subtree, and finally the root.

Exercise 4.8. Implement a predicate

```
is_sorted : 'a tree -> ('a -> 'a -> int) -> bool
```

such that `is_sorted t c` returns **`true`** if the given tree `t` is sorted with respect to the comparison function `c`.

Exercise 4.9. Implement deletion in binary search trees.

```
delete : 'a btree -> 'a -> 'a btree
```

Test the correctness of your function by examples that cover all possible cases. Does your function delete one or all occurrences of an element? Why is above question irrelevant for binary *search* trees?

Exercise 4.10. Define `successors : 'a tree -> 'a -> 'a list` that computes all successors of a given element in a tree. The function should fail with an exception in case the element is not contained in the tree.

Exercise 4.11. Consider the type

```
type 'a tree = Empty | Node of ('a * 'a tree list)
```

Implement depth-first-search and breadth-first-search for trees:

```
dfs : 'a tree -> 'a -> bool
bfs : 'a tree -> 'a -> bool
```

The functions should return true if and only if the tree contains the sought element.

Exercise 4.12. Use search trees to implement the module `St` for finite sets where the signature is given by

```
type 'a t
val diff : 'a t -> 'a t -> 'a t
val empty : 'a t
val insert : 'a -> 'a t -> 'a t
val is_empty : 'a t -> bool
val mem : 'a -> 'a t -> bool
val of_list : 'a list -> 'a t
val singleton : 'a -> 'a t
val to_list : 'a t -> 'a list
val union : 'a t -> 'a t -> 'a t
```

i.e., internally the type `'a t` is `'a btree` as defined above but that fact is hidden from the user. The operations have following specifications (where S and T are sets and s and t are elements):

$$\begin{aligned} \texttt{diff}\ S\ T &= S \setminus T \\ \texttt{empty} &= \varnothing \\ \texttt{insert}\ s\ S &= \{s\} \cup S \\ \texttt{mem}\ s\ S &= s \in S \\ \texttt{singleton}\ s &= \{s\} \\ \texttt{union}\ S\ T &= S \cup T \end{aligned}$$

Exercise 4.13. Consider a representation of rational numbers where every number is given by some integer part plus some fraction. E.g.,

$$2\frac{4\frac{1}{2}}{5\frac{3}{4}}\,1\frac{6\frac{5}{6}}{7\frac{7}{8}}.$$

In principle this is a (non-empty) binary tree having integers as nodes (just rotated 90 degrees counterclockwise). For this representation, give a function `reduce` that reduces every given rational to its representation using at most one minimal (i.e., no further reduction is possible) fraction. Above example would result in $1\frac{12096}{16813}$.

Hint: Split the task into three functions:

a) `gcd : int -> int -> int` computing the greatest common divisor of two integers,

b) `reduce' : int t -> int * int` reducing a binary tree of integers to a pair (representing the numerator and the denominator of the fraction resulting from multiplying out the complex representation recursively),

c) and finally `reduce : int t -> int t` which combines `reduce'` and `gcd` (and some easy mathematics) to get the final result.

Exercise 4.14. Write a function `print : ('a -> Strng.t) -> 'a t -> unit` for the `BinTree` module that given a function converting elements of some type to l-strings and a binary tree, prints the tree. E.g., the tree of Exercise 4.7 should be printed as

```
1
+- 2
|  +- 4
|  |
|  +- 5
|
+- 3
   +- 6
   |
   +- 7
```

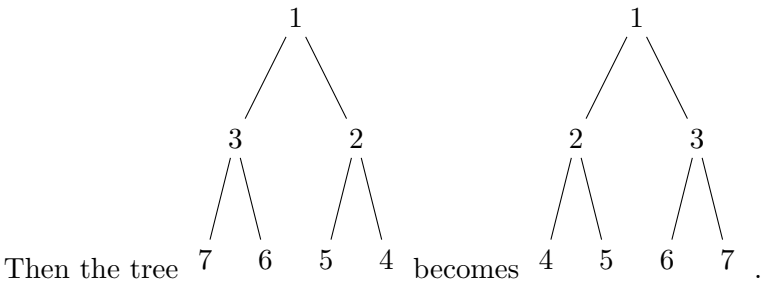
If you cannot achieve this in a reasonable amount of time, also

```
1
␣2
```

```
␣␣4
␣␣5
␣3
␣␣6
␣␣7
```

as output is acceptable (where ␣ denotes a space).

Exercise 4.15. Extend the module `BinTree` by a function `mirror : 'a t -> 'a t` that mirrors a tree.



.

A little bit of syntactic sugar helps you swallow the lambda calculus.

Peter Landin

5. λ-Calculus

Already Leibniz wanted to *create a universal language in which all possible problems can be stated*. Outcomes of such efforts you may already know are for instance Turing machines and register machines. From there also the expression *Turing completeness* stems. A computational model is said to be Turing complete if it can compute all "effectively" computable functions. In this chapter one of those Turing complete formal frameworks is introduced: the λ-calculus (speak "lambda calculus").

5.1. Syntax

The basic building blocks of the λ-calculus are λ-terms (or λ-expressions) where the possible shapes of a λ-term t—given a set of variables $\mathcal{V}$—are defined by the following BNF grammar:

$$t ::= x \mid (\lambda x.t) \mid (t\ t)$$

Here x is a variable from $\mathcal{V}$, $(\lambda x.t)$ is a *(lambda) abstraction* (somehow equivalent to a function definition like $f(x) = t$), and $(t\ t)$ is the *(function) application* of the left λ-term to the right one. The set of all λ-terms that can be built over some set of variables $\mathcal{V}$, is denoted by $\mathcal{T}(\mathcal{V})$. The OCaml equivalent to abstraction are anonymous functions. The term $(\lambda x.x)$ for example is equivalent (modulo typing) to the function `(fun x -> x)`.

Example 5.1. Some examples of well-formed λ-terms are:

$$x,$$
$$(\lambda x.x),$$
$$(\lambda x.(\lambda y.(x\ y))),$$
$$(((\lambda x.x)\ (\lambda x.x))\ (\lambda x.x)).$$

In order to save parentheses the conventions that outermost parentheses are dropped, that application binds tighter than abstraction, and that applications associate to the left are used. Then the above can be written as

$$x,$$
$$\lambda x.x,$$
$$\lambda x.(\lambda y.x\ y),$$
$$(\lambda x.x)\ (\lambda x.x)\ (\lambda x.x).$$

Furthermore nested abstractions associate to the right and in order to save λs, variables are grouped together, e.g., the rather longish term

$$(\lambda x.(\lambda y.(\lambda z.((x\ y)\ z))))$$

is written

$$\lambda xyz.x\ y\ z$$

using all of the above conventions. (In OCaml `fun x y z -> x y z` can be written instead of `(fun x -> (fun y -> (fun z -> ((x y) z))))`.)

5.1.1. Subterms

The simplest λ-term is a variable. All other λ-terms are built using smaller λ-terms.

Example 5.2. The term $\lambda x.x\ x$ consists of an abstraction on the smaller term $x\ x$, whereas this smaller term consists of the application of the term x to the term x.

In the above example, the terms x and $x\ x$ are called *(proper) subterms* of the term $\lambda x.x\ x$. The set $\mathcal{S}\mathsf{ub}(t)$ of subterms of a λ-term t is defined by

$$\mathcal{S}\mathsf{ub}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \{t\} \cup \mathcal{S}\mathsf{ub}(u) & \text{if } t = \lambda x.u \\ \{t\} \cup \mathcal{S}\mathsf{ub}(u) \cup \mathcal{S}\mathsf{ub}(v) & \text{if } t = u\ v \end{cases}$$

Notice that this also includes the term t itself. A term s is called a *proper* subterm of a term t, if $s \in \mathcal{S}\mathsf{ub}(t)$ and additionally $s \neq t$.

5.1.2. Free and Bound Variables

The set $\mathcal{V}\mathsf{ar}(t)$ of *variables* occurring in a λ-term t is defined by

$$\mathcal{V}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \{x\} \cup \mathcal{V}\mathsf{ar}(u) & \text{if } t = \lambda x.u \\ \mathcal{V}\mathsf{ar}(u) \cup \mathcal{V}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

The set $\mathcal{FV}\mathsf{ar}(t)$ of *free variables* of a term t consists of all variables that occur outside of a lambda abstraction.

$$\mathcal{FV}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \mathcal{FV}\mathsf{ar}(u) \setminus \{x\} & \text{if } t = \lambda x.u \\ \mathcal{FV}\mathsf{ar}(u) \cup \mathcal{FV}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

The set $\mathcal{BV}\mathsf{ar}(t)$ of *bound variables* in a term t is defined by

$$\mathcal{BV}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \varnothing & \text{if } t \text{ is a variable} \\ \{x\} \cup \mathcal{BV}\mathsf{ar}(u) & \text{if } t = \lambda x.u \\ \mathcal{BV}\mathsf{ar}(u) \cup \mathcal{BV}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

Definition 5.1 (Closed terms). A λ-term t without any free variables (i.e., $\mathcal{FV}\mathsf{ar}(t) = \varnothing$) is called *closed*.

5.2. Evaluation of Lambda Expressions

Until now you know how the syntax of the λ-calculus looks like, but it only starts to get interesting after knowing how to do computations using such syntactic constructs. The surprising fact is that the λ-calculus does only need one rule to receive its full computational power. The name of the rule is β.

5.2.1. Substitutions

Before the β-rule is given, something about *substitutions* has to be said. For the purpose of β-reduction (see the next section), a substitution is a mapping from a variable to a λ-term. We use the notation $\{x/t\}$ to denote the substitution replacing the variable x by the term t.

The application of a substitution $\{x/s\}$ to a λ-term t (written as $t\{x/s\}$) is defined by

$$t\{x/s\} \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \text{ and } x \neq y \\ (u\{x/s\})\ (v\{x/s\}) & \text{if } t = u\ v \\ t & \text{if } t = \lambda x.u \\ \lambda y.u\{x/s\} & \text{if } t = \lambda y.u,\ y \neq x, \text{ and } y \notin \mathcal{FV}\mathsf{ar}(s) \\ \lambda z.u\{y/z\}\{x/s\} & \text{if } t = \lambda y.u,\ y \neq x, \text{ and } y \in \mathcal{FV}\mathsf{ar}(s) \end{cases}$$

Note that the variable z in the last case is assumed to be *fresh*, i.e., it is different from all the variables occurring in u and s, and also unequal to x. Due to case four, bound variables are not substituted. To understand the last two cases, note what would happen if it was allowed to apply a substitution $\{x/s\}$ directly to a term $t = \lambda y.u$ with $y \in \mathcal{FV}\mathsf{ar}(s)$. This would for example yield $(\lambda x.y)\{y/x\} = \lambda x.x$ and $(\lambda z.y)\{y/x\} = \lambda z.x$. Where $\lambda x.y$ and $\lambda z.y$ provide the same results for the same inputs, but the two λ-terms after the substitution do not behave identical on identical input. (This problem is sometimes referred to as *variable capture*.)

5.2.2. The β-Rule

Computations within the λ-calculus are done by applying the β-rule stepwise (which is called β-reduction). A special kind of terms are contexts. A context C is built according to the following grammar

$$C ::= \Box \mid \lambda x.C \mid t\ C \mid C\ t$$

where $x \in \mathcal{V}$, $t \in \mathcal{T}(\mathcal{V})$, and $\Box \notin \mathcal{V}$ is a special symbol called *hole*. The set of all contexts built over some set of variables $\mathcal{V}$ is denoted by $\mathcal{C}(\mathcal{V})$. With $C[s]$ we denote the replacement of $\Box$ (in C) by the term s. Note that since every context contains exactly one hole (proving that is left as an exercise, cf. Exercise 5.6) the result of this operation is a term.

The β-rule is defined by

$$s \to_\beta t$$

if there is a subterm u of s (i.e., $u \in \mathcal{S}\mathsf{ub}(s)$) that is of the form $u = (\lambda x.v)\ w$. Then t is obtained from s by replacing u with $v\{x/w\}$. It is said that s *β-reduces* to t in one step. An alternative way to write this down would be: If there exist a context C and terms s, v, and w, such that

$$s = C[(\lambda x.v)\ w]$$

(which is equivalent to stating that $(\lambda x.v)\ w$ is a subterm of s) then

$$s \to_\beta C[v\{x/w\}]$$

Let s and t be λ-terms. If s reduces to t in a number of β-steps then we denote this by $s \to_\beta^* t$.

Consider for example the reduction step

$$(\lambda x.x)\ (\lambda x.x) \to_\beta x\{x/\lambda x.x\} = \lambda x.x$$

Here $(\lambda x.x)\ (\lambda x.x)$ is called a *redex* (the short form of *red*ucible *ex*pression) and $\lambda x.x$ is its *contractum*. In principle this alone is sufficient to define and evaluate every effectively computable function. The only remaining question is: What is the result of a computation?

5.2.3. Normal Forms

A λ-term is said to be in *normal form* (NF) if it is not possible to apply any β-reduction. A normal form can be considered as the outcome of a computation. Note that there are λ-terms that do not have any normal form. For others it might depend on the order of β-steps whether a normal form is reached or not.

Example 5.3. Reducing the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ results in the following reduction sequence

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \to_\beta (\lambda x.x\ x)\ (\lambda x.x\ x) \to_\beta (\lambda x.x\ x)\ (\lambda x.x\ x) \to_\beta \ldots$$

In fact this term does not have a normal form. Compare the above term with the following $(\lambda yz.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$. There are two redexes: The first one is the whole term itself and the second one is $(\lambda x.x\ x)\ (\lambda x.x\ x)$. If the first one is contracted then a normal form is reached immediately (namely $\lambda z.z$), but the second redex can be contracted indefinitely.

5.3. Representing Data Types in the λ-Calculus

To get a grasp of the power of the λ-calculus it is shown how some data types and operations on them that are frequently used in functional programming languages—like Booleans with Boolean connectives, integers with integer arithmetic, pairs, and lists with list operations—can be encoded in the λ-calculus.

5.3.1. Booleans and Conditionals

Consider an expression like `if` b `then` t `else` e. To encode this as a λ-term something of the shape $\lambda bte.s$ is needed, where s has to specify that if b holds then the result should be t and otherwise it should be e. In order to achieve such behavior of s the Boolean values `true` and `false` have to be encoded as λ-terms. One nice possibility is

$$\mathsf{true} \stackrel{\text{def}}{=} \lambda xy.x$$
$$\mathsf{false} \stackrel{\text{def}}{=} \lambda xy.y$$

Then the `if` b `then` t `else` e of OCaml can be encoded as

$$\mathsf{if} \stackrel{\text{def}}{=} \lambda xyz.x\ y\ z$$

since

$$\mathsf{if}\ \mathsf{true}\ t\ e = (\lambda xyz.x\ y\ z)\ (\lambda xy.x)\ t\ e \to_\beta^3 (\lambda xy.x)\ t\ e \to_\beta^2 t$$

and

$$\mathsf{if}\ \mathsf{false}\ t\ e = (\lambda xyz.x\ y\ z)\ (\lambda xy.y)\ t\ e \to_\beta^3 (\lambda xy.y)\ t\ e \to_\beta^2 e$$

5.3.2. Natural Numbers

One way to encode (natural) numbers, i.e., only the non-negative part of integers, in the λ-calculus are the so called *Church numerals*.

Definition 5.2. Let s and t be λ-terms, and $n \in \mathbb{N}$.[1] Then $s^n\ t$ is defined inductively by

$$s^0\ t \stackrel{\text{def}}{=} t$$
$$s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

The Church numerals (represented by $\overline{0}$, $\overline{1}$, $\overline{2}$, ... in the following) are defined by

$$\overline{\mathsf{n}} \stackrel{\text{def}}{=} \lambda f x.f^n\ x$$

Example 5.4. Using the above definition the first four Church numerals are given by

$$\overline{0} \stackrel{\text{def}}{=} \lambda f x.x$$
$$\overline{1} \stackrel{\text{def}}{=} \lambda f x.f\ x$$
$$\overline{2} \stackrel{\text{def}}{=} \lambda f x.f\ (f\ x)$$
$$\overline{3} \stackrel{\text{def}}{=} \lambda f x.f\ (f\ (f\ x))$$

On first sight this definition does not look very obvious (a reason for that could be that it is not), however using the above encoding for numbers, the definitions of addition, multiplication, and exponentiation are very easy, namely

$$\mathsf{add} \stackrel{\text{def}}{=} \lambda m n f x.m\ f\ (n\ f\ x)$$
$$\mathsf{mult} \stackrel{\text{def}}{=} \lambda m n f.m\ (n\ f)$$
$$\mathsf{exp} \stackrel{\text{def}}{=} \lambda m n.n\ m$$

Example 5.5. To familiarize with Church numerals we reduce the λ-term $\mathsf{add}\ \overline{1}\ \overline{2}$ to normal form (the contracted redex is underlined in each step):

$$\begin{aligned}
\mathsf{add}\ \overline{1}\ \overline{2} &= \underline{(\lambda m n f x.m\ f\ (n\ f\ x))\ \overline{1}}\ \overline{2} \\
&\to_\beta \underline{(\lambda n f x.\overline{1}\ f\ (n\ f\ x))\ \overline{2}} \\
&\to_\beta \lambda f x.\overline{1}\ f\ (\overline{2}\ f\ x) \\
&= \lambda f x.\overline{1}\ f\ (\underline{(\lambda f x.f\ (f\ x))\ f}\ x) \\
&\to_\beta \lambda f x.\overline{1}\ f\ (\underline{(\lambda x.f\ (f\ x))\ x}) \\
&\to_\beta \lambda f x.\overline{1}\ f\ (f\ (f\ x)) \\
&= \lambda f x.\underline{(\lambda f x.f\ x)\ f}\ (f\ (f\ x)) \\
&\to_\beta \lambda f x.\underline{(\lambda x.f\ x)\ (f\ (f\ x))} \\
&\to_\beta \lambda f x.f\ (f\ (f\ x)) = \overline{3}
\end{aligned}$$

After we have seen that $\mathsf{add}\ \overline{1}\ \overline{2} \to_\beta^* \overline{3}$ we investigate why the add combinator works as it should. First we explain the $\lambda m n f x.\_$ part. The m and n are the two parameters add will take. The f and the x are needed since the result of $\mathsf{add}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}}$ should be a church numeral, i.e., of the shape $\lambda f x.\_$. Now what happens if we apply add to church numerals $\overline{\mathsf{m}}$ and $\overline{\mathsf{n}}$? We have

$$\mathsf{add}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}} = (\lambda m n f x.m\ f\ (n\ f\ x))\ \overline{\mathsf{m}}\ \overline{\mathsf{n}} \to_\beta^2 (\lambda f x.\overline{\mathsf{m}}\ f\ (\overline{\mathsf{n}}\ f\ x)) \to_\beta^* ?$$

[1] For the purpose of this lecture $\mathbb{N}$ does always denote the set $\{0, 1, 2, 3, \dots\}$ of positive integers together with 0 (which itself is neither negative nor positive).

To determine the ? we first look at the redex $\overline{\mathsf{n}}\ f\ x$. We have

$$\overline{\mathsf{n}}\ f\ x = (\lambda f\ x.f^n\ x)\ f\ x \to^2_\beta f^n\ x$$

and hence for the redex $\overline{\mathsf{m}}\ f\ (\overline{\mathsf{n}}\ fx)$ we have

$$\overline{\mathsf{m}}\ f\ (\overline{\mathsf{n}}\ f\ x) \to^2_\beta \overline{\mathsf{m}}\ f\ (f^n\ x) = (\lambda f\ x.f^m\ x)\ f\ (f^n\ x) \to^2_\beta f^m\ (f^n\ \ x) = f^{m+n}\ x$$

The last identity we leave as an exercise (see Exercise 5.10). Consequently we have that $\mathsf{add}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}} \to^*_\beta \overline{\mathsf{m}+\mathsf{n}}$.

The reasoning for mult is similar but a bit more involved. We have

$$\mathsf{mult}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}} = \lambda mnf.m\ (n\ f) \to^2_\beta \lambda f.\overline{\mathsf{m}}\ (\overline{\mathsf{n}}\ f)$$

and for the redex $\overline{\mathsf{n}}\ f$ we have

$$\overline{\mathsf{n}}\ f = (\lambda fx.f^n\ x)\ f \to_\beta \lambda x.f^n\ x$$

and hence for the redex $\overline{\mathsf{m}}\ (\overline{\mathsf{n}}\ f)$ we have

$$\begin{aligned}\overline{\mathsf{m}}\ (\overline{\mathsf{n}}\ f) &= (\lambda fx.f^m\ x)\ ((\lambda fx.f^n\ x)\ f) \to_\beta (\lambda fx.f^m\ x)\ (\lambda x.f^n\ x)\\ &\to_\beta \lambda x.(\lambda x.f^n\ x)^m\ x \to^*_\beta \lambda x.f^{nm}\ x\end{aligned}$$

We leave the proof for $(\lambda x.f^n\ x)^m\ x \to^*_\beta f^{nm}\ x$ as an exercise (see Exercise 5.11). Consequently $\mathsf{mult}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}} \to^*_\beta \overline{\mathsf{mn}}$.

5.3.3. Pairs

Concerning pairs, some means to construct them—given two values—and to select the first and second component respectively, are needed. This is done via the λ-terms

$$\begin{aligned}\mathsf{pair} &\stackrel{\mathsf{def}}{=} \lambda xyf.f\ x\ y\\ \mathsf{fst} &\stackrel{\mathsf{def}}{=} \lambda p.p\ \mathsf{true}\\ \mathsf{snd} &\stackrel{\mathsf{def}}{=} \lambda p.p\ \mathsf{false}\end{aligned}$$

The reader may already have missed subtraction on natural numbers. The reason is, that pairs are needed before subtraction can be defined. For Church numerals subtraction is an awfully complex (and slow) operation. We will compute subtraction by repeated predecessor operations. Hence the first problem is to get $\lambda fx.f^n\ x$ from $\lambda fx.f^{n+1}\ x$, i.e., get the Church numeral $\overline{\mathsf{n}}$ from $\overline{\mathsf{n+1}}$. Consider the function

$$\mathsf{ffstfst} \stackrel{\mathsf{def}}{=} \lambda fp.\mathsf{pair}\ (f\ (\mathsf{fst}\ p))\ (\mathsf{fst}\ p)$$

which may be easier understandable using pattern matching, i.e.,

$$\mathsf{ffstfst} \stackrel{\mathsf{def}}{=} \lambda f(x,\_).(f\ x, x)$$

If $(\mathsf{ffstfst}\ f)$ is applied $n+1$ times to an argument pair (x,y) then the result is obviously $(\mathsf{ffstfst}\ f)^{n+1}\ (x,y) = (f^{n+1}\ x, f^n\ x)$. The encoding of a Church numeral $\lambda fx.f^n\ x$ basically is the function that applies f^n to x (i.e., f is applied n times to x). Hence the result of

$$(\lambda fx.f^{n+1}\ x)\ (\mathsf{ffstfst}\ f)\ (\mathsf{pair}\ x\ x)$$

is $(f^{n+1}\ x, f^n\ x)$ and by selecting the second component the predecessor $\overline{\mathsf{n}}$ of $\overline{\mathsf{n+1}}$ can be obtained. This facilitates the definitions

$$\begin{aligned}\mathsf{pre} &\stackrel{\mathsf{def}}{=} \lambda nfx.\mathsf{snd}\ (n\ (\mathsf{ffstfst}\ f)\ (\mathsf{pair}\ x\ x))\\ \mathsf{sub} &\stackrel{\mathsf{def}}{=} \lambda mn.n\ \mathsf{pre}\ m\end{aligned}$$

for the predecessor function and subtraction $(m-n)$.

5.3.4. Lists

Having pairing and Booleans, a nonempty list $x :: y$ can be encoded by the nested pairs $\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$ (using pattern matching $(\mathsf{false}, (x, y))$) where false denotes that the list is not empty. Before defining the encoding for empty lists consider the functions that should work on lists. Those are: hd, tl, null (checking whether a list is empty), and cons. Most of them are easy:

$$\mathsf{cons} \stackrel{\mathsf{def}}{=} \lambda xy.\mathsf{pair}\ \mathsf{false}\ (\mathsf{pair}\ x\ y)$$
$$\mathsf{hd} \stackrel{\mathsf{def}}{=} \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z)$$
$$\mathsf{tl} \stackrel{\mathsf{def}}{=} \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z)$$

Now for null the desired result for empty lists is true whereas for nonempty lists it is false. For nonempty lists it would suffice to return the first component of the given pair. It only remains to define nil in a way that $\mathsf{fst}\ l$ evaluates to false for nonempty lists and to true for empty lists. Since $\mathsf{fst} = \lambda p.p\ \mathsf{true}$ the solution is

$$\mathsf{nil} \stackrel{\mathsf{def}}{=} \lambda x.x$$
$$\mathsf{null} \stackrel{\mathsf{def}}{=} \mathsf{fst}$$

5.4. Recursion

Consider an implementation of a recursive function in the λ-calculus. For instance the list function `length`. In OCaml it could be implemented by

```
let rec length x = if x = [] then 0 else 1 + length(tl x)
```

Hence it should be possible to write something like

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \lambda x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (\mathsf{length}\ (\mathsf{tl}\ x)))$$

The only problem here is that the definition of length already needs the length function (which after all is the point of recursive definitions). The idea is to extend length by an additional parameter f and replace all occurrences of length within its definition by this parameter. The result is

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \lambda fx.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x)))$$

Since in the end, length should only take one argument, another λ-term has to be added. Currently it is not clear how this term should look like, but lets call it Y. Then length is defined by

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \mathsf{Y}\ (\lambda fx.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x))))$$

Suppose that Y has the property that for every λ-term t it holds that $\mathsf{Y}\ t \leftrightarrow_\beta^*\ t\ (\mathsf{Y}\ t)$[2] then the following reduction is possible (where length corresponds to $\mathsf{Y}\ t$):

$$\begin{aligned}\mathsf{length} &\to_\beta^* (\lambda fx.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x))))\ \mathsf{length}\\ &\to_\beta \lambda x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (\mathsf{length}\ (\mathsf{tl}\ x)))\end{aligned}$$

which yields the desired result. Indeed such a Y exists.

[2] Here $\leftrightarrow_\beta^*$ means that we can apply β-steps in both directions.

Definition 5.3. The *'Y'-combinator* Y—discovered by Haskell B. Curry—is defined by

$$\mathsf{Y} \stackrel{\text{def}}{=} \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

and has the *fixed point*[3] *property*, i.e., for all λ-terms t

$$\mathsf{Y}\ t \leftrightarrow^*_\beta t\ (\mathsf{Y}\ t)$$

A remarkable result is that the content of this chapter until here is sufficient to encode almost all OCaml programs that were implemented so far during the lecture, only using the λ-calculus. For example, strings are lists of characters, however, on a computer characters are essentially numbers.

5.5. Evaluation Strategy

The *evaluation strategy* (or *strategy* for short) determines which redex to choose if there is more than one possibility. Until now the decision was arbitrary, but when implementing β-reduction the computer needs to know exactly what to do. Two "natural" choices of evaluation strategies are outlined in the following.

5.5.1. Outermost Reduction

The (leftmost) outermost strategy always chooses the (leftmost) outermost redex in a term to apply a β-step. An outermost redex is one that is not a subterm of some other redex.

5.5.2. Innermost Reduction

The (leftmost) innermost strategy always chooses the (leftmost) innermost redex in a term to apply a β-step. An innermost redex is one that does not contain any redexes as proper subterms.

5.5.3. Call-by-Value vs. Call-by-Name

From the above (rewrite) strategies two evaluation strategies for functional programs can be extracted. The first is called *call-by-name* and the second *call-by-value*. Call-by-value is the evaluation strategy adopted by most programming languages. In this evaluation strategy the arguments (value) of a function are evaluated before the function is called on them. For example the function call

```
let f x = x + 1 in f (3 + 2)
```

will be evaluated in the following order in OCaml

```
f (3 + 2) → f 5
          → 5 + 1
          → 6
```

Still it is thinkable to do the derivation in a different order, as in

```
f (3 + 2) → (3 + 2) + 1
          → 5 + 1
          → 6
```

[3] Intuitively speaking, a *fixed point* of a function f is a value v such that applying f to v always results in v. Consequently $v = f\ v = f\ (f\ v) = \cdots = f^n\ v$ for an arbitrary $n \in \mathbb{N}$. The fixed point combinator somehow computes such a fixed point for a given function.

which would be call-by-name, i.e., evaluate the function(-name) and pass the argument as it is.

It can be seen that there is a tight correspondence between call-by-name and outermost reduction as well as between call-by-value and innermost reduction. Next we will elaborate on the slight difference: In addition to only reduce outermost (innermost) redexes, call-by-name (call-by-value) does only consider terms that are in *weak head normal form*.

Definition 5.4. A λ-term t is said to be in *weak head normal form* (WHNF) if it is not a function application, i.e., there do not exist λ-terms u and v s.t.

$$t = u\ v$$

Intuitively this means that the function body of a function without arguments is considered as normal forms, e.g., in

```
let foo = (fun x -> 1 + 2)
```

`foo` is not reduced to `fun x -> 3` as long as it does not get any argument. This is of practical interest because if the function `foo` from above is never applied to any argument in the remainder of the program then the (useless) effort of evaluating `1+2` is avoided. Note that in general the function body might include more costly (even non-terminating) computations.

In other words call-by-name corresponds to outermost reduction to WHNF whereas call-by-value corresponds to innermost reduction to WHNF.

Example 5.6. Consider the λ-term $\mathsf{length}\ \mathsf{nil}$, computing the length of the empty list. This corresponds to the term

$$(\mathsf{Y}\ (\lambda f x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x)))))\ (\lambda x.x)$$

having the eight redexes

$$\mathsf{tl}\ x \tag{5.1}$$
$$\mathsf{add}\ \overline{1} \tag{5.2}$$
$$\mathsf{null}\ x \tag{5.3}$$
$$\mathsf{snd}\ z \tag{5.4}$$
$$\mathsf{if}\ (\mathsf{null}\ x) \tag{5.5}$$
$$\mathsf{snd}\ (\mathsf{snd}\ z) \tag{5.6}$$
$$(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \tag{5.7}$$
$$\mathsf{Y}\ (\lambda f x.\mathsf{if}\ (\mathsf{null}\ x)\ \overline{0}\ (\mathsf{add}\ \overline{1}\ (f\ (\mathsf{tl}\ x)))) \tag{5.8}$$

where (5.4) and (5.6) are hidden in the definition of tl and (5.7) is hidden in the definition of Y. If scanning for redexes starting at the left then the first one obtained is (5.8), which turns out to be the leftmost outermost redex since it is not a subterm of any other redex. Using an outermost strategy will yield the result in some reduction steps. However, the leftmost innermost redex of the above term that is not in WHNF is (5.7). This term is the starting point of the reduction sequence

$$\begin{aligned}(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) &\to_\beta f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\\ &\to_\beta f\ (f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))))\\ &\dots\end{aligned}$$

using innermost reduction to WHNF. This does not terminate. Indeed every recursive definition using Y is nonterminating under call-by-value evaluation.

As can be seen from the above example, Y is not suitable for call-by-value reduction. Gladly there is an alternative to Y which does work also in this case.

Definition 5.5. The *'Z'-combinator* Z—which is a slight variation of Y—is a fixed point combinator (i.e., having the fixed point property) that can be used for call-by-value reduction and is defined by

$$\mathsf{Z} \stackrel{\mathsf{def}}{=} \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$$

As already mentioned, two different evaluation strategies are considered. Call-by-value is tightly connected to *strict* or *eager* evaluation (that is adopted by most imperative programming languages and also by OCaml). A similar connection can be found between call-by-name and *non-strict* or *lazy*[4] evaluation.

5.6. Chapter Notes

Similar examples and further information can be found in [2,5,13]. An important property of the λ-calculus is that β-reduction (=computation) gives unique results, i.e., there is no term s such that $s \to^*_\beta t$ and $s \to^*_\beta u$ with different normal forms t and u. The lambda calculus satisfies this property since it satisfies the stronger "Church-Rosser property", claiming that for all terms t and u with $t \leftrightarrow^*_\beta u$ we can find a common reduct v of t and u, i.e., $t \to^*_\beta v$ and $u \to^*_\beta v$.

5.7. Exercises

Exercise 5.1. Use the conventions backwards to write the following λ-terms in 'full-detail'. Which one is a normal form. Which one is not?

- $\lambda x.x\ y$
- $(\lambda x.x)\ y$

Exercise 5.2. Use the conventions to simplify the following λ-term

$$(\lambda x.(\lambda y.(\lambda z.(((x\ y)\ (y\ x))\ z))))$$

Use the conventions backwards to write the following λ-term in 'full-detail'

$$\lambda abcd.a\ b\ c\ d\ (d\ c\ b\ a)$$

Exercise 5.3. A well-known λ-term is the so called S-*combinator*; defined by

$$\mathsf{S} \stackrel{\mathsf{def}}{=} \lambda xyz.x\ z\ (y\ z)$$

Give set $\mathcal{S}\mathsf{ub}(\mathsf{S})$ of all its subterms.

Exercise 5.4. For each λ-term t out of $\{\lambda x.x\ y, \lambda xy.z, \lambda x.x\ (y\ z)\}$ give the sets $\mathcal{V}\mathsf{ar}(t)$, $\mathcal{BV}\mathsf{ar}(t)$, and $\mathcal{FV}\mathsf{ar}(t)$—the *set of variables*, *bound variables*, and *free variables* in t, respectively.

Exercise 5.5. Consider the substitution $\sigma = \{x/\lambda x.x\}$ and the λ-term $(\lambda xz.x)\ (y\ x)$. What is the result of $t\sigma$, i.e., applying σ to t?

[4] Lazy evaluation and call-by-name are not the same, but they are quite similar. Lazy evaluation corresponds to call-by-name evaluation where additionally a technique called sharing (or memoization) is used to avoid multiple computations of the same expression. However, that's a different story.

Exercise 5.6. Prove that every context $C \in \mathcal{C}(\mathcal{V})$ contains exactly one occurrence of $\Box$.

Exercise 5.7. Consider the λ-term $(\lambda xy.y)\ (\lambda x.x\ x)\ (\lambda x.x\ x)$. Reduce it to normal form.

Exercise 5.8. Reduce each of the following λ-terms to normal form.

$$(\lambda w.w)\ ((\lambda xy.y)\ (z\ z))$$
$$(\lambda xy.x)\ (\lambda z.y\ z)$$
$$\lambda z.(\lambda xy.y\ z)\ (\lambda x.x\ z\ y)$$
$$\lambda xy.y\ (\lambda yz.y\ x\ (\lambda w.w))$$

Exercise 5.9. Reduce $\mathsf{mult}\ \overline{2}\ \overline{3}$ and $\mathsf{exp}\ \overline{2}\ \overline{3}$ to normal form.

Exercise 5.10. Show that $s^l\ (s^k\ t) = s^{l+k}\ t$ for all natural numbers $l, k \in \mathbb{N}$ and lambda terms $s, t \in \mathcal{T}(\mathcal{V})$.
Hint: Use induction on l.

Exercise 5.11. Show that $(\lambda x.f^n\ x)^m\ x \to_\beta^* f^{nm}\ x$ for all natural numbers $n, m \in \mathbb{N}$.
Hint: Use induction on m.

Exercise 5.12. Use the following type for λ-terms

```
type var  = Strng.t
type term = Var of var
          | App of (term * term)
          | Abs of (var  * term)
```

to implement the functions:

```
subterms : term -> term list
vars     : term -> var list
fvars    : term -> var list
bvars    : term -> var list
```

Exercise 5.13. Which of the following terms are in normal form (NF), which are in weak head normal form (WHNF)?

a) $\lambda x.x$

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

c) $(\lambda x.x)\ y\ x$

d) $x\ x$

e) $\lambda x.(\lambda y.y)\ x$

Exercise 5.14. Reduce $\mathsf{add}\ \overline{2}\ \overline{3}$ to WHNF, applying

a) leftmost innermost and

b) leftmost outermost

reduction.

Exercise 5.15. Using the definitions

$$\mathsf{true} = \lambda xy.x$$
$$\mathsf{false} = \lambda xy.y$$

give λ-terms corresponding to the Boolean connectives not, and, and or.
Hint: For not find a λ-term such that $\mathsf{not}\ \mathsf{true} \to_\beta^* \mathsf{false}$ and $\mathsf{not}\ \mathsf{false} \to_\beta^* \mathsf{true}$.

Exercise 5.16. Give all subterms of the λ-term Y. Which of those are proper subterms? For each subterm t_i give the sets $\mathcal{V}\mathsf{ar}(t_i)$, $\mathcal{FV}\mathsf{ar}(t_i)$, and $\mathcal{BV}\mathsf{ar}(t_i)$.

Exercise 5.17. Consider the closed λ-terms

$$\mathsf{I} \stackrel{\mathsf{def}}{=} \lambda x.x,$$
$$\mathsf{K} \stackrel{\mathsf{def}}{=} \lambda xy.x,$$
$$\mathsf{S} \stackrel{\mathsf{def}}{=} \lambda xyz.x\ z\ (y\ z).$$

Give all β-reduction sequences of the term $\mathsf{S}\ \mathsf{K}\ \mathsf{I}\ \mathsf{I}$.

Exercise 5.18. Give a λ-expression succ computing the successor of a Church numeral to which it is applied, i.e., $\mathsf{succ}\ \overline{\mathsf{n}} \to_\beta^* \overline{\mathsf{n+1}}$.

Exercise 5.19. Prove that for every λ-term t there exists a λ-term X such that

$$\mathsf{X} \to_\beta^* t\ \mathsf{X}.$$

Hint: Construct such an X for a given t.

Exercise 5.20. Give a λ-term that corresponds to `Lst.map`.

Exercise 5.21. Show that $\mathsf{Y}\ t \leftrightarrow_\beta^* t\ (\mathsf{Y}\ t)$ for every term t.

Exercise 5.22. Try to reduce the λ-term $\mathsf{hd}\ (\mathsf{cons}\ \overline{\mathsf{1}}\ \mathsf{nil})$ to WHNF using the leftmost innermost strategy.

Exercise 5.23. Reduce the λ-term $\mathsf{hd}\ (\mathsf{cons}\ \overline{\mathsf{1}}\ \mathsf{nil})$ to WHNF using the leftmost outermost strategy.

Exercise 5.24. Consider the infinite list of natural numbers `nats`, defined by

```
let rec from n = n :: from(n+1)
let nats       = from 0
```

Give the computation steps of `hd nats` using call-by-name. What happens using call-by-value?

Exercise 5.25. Consider the list ls = `[1;2;3]`.

a) Write ls as a λ-term L.

b) Reduce $\mathsf{hd}\ (\mathsf{tl}\ \mathsf{L})$ to β-normal form.

c) Reduce $\mathsf{tl}\ \mathsf{nil}$ to β-normal form.

Hint: Use Church numerals to represent natural numbers.

Exercise 5.26. Consider Church numerals $\overline{\mathsf{m}}$ and $\overline{\mathsf{n}}$ and leftmost innermost evaluation.

a) How many β-steps does the evaluation of $\mathsf{add}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}}$ to normal form need?

b) How many β-steps does the evaluation of $\mathsf{mult}\ \overline{\mathsf{m}}\ \overline{\mathsf{n}}$ to normal form need?

c) How do you explain the results?

Exercise 5.27. Consider the following OCaml program

```
\exlabel{ocaml_whnf}
let print s _ = Format.printf "%s␣" s;;

let p = print "hello" ();;
let p1 _ = p;;
let p2 _ = print "world" ();;

p1 ();;
p2 ();;
p1 ();;
```

What is the output of the program (and why)?

Exercise 5.28. Explain the output of the following OCaml program

```
let x = ref 0;;

let f x () = Format.print_int x;;

let p1 = f !x;;
let p2 _ = f !x ();;

x := 1;;

p1 ();;
p2 ();;
```

All animals are equal but some
animals are more equal than others.

George Orwell

6. Reasoning About Functional Programs

Perhaps the most favored property of programs is that they are correct, i.e., do not contain errors (in the end it is often enough to have as few errors as possible). Programmers are sometimes already satisfied when they have a close look at their program and do not find anything wrong. Obviously that does not suffice. The best thing that could happen would be if one was able to *prove* that some program is correct. Since functional programming is so close to mathematics, some mathematical proof methods (most notably *induction*) can directly be applied to functional programs giving rise to rigorous correctness proofs. The process of proving the correctness of programs is called *program verification*. In this chapter one method to verify programs is presented: *Structural induction*.

6.1. Structural Induction

Structural induction is a generalization of induction over natural numbers (aka *mathematical induction*). In mathematical induction the goal is to prove that some property holds for all natural numbers.

Example 6.1. Consider the formula

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}, \tag{6.1}$$

stating that the sum of the first n natural numbers can be computed by the formula $(n \cdot (n+1))/2$.

Proof. Using the principle of mathematical induction this can be proved by first considering the *base case*, which happens to be $n = 0$. Clearly the sum of the first 0 natural numbers is 0. Substituting 0 for n in the rhs (right-hand side) of (6.1) results in

$$\frac{0 \cdot (0+1)}{2} = 0.$$

Hence the statement is true for the base case. Afterwards the *induction step* (or *step case*) is considered. For natural numbers that is proving—under the assumption that the desired property holds for n—that the property does hold for $n+1$. The assumption is called *induction hypothesis* (IH). In the current example the IH is

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}.$$

It remains to be shown, that the lhs (left-hand side) of (6.1) equals the rhs of (6.1) if $n + 1$ is substituted for n. After the substitution the lhs becomes $1 + 2 + \cdots + (n + 1)$

which can be transformed as follows:

$$
\begin{aligned}
&(1+2+\cdots+n)+(n+1) \\
&= \frac{n\cdot(n+1)}{2}+(n+1) && \text{(by IH on } 1+2+\cdots+n\text{)} \\
&= \frac{n\cdot(n+1)+2n+2}{2} && \text{(denominator adaption)} \\
&= \frac{n^2+n+2n+2}{2} && \text{(multiplication)} \\
&= \frac{(n+1)\cdot(n+2)}{2} && \text{(expansion)}
\end{aligned}
$$

which is the rhs of (6.1) where $n+1$ is substituted for n and thus concludes the proof. □

The intuition behind this proof method is the following: Suppose you want to convince yourself that (6.1) does hold for $n = 3$. The available ingredients are a proof that the formula does hold for $n = 0$ and a proof of the implication:

$$\text{If } 1+2+\cdots+n = \frac{n\cdot(n+1)}{2} \text{ then } 1+2+\cdots+n+(n+1) = \frac{(n+1)\cdot(n+2)}{2}.$$

Then starting at $0 = (0\cdot(0+1))/2$ the implication can be used to get the result for 1 namely $1 = (1\cdot(1+1))/2$. Applying the implication another two times yields the desired result $6 = (3\cdot(3+1))/2$. In this way every natural number can be reached and hence the property has to hold for all natural numbers.

Definition 6.1. The principle of *mathematical induction* states that if $P(0)$ holds and $P(n) \to P(n+1)$ holds for all $n \in \mathbb{N}$ and some property P, then $P(n)$ for all $n \in \mathbb{N}$. Put more formally,

$$(P(0) \wedge \forall n.(P(n) \to P(n+1))) \to \forall n.P(n).$$

Now it is obvious that the second bound occurrence of n does not depend on the first and hence it could also be stated, e.g.,

$$(P(0) \wedge \forall k.(P(k) \to P(k+1))) \to \forall n.P(n)$$

since renaming bound variables does not change the meaning.

Let us have a second look at the proof of (6.1). This time concentrating on the different ingredients of the principle of mathematical induction that occur in it. The property P is identified to be

$$P(x) = \left(\sum_{i=1}^{x} i = \frac{x\cdot(x+1)}{2}\right),$$

i.e., the whole equation from (6.1) (which should be indicated by the surrounding parentheses). From a functional programming point of view, P can be seen as a function of type `int -> bool`, returning **`true`** if the given number satisfies (6.1) and **`false`** otherwise. A different reading of the formula

$$(P(0) \wedge \forall k.(P(k) \to P(k+1))) \to \forall n.P(n)$$

would be: "In order to prove $\forall n.P(n)$, it suffices to show that $P(0)$ is true (base case) and $\forall k.(P(k) \to P(k+1))$ is true (step case)." Hence there are two things to show. Firstly $P(0)$ which is called the *base case* and secondly $\forall k.(P(k) \to P(k+1))$ which is called the *step case*. A different reading of the step case would be: "Assuming that $P(k)$ for arbitrary k ($\in \mathbb{N}$) show that also $P(k+1)$." Hence to prove the step case, $P(k)$ is used as a fact (called *induction hypothesis*) and using this fact, $P(k+1)$ has to be shown. Consider the following proof of (6.1)

Base Case The property P has to be shown for 0. By substituting x by 0 in $P(x)$ this translates to

$$\sum_{i=1}^{0} i = \frac{0 \cdot (0+1)}{2}.$$

This is obviously true, hence $P(0)$ has been shown.

Step Case Assume $P(k)$ holds for an arbitrary k, i.e., $\sum_{i=1}^{k} = \frac{k \cdot (k+1)}{2}$. Using this try to show $P(k+1)$, i.e.,

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

This can be shown in a similar way as in the previous proof of (6.1).

Reconsidering what has been shown yields $P(0)$ (from the base case) and $\forall k.(P(k) \rightarrow P(k+1))$ (from the step case, since k has been arbitrary). Combining these two formulas yields

$$P(0) \wedge \forall k.(P(k) \rightarrow P(k+1)).$$

This is exactly the premise of

$$(P(0) \wedge \forall k.(P(k) \rightarrow P(k+1))) \rightarrow \forall n.P(n)$$

and hence it follows that $\forall n.P(n)$ denoting that P is true for all natural numbers.

In the case of structural induction a proof is very similar, the only difference being that the number of base cases and step cases depends on the exact structure induction is applied upon. In the following this *structure* is always a variant type, where the base cases correspond to constructors that do not refer recursively to the defined type and the step cases correspond to those that do.

6.1.1. Structural Induction Over Lists

Recall the type of lists that could be defined by

```
type 'a list = [] | (::) of ('a * 'a list)
```

With respect to induction, lists are very similar to natural numbers. The base case being '`[]`' (i.e., a list of length 0) and the step case $xs \rightarrow x$ `::` xs (i.e., assuming that the property holds for lists of length n it does also hold for lists of length $n+1$). Indeed structural induction over lists is exactly the same as mathematical induction on the length of lists.

Example 6.2. As an example it is proved that the sum of the lengths of two lists is the same as the length of the combined list, i.e., for all lists xs and ys it holds that

$$\texttt{length}\ xs + \texttt{length}\ ys = \texttt{length}(xs\ \texttt{@}\ ys)$$

Where `length` is defined by

```
let rec length = function []    -> 0
                        | _::xs -> 1 + length xs
```

and recall that '`@`' is defined as

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

Proof. We use induction over xs to show the property

$$P(xs) = (\texttt{length}\ xs + \texttt{length}\ ys = \texttt{length}(xs\ \texttt{@}\ ys))$$

Base Case ($xs = \texttt{[]}$). We have to show $P(\texttt{[]})$, i.e.,

$$\texttt{length [] } + \texttt{length}\ ys = \texttt{length}(\texttt{[] @}\ ys)$$

By the definition of `length` the length of an empty list is 0. Hence the lhs equals `length` ys. By the definition of '`@`' the rhs also yields `length` ys.

Step Case ($xs = z :: zs$).
The IH is $P(zs)$, i.e., $\texttt{length}\ zs + \texttt{length}\ ys = \texttt{length}(zs\ \texttt{@}\ ys)$.
We have to show $P(z :: zs)$, i.e.,

$$\texttt{length (}z :: zs\texttt{) + length}\ ys = \texttt{length ((}z :: zs\texttt{) @}\ ys\texttt{)}$$

Let us try to transform the lhs to the corresponding rhs:

$$\begin{aligned}\texttt{length}(z :: zs) + \texttt{length}\ ys &= 1 + \texttt{length}\ zs + \texttt{length}\ ys \\ &\overset{\text{IH}}{=} 1 + \texttt{length}(zs\ \texttt{@}\ ys) \\ &= \texttt{length}(z :: (zs\ \texttt{@}\ ys)) \\ &= \texttt{length}((z :: zs)\ \texttt{@}\ ys)\end{aligned}$$

□

Often induction over lists is used to prove equality between two expressions. If one of the expressions is intuitively easy to understand but inefficient and the other is very complex but fast, then induction is a nice way to make sure that replacing the easy expressions by the complex ones will not alter the result of a program (but maybe the program will be much faster afterwards). However, to prove such equalities there is still something missing. Consider for example the function `hd` that is defined by

```
let hd = function x::_ -> x
                | _    -> failwith "empty list"
```

What is the result of this function if it is applied to an empty list? In terms of program execution some exception is raised and the program is aborted. But for the purpose of induction proofs it is assumed that the result is *undefined*. Therefor the value $\bot$ (speak 'bottom') is introduced, representing undefined results of computations. Then

$$\texttt{hd [] } = \bot.$$

Additionally every function applied to $\bot$ results in $\bot$.[1] E.g., extracting the second element of a list xs by

$$\texttt{hd(tl}\ xs\texttt{)}$$

is $\bot$ if $\texttt{length}\ xs < 2$ and the second element otherwise.

[1] At least for eager evaluation schemes. Since OCaml is a strict functional language, this assumption is fine for proofs about OCaml functions.

Properties of List Functions

Recall the function '@' that is implemented as

```
let rec (@) xs ys = match xs with []      -> ys
                                | x::xs -> x::(xs @ ys)
```

where xs `@` ys is written instead of (`@`) xs ys. First it can be shown that nil is a left identity w.r.t. list concatenation.

Lemma 6.1. *'`[]`' is a left identity of '`@`', i.e.,*

$$[] \mathbin{@} ys = ys$$

for all lists ys.

Proof. This follows immediately from the definition of '`@`'. □

It can also be shown that nil is a right identity for list concatenation.

Lemma 6.2. *'`[]`' is a right identity of '`@`', i.e.,*

$$xs \mathbin{@} [] = xs$$

for all lists xs.

Proof. By induction over the list xs.

Base Case ($xs = []$). By the definition of '`@`' it follows immediately that `[] @ [] = []`.

Step Case ($xs = z :: zs$). The IH is $zs \mathbin{@} [] = zs$.

$$\begin{aligned}(z :: zs) \mathbin{@} [] &= z :: (zs \mathbin{@} []) && \text{(definition of '@')}\\ &\overset{\text{IH}}{=} z :: zs.\end{aligned}$$

□

Then by induction it can be proved that the evaluation order of '`@`' is irrelevant.

Lemma 6.3. *Concatenation of lists is associative, i.e.,*

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

Proof.

Base Case ($xs = []$). Starting at the lhs, the following derivation can be done

$$([] \mathbin{@} ys) \mathbin{@} zs = ys \mathbin{@} zs. \qquad \text{(by Lemma 6.1)}$$

The same result can be obtained starting at the rhs:

$$[] \mathbin{@} (ys \mathbin{@} zs) = ys \mathbin{@} zs. \qquad \text{(by Lemma 6.1)}$$

Step Case ($xs = w :: ws$). The IH is $(ws \mathbin{@} ys) \mathbin{@} zs = ws \mathbin{@} (ys \mathbin{@} zs)$. For the lhs one gets:

$$\begin{aligned}((w :: ws) \mathbin{@} ys) \mathbin{@} zs &= (w :: (ws \mathbin{@} ys)) \mathbin{@} zs && \text{(definition of '@')}\\ &= w :: ((ws \mathbin{@} ys) \mathbin{@} zs) && \text{(definition of '@')}\\ &\overset{\text{IH}}{=} w :: (ws \mathbin{@} (ys \mathbin{@} zs)).\end{aligned}$$

And for the rhs the same result is obtained by the derivation step:

$$(w :: ws) \mathbin{@} (ys \mathbin{@} zs) = w :: (ws \mathbin{@} (ys \mathbin{@} zs)). \qquad \text{(definition of '@')}$$

□

In mathematics a structure consisting of a set (here the set of lists) and a binary operation on it (here list concatenation) such that the binary operation is associative and has an identity element (here the empty list) is called a *monoid*. Hence lists together with list concatenation build a monoid.

Another application of induction is to prove that '`@`' can alternatively be implemented in terms of `foldr`, where `foldr` from page 9 is defined by

```
let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)
```

This amounts to proving the following lemma.

Lemma 6.4.

$$xs \texttt{ @ } ys = \texttt{foldr (fun } z\ zs \texttt{ -> } z \texttt{ :: } zs) \ ys\ xs$$

Proof.

Base Case ($xs = \texttt{[]}$). The base case follows immediately from the definitions of '`@`' and `foldr`.

Step Case ($xs = w \texttt{ :: } ws$). The IH is $ws \texttt{ @ } ys = \texttt{foldr (fun } z\ zs \texttt{ -> } z \texttt{ :: } zs) \ ys\ ws$. Then by transforming the lhs one gets:

$$\begin{aligned}(w \texttt{ :: } ws) \texttt{ @ } ys &= w \texttt{ :: } (ws \texttt{ @ } ys) \\ &\overset{\text{IH}}{=} w \texttt{ :: } (\texttt{foldr (fun } z\ zs \texttt{ -> } z \texttt{ :: } zs) \ ys\ ws) \\ &= \texttt{foldr (fun } z\ zs \texttt{ -> } z \texttt{ :: } zs) \ ys\ (w \texttt{ :: } ws).\end{aligned}$$

□

6.1.2. General Structures

Not only lists are usable for structural induction. In principle every variant type gives rise to possible structural induction proofs over that type.

Binary Trees

As an example binary trees are used to show the more general case of structural induction with several step cases. Recall the definition of the type `'a btree` given by

```
type 'a btree = Empty | Node of ('a btree * 'a * 'a btree)
```

A binary tree is called *perfect* if all leaf nodes have the same height (i.e., all paths from the root to some leaf node have the same length). By structural induction the following lemma can be shown (recall the definition of height on page 20).

Lemma 6.5. *A perfect binary tree t of height k has exactly $2^k - 1$ nodes.*

Proof.

Base Case ($t = \texttt{Empty}$). By definition of `height`, the height of an empty tree is 0. Substituting 0 for k in the goal results in $2^0 - 1 = 0$ which happens to be the number of nodes in an empty tree.

Step Case ($t = \texttt{Node } (l, v, r)$). Since t is a perfect binary tree of height $k + 1$ it follows that l and r are perfect binary trees of respective heights k. Hence by IH it holds that l and r each have $2^k - 1$ nodes. Since t is built by combining l, r, and one additional node, the number of nodes in t equals the number of nodes in l plus the

number of nodes in r plus one, i.e., $2 \cdot (2^k - 1) + 1$. The proof concludes by the following derivation:

$$\begin{aligned} 2 \cdot (2^k - 1) + 1 &= 2 \cdot 2^k - 2 + 1 && \text{(multiplication)} \\ &= 2 \cdot 2^k - 1 && \text{(addition)} \\ &= 2^{k+1} - 1. \end{aligned}$$

□

λ-Terms

Another example are λ-terms. Recall that a λ-term t is of the form

$$t ::= x \mid (\lambda x.t) \mid (t\ t)$$

with $x \in \mathcal{V}$, and that the (OCaml) type of λ-terms is defined by

```
type term = Var of var
          | Abs of (var  * term)
          | App of (term * term)
```

The base case for induction proofs is the case without a recursive reference to the definition of terms itself (i.e., x for λ-terms and `Var` for the OCaml type). For the step cases abstractions ($(\lambda x.t)$ and `Abs`, respectively) and applications ($(t\ t)$ and `App`, respectively) have to be considered for λ-terms and the corresponding type. First let's prove that under the assumption that there is a unique mapping between variables $x \in \mathcal{V}$ and OCaml values of type `var`, there is exactly one instance of the type `term` for every λ-term t. This is done via structural induction over t.

Base Case ($t = x$). For the case of a variable `Var x` can be taken for a uniquely determined value `x` of type `var` by assumption.

Step Case ($t = (\lambda x.s)$). The IH is that there is a unique instance of `term` that corresponds to the term s. Let us call this instance `s`. Then by taking a uniquely determined identifier `x`, the instance `Abs(x,s)` can be built.

Step Case ($t = (u\ v)$). By IH there are unique representations for u and v respectively (since they are both structurally smaller than t). Let us call these values `u` and `v`. Then the instance `App(u,v)` can be built.

It is left as an exercise (cf. Exercise 6.6) to show that (under the above assumption) there is a unique λ-term t for every value of type `term`. Both proofs together establish that λ-terms and values of type `term` are equivalent, i.e., it does not matter whether to use induction over a λ-term t or its `term t`.

Example 6.3. It can be shown that for every λ-term t the application $(t\ t)$, i.e., t applied to itself, has an odd number of opening parentheses.

Proof.

Base Case ($t = x$). The term $(x\ x)$ has one opening parenthesis. Since 1 is an odd number that concludes the base case.

Step Case ($t = (\lambda x.u)$). The IH is that $(u\ u)$ has an odd number of opening parentheses. In the application $((\lambda x.u)\ (\lambda x.u))$ two more opening parentheses are added to those of $(u\ u)$. This results in an odd number.

Step Case ($t = (u\ v)$). By IH $(u\ u)$ and $(v\ v)$ both have an odd number of opening parentheses. In the application $((u\ v)\ (u\ v))$ one more opening parenthesis in addition to those of $(u\ u)$ and $(v\ v)$ is added. This results in an odd number.

□

6.2. Exercises

Exercise 6.1. Use mathematical induction on n to prove for all $n \geq 1$:

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$$

Exercise 6.2. For the induction proof of Exercise 6.1 identify the property P, the base case, the step case, and the induction hypothesis corresponding to the principle of mathematical induction (note that for Exercise 6.1 we start at 1 instead of 0).

Exercise 6.3. Use mathematical induction on n to show that for all $n > 1$:

$$n^3 > 3 \cdot n$$

Exercise 6.4. Prove the following equation by induction over n:

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Exercise 6.5. Prove the following equation by induction over natural numbers:

$$x^{m+n} = x^m \cdot x^n$$

for all natural numbers m, n, and x.

Hint: Use induction over m. Can you also succeed by induction over n?

Exercise 6.6. Prove by structural induction over `t` (i.e., an instance of type `term`) that under the assumption that there is a unique mapping between variables $x \in \mathcal{V}$ and OCaml values of type `var`, there is exactly one λ-term t for every instance `t` of the OCaml type `term` as defined above.

Exercise 6.7. Use structural induction over xs to show that

$$\texttt{map}\ f\ xs = \texttt{foldr}\ c_f\ \texttt{[]}\ xs$$

where $c_f = (\texttt{fun}\ y\ ys\ \texttt{->}\ (f\ y)\ \texttt{::}\ ys)$, `map` is defined by

```
let rec map f = function []    -> []
                       | x::xs -> f x::map f xs
```

and `foldr` by

```
let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)
```

Exercise 6.8. Prove by structural induction that for all lists xs and zs and elements y it holds that

$$(xs\ \texttt{@}\ \texttt{[}y\texttt{]})\ \texttt{@}\ zs = xs\ \texttt{@}\ (y\ \texttt{::}\ zs)$$

where '`@`' is defined by

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

Hint: Use induction on xs.

Exercise 6.9. Can associativity of '`@`' (Lemma 6.3) be used to prove the equation in Exercise 6.8?

Exercise 6.10. Prove the following equation by structural induction over lists:

$$\texttt{reverse}(xs \mathbin{\texttt{@}} ys) = \texttt{reverse}\ ys \mathbin{\texttt{@}} \texttt{reverse}\ xs$$

where '`@`' is defined as

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

and `reverse` is defined by

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]
```

and you may use the equations

$$xs \mathbin{\texttt{@}} [] = xs \tag{$\star$}$$

$$(xs \mathbin{\texttt{@}} ys) \mathbin{\texttt{@}} zs = xs \mathbin{\texttt{@}} (ys \mathbin{\texttt{@}} zs) \tag{$\star\star$}$$

Hint: Use induction on xs.

Exercise 6.11. Prove by structural induction that for all lists xs

$$\texttt{reverse}\ (\texttt{reverse}\ xs) = xs$$

where `reverse` is defined by

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]
```

and '`@`' is defined as

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

Hint: Use the equation from Exercise 6.10.

Exercise 6.12. Consider the functions `length` defined by

```
let rec length = function []    -> 0
                        | _::xs -> 1 + length xs
```

and `length2` defined by

```
let length2 xs = foldr (fun _ acc -> 1 + acc) 0 xs
```

where `foldr` is defined by

```
let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)
```

Prove that `length` xs = `length2` xs for all lists xs.

Exercise 6.13. Consider the following functions

```
let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)

let sum xs = Lst.foldr ( + ) 0 xs
```

```
let rec sum2 = function
 | [] -> 0
 | x::xs -> x + sum2 xs
```

Use induction over xs to show the property

$$P(xs) = (\texttt{sum}\ xs = \texttt{sum2}\ xs)\,.$$

Exercise 6.14. Consider *Towers of Hanoi.*

a) Prove by induction on the height h of the tower that a solution can be obtained in $2^h - 1$ steps.

b) Implement an algorithm which solves *Towers of Hanoi.*

Hint: Consider the slightly stronger property that a tower of height h can be moved from one place to another in $2^h - 1$ steps.

Exercise 6.15. Consider the mirror function from Exercise 4.15. Prove by structural induction that for all binary trees t the following property holds:

$$\texttt{mirror}\ (\texttt{mirror}\ t) = t$$

Exercise 6.16. Consider the following OCaml type for binary trees

```
type 'a t = Empty | Node of ('a t * 'a * 'a t)
```

and the functions:

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]

let rec mirror = function
 | Empty -> Empty
 | Node (l,x,r) -> Node(mirror r,x,mirror l)

let rec flatten = function
 | Empty       -> []
 | Node(l,v,r) -> (flatten l)@(v::flatten r)
```

Prove by structural induction that for all binary trees t

$$\texttt{reverse}\ (\texttt{flatten}\ t) = \texttt{flatten}\ (\texttt{mirror}\ t)$$

You may use

$$xs\ \texttt{@}\ (ys\ \texttt{@}\ zs) = (xs\ \texttt{@}\ ys)\ \texttt{@}\ zs \qquad (\star)$$

$$\texttt{reverse}(xs\ \texttt{@}\ ys) = \texttt{reverse}\ ys\ \texttt{@}\ \texttt{reverse}\ xs \qquad (\star\star)$$

Hint: You may abbreviate `flatten` by `f`, `reverse` by `r`, and `mirror` by `m`.

a) Base case: Show the base case.

b) Step case: Identify the property to prove, the induction hypothesis, and prove the step case.

Software efficiency halves every 18 month, compensating Moore's law.

David May

7. Efficiency

Until now we have almost been recklessly careless about the computational expensiveness of a function. In real world applications however, it often is very important that functions are implemented efficiently (i.e., do not make more steps than absolutely necessary). In the following it is shown that some intuitive function definitions are very inefficient (i.e., there are much faster implementations) and two techniques are provided that often yield more efficient implementations.

7.1. The Fibonacci Numbers

In many textbooks one of the first examples of recursive functions is to compute the n-th Fibonacci number.

Definition 7.1. The *Fibonacci numbers* are given by the equations

$$\begin{aligned} \mathrm{fib}\, 0 &= 1 \\ \mathrm{fib}\, 1 &= 1 \\ \mathrm{fib}\, n &= \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \qquad \text{for } n > 1 \end{aligned}$$

A straightforward implementation in OCaml can directly be inferred:

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

Using the above definition of `fib`, the computation of the n-th Fibonacci number does need an exponential (w.r.t. n) number of recursive calls to itself. The claim is that $2 \cdot \texttt{fib}\ n - 1$ calls to `fib` are needed in order to compute `fib` n. This can be proved by induction on n. Notice that there are two base cases, since the recursion stops either at 1 or at 0.

Lemma 7.1. *To compute the n-th Fibonacci number $2 \cdot \texttt{fib}\ n - 1$ calls to the function `fib` are needed.*

Proof.

Base Case ($n = 0$). To evaluate `fib` 0, one call to `fib` is needed. Since the 0^{th} Fibonacci number is 1 this concludes the first base case.

Base Case ($n = 1$). Also to evaluate `fib` 1, one call to `fib` is needed. Since the 1^{st} Fibonacci number is 1 this concludes the second base case.

Step Case ($n = m + 2$). The IHs are that the number of calls to `fib` when computing $\texttt{fib}(m+1)$ equals $2 \cdot \texttt{fib}(m+1) - 1$ and the number of calls to `fib` when computing `fib` m equals $2 \cdot \texttt{fib}\ m - 1$. It is easily observed that the number of calls to `fib` when computing $\texttt{fib}(m+2)$ is equal to the number of calls needed for $\texttt{fib}(m+1)$ plus the number of calls needed for `fib` m plus 1. Hence

$$\begin{aligned} &(2 \cdot \texttt{fib}(m+1) - 1) + (2 \cdot \texttt{fib}\ m - 1) + 1 \\ &\quad = 2 \cdot (\texttt{fib}(m+1) + \texttt{fib}\ m) - 1 \\ &\quad = 2 \cdot (\texttt{fib}(m+2)) - 1 \end{aligned}$$

□

It remains to be shown, that $2 \cdot \texttt{fib}\ n - 1$ is an exponential number w.r.t. n. Since $2 \cdot \texttt{fib}\ n - 1 \geq \texttt{fib}\ n$ it suffices if $\texttt{fib}\ n \geq 2^{C \cdot n}$ for some constant C. This can be shown by the following derivation:

$$
\begin{aligned}
\texttt{fib}\ n &= \texttt{fib}(n-1) + \texttt{fib}(n-2) && \text{(definition of } \texttt{fib}\text{)} \\
&= \texttt{fib}(n-2) + \texttt{fib}(n-3) + \texttt{fib}(n-2) && \text{(definition of } \texttt{fib}\text{)} \\
&= 2 \cdot \texttt{fib}(n-2) + \texttt{fib}(n-3) \\
&\geq 2 \cdot \texttt{fib}(n-2) \\
&\geq 2 \cdot (2 \cdot \texttt{fib}(n-2-2)) \\
&= 4 \cdot (\texttt{fib}(n-4)) \\
&\geq 4 \cdot (2 \cdot \texttt{fib}(n-4-2)) \\
&\quad \vdots \\
&\geq 2^k \cdot \texttt{fib}(n - 2 \cdot k)
\end{aligned}
$$

for $n > 1$ and $n - 2 \cdot k \geq 0$. If n is even, a base case is reached at $n - 2 \cdot k = 0$, otherwise at $n - 2 \cdot k = 1$. In both cases $k = n/2$ (using integer division). Since $\texttt{fib}\ 0 = \texttt{fib}\ 1 = 1$, the result $\texttt{fib}\ n \geq 2^{n/2}$ is obtained. This inefficiency stems from the fact, that work is repeated unnecessarily. For example to compute `fib` 3, `fib` 2 and `fib` 1 are computed. Then to compute `fib` 2, `fib` 1 (again) and `fib` 0 are computed. Hence `fib` 1 is computed twice, repeating work that has already been done. In order to make the implementation of `fib` more efficient, results that are needed later on in the computation have to be stored somehow.

7.2. Tupling

The technique that can be used to achieve this goal is called *tupling*. Consider for example the function

```
let rec fibpair n = if n < 1 then (0,1) else (
 if n = 1 then (1,1)
          else let (f1,f2) = fibpair (n-1) in (f2,f1+f2)
)
```

Since there is just a single recursive call to `fibpair` in the function body and the argument (n) is reduced by 1, it is clearly the case that only a linear number of recursive function calls is needed. Furthermore it is claimed, that `fibpair` can be used to compute the n-th Fibonacci number.

Lemma 7.2. *The two components of the result of* `fibpair` $(n+1)$ *are the n-th and $(n+1)$-st Fibonacci numbers, i.e.,*

$$\texttt{fibpair}\ (n+1) = (\texttt{fib}\ n, \texttt{fib}\ (n+1))$$

for $n \geqslant 0$.

Proof.

Base Case $(n = 0)$. $\texttt{fibpair}\ 1 = (1,1) = (\texttt{fib}\ 0, \texttt{fib}\ 1)$.

Step Case $(n = m + 1)$. The IH is that `fibpair` $(m+1) = (\texttt{fib}\ m, \texttt{fib}\ (m+1))$. We have to show `fibpair` $(m+2) = (\texttt{fib}\ (m+1), \texttt{fib}\ (m+2))$. The proof concludes by the derivation:

$$\begin{aligned}\texttt{fibpair}\ (m+2) &= (f_2, f_1 + f_2) \quad (\text{with } (f_1, f_2) = \texttt{fibpair}\ (m+1)) \\ &\overset{\text{IH}}{=} (\texttt{fib}\ (m+1), \texttt{fib}\ m + \texttt{fib}\ (m+1)) \\ &= (\texttt{fib}\ (m+1), \texttt{fib}\ (m+2)).\end{aligned}$$

□

Lemma 7.3. *The function* `fibpair` *can be used to implement* `fib` *as follows:*

```
let fib n = snd(fibpair n)
```

Proof. From Lemma 7.2 it is known that `fibpair` $n = (\texttt{fib}(n-1), \texttt{fib}\ n)$, i.e., for $n > 0$ the second component is the n-th Fibonacci number. Since `fibpair` $0 = (0, 1)$, also for $n = 0$, the second component is the n-th Fibonacci number. □

In general, tupling is used to modify existing functions in a way that they return more than one result, aiming at a more efficient implementation. Consider for example a function `average`, computing the average of the elements of an integer list. Therefor the sum of all elements and the number of elements is needed. This could be implemented as

```
let average xs = IntList.sum xs / Lst.length xs
```

however, in this case the list `xs` is traversed twice, once to compute the sum, and a second time to compute the length of the list. This could be combined into the function

```
let rec sumlen = function
  | []    -> (0,0)
  | x::xs -> let (s,l) = sumlen xs in (x+s,l+1)
```

Then `average` can be implemented by

```
let average xs = let (s,l) = sumlen xs in s/l
```

7.3. Tail Recursion

A special kind of recursion is *tail recursion*. A function is said to be tail recursive, if the recursive call is the *last* thing to do.[1] This kind of recursion is special, since it can automatically be transformed (by the compiler) into a loop, that does need constant stack space only. Therefore tail recursive functions are sometimes also called *iterative*.

On a standard computer the function stack (or call stack, or execution stack) stores information about a function call until it is finished. Hence at the call, information is pushed on top of the stack and as the function finishes, popped off the stack. However, if a recursive call occurs within a function call, then, the information for this call is pushed on top of the stack before popping the former call. If the recursive call again causes a recursive call additional call information is pushed on top of the stack. This continues until the last recursive call, after which, the call information can be popped one after the other computing the result of the function. Hence the used stack space depends on the size of the input, e.g., a recursive function on a list containing 100,000 elements, will push 100,000 entries on top of the stack before removing anything. If the stack grows too big, a stack overflow is generated and the function is aborted.

[1] Note that as a consequence there can be at most one recursive call.

Tail recursion does circumvent this problem, since any serious compiler will transform a tail recursive function into machine code using constant stack space only. The concept from the next section will give us the means to write tail recursive functions.

7.4. Parameter Accumulation

For tupling the idea was to introduce additional result values to a function. In *parameter accumulation* the idea is to introduce new parameters that are used to transfer intermediate results between function calls. In this way, often tail recursive variants of existing functions can be achieved. E.g., the above `sumlen` is not tail recursive (after the recursive call additions are performed and a pair is constructed). Consider the following implementation

```
let rec sumlen_acc sum len = function
 | []    -> (sum,len)
 | x::xs -> sumlen_acc (x+sum) (len+1) xs

let sumlen xs = sumlen_acc 0 0 xs
```

which can also be written as

```
let sumlen xs =
 let rec sumlen sum len = function
  | []    -> (sum,len)
  | x::xs -> sumlen (x+sum) (len+1) xs
 in
 sumlen 0 0 xs
```

The second variant is used more often in this lecture since most of the time the auxiliary functions are not needed somewhere else.

As a second example consider the `range` function of the `IntLst` module. Try to evaluate `range 1 1000000`. Maybe a tail recursive version of `range` could do a better job. But then the resulting list has to be built from right to left rather than from left to right, i.e.,

```
let range_tl m n =
 let rec range acc m n =
  if m >= n then acc else range ((n-1)::acc) m (n-1)
 in
 range [] m n
```

7.5. Linear vs. Quadratic Complexity

In this section we demonstrate that for large values of n it can already be problematic if a function runs in time $O(n^2)$.

Consider the function

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

Obviously this function takes linear time in the size of its first parameter, which is harmless. More precisely if xs has n elements, then xs `@` ys takes $n + 1$ computation steps (do not forget the base case, i.e., when $xs =$ `[]`).

Let us in addition consider the function

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]
```

which calls itself and '@' linearly often. While this might also look harmless (at first sight), it definitely is not. What happens is the following. The function '@' is called linearly often but has linear complexity itself, resulting in a quadratic complexity in total. This can be seen if we evaluate `reverse [1;...;n]` as follows:

$$\begin{aligned}
\texttt{reverse}\ [1;\ldots;n] &\to^{n+1} ((([]\ \texttt{@}\ [n])\ \texttt{@}\ [n-1])\ \texttt{@}\ [n-2])\ \texttt{@}\ \cdots\ \texttt{@}\ [1] \\
&\to^{1} (([n]\ \texttt{@}\ [n-1])\ \texttt{@}\ [n-2])\ \texttt{@}\ \cdots\ \texttt{@}\ [1] \\
&\to^{2} ([n;n-1]\ \texttt{@}\ [n-2])\ \texttt{@}\ \cdots\ \texttt{@}\ [1] \\
&\to^{3} [n;n-1;n-2]\ \texttt{@}\ \cdots\ \texttt{@}\ [1] \\
&\ldots \\
&\to^{n} [n;n-1;n-2;\ldots;1]
\end{aligned}$$

Hence in total we have

$$\begin{aligned}
(n+1)+1+2+3+\ldots+n &= (n+1)+\frac{n\cdot(n+1)}{2} \\
&= \frac{(n+1)\cdot(n+2)}{2} \in O(n^2)
\end{aligned}$$

computation steps. Here we used (6.1) in the first equality.

Consider in contrast the alternative implementation:

```
let rev xs =
 let rec rev acc = function []    -> acc
                          | x::xs -> rev (x::acc) xs
 in
 rev [] xs
```

With the help of the accumulator this function needs linear time only (to be more precise for a list of length n this function requires $n+1$ computation steps). We leave it as an exercise to evaluate these two functions for lists of arbitrary length to see the (dramatic) differences in execution time for larger n, e.g., $n = 100000$.

7.6. Chapter Notes

See also [15] for a discussion on how recursive algorithms should be treated in computer science courses. Additionally to Fibonacci numbers also binomial coefficients are used there as example. Note that for the Fibonacci numbers a closed expression is known, i.e.,

$$\text{fib}(n) = \frac{1}{\sqrt{5}}\cdot\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\cdot\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

7.7. Exercises

Exercise 7.1. Which of the following functions is tail recursive, which one is not?

```
let rec map f = function []    -> []
                       | x::xs -> f x::map f xs

let rec foldl f b = function []    -> b
                           | x::xs -> foldl f (f b x) xs

let rec foldr f b = function []    -> b
                           | x::xs -> f x (foldr f b xs)
```

Exercise 7.2. Implement a *tail recursive* function `rev_append_tl` such that

`rev_append_tl [1;2;3] [4;5;6]` = `[3;2;1;4;5;6]`

Hint: Your function should have the type `'a list -> 'a list -> 'a list`

Exercise 7.3. Consider '`@`' defined as

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

and `reverse` defined as

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]
```

Prove by induction on xs that your function `rev_append_tl` from Exercise 7.2 satisfies

(`reverse` xs) `@` ys = `rev_append_tl` xs ys

Hint: You may use Lemma 6.3.
Bonus: Can you also prove xs `@` ys = `rev_append_tl` (`reverse` xs) ys?

Exercise 7.4. Consider the function `Lst.replicate`

```
let rec replicate n x =
 if n < 1 then [] else x::replicate (n-1) x
```

Is this function tail recursive? If yes, justify your answer, otherwise give a tail recursive implementation.

Exercise 7.5. Give a *tail recursive* version of the function `Lst.length`

```
let rec length = function []    -> 0
                        | _::xs -> 1 + length xs
```

Exercise 7.6. Use induction over lists to prove that your function from Exercise 7.5, produces the same results as the non-tail recursive one.

Exercise 7.7. Is the function `fibpair` from Chapter 7.2 tail recursive? If yes, justify your answer, otherwise give a tail recursive implementation.

Exercise 7.8. Use *tupling* to implement a version of `Lst.split_at`

```
let split_at n xs = (take n xs,drop n xs)
```

that just needs one list traversal to compute its result.

Exercise 7.9. Use *tupling* to implement a version of `Lst.span`

```
let span p xs = (take_while p xs, drop_while p xs)
```

that just needs one list traversal to compute its result.

Exercise 7.10. Use induction over lists to prove the equation

`sumlen` xs = (`sum` xs, `length` xs)

using the function definitions

```
let rec sum = function []    -> 0
                     | x::xs -> x + sum xs

let rec length = function []    -> 0
                        | _::xs -> 1 + length xs

let rec sumlen = function
 | []    -> (0,0)
 | x::xs -> let (s,l) = sumlen xs in (s+x,l+1)
```

Exercise 7.11. Find a non tail recursive function in the modules from the lecture that has not been treated yet. Justify why it is not tail recursive.

Exercise 7.12. Consider '`@`' in OCaml, i.e.,

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

a) From Lemma 6.3 we know that '`@`' is associative. However, internally OCaml has to evaluate the operator either left- or right-associative. What is the choice for '`@`' in OCaml? Give evidence.

Hint: Which of the following reductions is performed by OCaml: Either

$$\texttt{[1]@[2]@[3]} \to \texttt{[1;2]@[3]} \to \texttt{[1;2;3]}$$

or

$$\texttt{[1]@[2]@[3]} \to \texttt{[1]@[2;3]} \to \texttt{[1;2;3]}\ ?$$

b) Determine the number of computation steps OCaml needs to evaluate

i) `(([1]@[2])@` $\cdots$ `)@[`n`]` (left-associative)

ii) `[1]@([2]@(` $\cdots$ `@[`n`]))` (right-associative)

Hint: Recall that OCaml adopts an eager evaluation strategy.

Exercise 7.13. Consider the following functions:

```
let rec reverse = function    [] -> []
                          | x::xs -> (reverse xs) @ [x]

let rev xs =
 let rec rev acc = function []    -> acc
                          | x::xs -> rev (x::acc) xs
 in
 rev [] xs
```

Show that for all lists xs we have

$$\texttt{reverse}\ xs = \texttt{rev}\ xs\,.$$

You may use Lemma 6.3.

Hint: Show (`reverse` xs) `@` acc = `rev` acc xs by structural induction on xs and conclude the result using Lemma 6.2.

Exercise 7.14. Implement a tail recursive variant of `foldr`.

Words – so innocent and powerless as they are, as standing in a dictionary, how potent for good and evil they become in the hands of one who knows how to combine them.

Nathaniel Hawthorne

8. Combinator Parsing

Often some information is only available in a textual representation—like for example the source code of a program—but a more structured representation would be desirable. The process of scanning textual input and transforming it to something more structured is called *parsing*.[1] Parsing is a fundamental part of many applications in computer science, e.g., every program source code has to be parsed in order to be transformed to something understandable by the machine. Parsing can be implemented in different ways, to mention two: finite automata and parser combinators. In the following the latter method is used, since it is a very nice application of the functional programming paradigm. It is assumed that textual information is given as a sequence of tokens.

Hence a parser is a function of type

```
type ('a,'t)t = 't list -> ('a * 't list)option
```

where the input is a sequence (i.e., list) of tokens (`'t list`) and the output is an option type. If a parser returns `None` this indicates that an error was encountered and otherwise the result is `Some (x,ts)` where `x` is the parsed object and `ts` is the remaining list of tokens to be parsed.

Combinator parsing is very modular. The smallest components are primitive parsers and character parsers. Parser combinators are then used to create more complex parsers from some given ones.

8.1. Implementation of Parsers

In the following the internal representation of parsers used in this chapter is discussed and some basic examples are given.

8.1.1. Applying a Parser

Before parsers (i.e., functions that are used for parsing) are defined, some framework to apply a parser to a given input is needed. Therefor consider

```
let parse p input = match p input with None      -> None
                                     | Some (x,_) -> Some x
```

taking a parser `p` and some l-string `input` as arguments and returning the result of applying `p` to `input` (if it exists). For testing purposes also a version of `parse` that works on OCaml strings (`s`) and raises an exception on failure is given

```
let test p s = match p (Strng.of_string s) with
  | None      -> failwith "parse␣error"
  | Some (x,ts) -> (x,ts)
```

[1] More precisely this process happens in two stages. In the first stage the *lexer* decomposes the stream of input symbols into so called *tokens*. This can be seen as the syntactic check of the program. In the second stage the *parser* then checks if the sequence of tokens corresponds to sentences of a given grammar. This can be seen as the semantic check of the program. Since humans tend to be imprecise whenever possible, both processes together are also referred to as parsing.

8.1.2. Lexing

To process a single token, `token` takes a function `f` and a list of tokens as parameters and applies `f` to the first token. For simplicity, in our setting a single token will always be a character.

```
let token f = function
  | []    -> None
  | t::ts -> match f t with
    | Some x -> Some (x,ts)
    | None   -> None
```

Note that `f` is not a parser since `f : 't -> 'a option` but `token f` has the correct type.

8.1.3. Some Simple Parsers

The first example of a parser does just accept any token satisfying a given property and returns it (or indicates an error if either the property was not satisfied or no input was left), i.e.,

```
let sat p ts = token(fun t -> if p t then Some t else None) ts
```

Note that `sat : ('t -> bool) -> ('t, 't) t`. Hence applying to `sat` a function that tests a property of a token (`'t -> bool`) yields a parser. Using `sat` a parser accepting any token can be defined as follows

```
let any ts = sat (fun _ -> true) ts
```

If this parser is applied to any input that is not empty, it will succeed.

Example 8.1. The `any` parser can be tested on the input `"test"`. This is done via the function call

```
test any "test"
```

where the result will be `('t',"est")`, i.e., the first component of the pair is the parsed character of the given input and the remaining input will be `"est"`. [2] Applying `any` to an empty input, i.e.,

```
test any ""
```

will result in an error.

Again using `sat` we can define a character parser accepting only a specific character

```
let char c ts = sat ((=) c) ts
```

The parser `char c` does only succeed if the next character in the input is `c`.

8.1.4. Parser Combinators

Parser combinators make more complex parsers using the simple parsers from before. Often (but not always) they combine two parsers into a single one.

The first parser combinator discussed here—called *bind*[3]—is used to implement sequential composition. The special infix syntax `(>>=)` is given to *bind*, to make its use more comfortable. Bind applies its first argument (a parser) `p` to the input. If the result is an error then the error is passed on. Otherwise the result (`x`) of the first parser is given to `f`, which results in a second parser that in turn is applied to the remaining input `ts`.

[2] In the sequel we assume that the *pretty printer* for the `Strng` module is installed, i.e., a char list `['H';'e';'l';'l';'o']` is then printed as `"Hello"`.

[3] The name stems from its connection to monads. Never mind if you don't know what monads are.

```
let (>>=) p f ts = match p ts with None      -> None
                                 | Some(x,ts) -> f x ts
```

From the type of bind `(>>=) : ('a,'t)t -> ('a -> ('b,'t)t) -> ('b,'t)t` we see that `p` is a parser and `f` yields a parser, if applied to the result of `p`.

Lemma 8.1. Bind *is associative, i.e.,* $(p \texttt{ >>= } f) \texttt{ >>= } g = p \texttt{ >>= } (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g)$.[4]

Proof. As usual we consider two functions as equal, if they produce the same result on the same input. Hence the above equation is true if and only if

$$((p \texttt{ >>= } f) \texttt{ >>= } g)\ ts = (p \texttt{ >>= } (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g))\ ts$$

for every input ts. Then the property follows by case distinction on $p\ ts$.

Case 1 ($p\ ts = \texttt{None}$). Hence

$$\begin{aligned}((p \texttt{ >>= } f) \texttt{ >>= } g)\ ts &= \texttt{match } (p \texttt{ >>= } f)\ ts \texttt{ with } \ldots \\ &= \texttt{None} \\ &= (p \texttt{ >>= } (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g))\ ts\end{aligned}$$

Case 2 ($p\ ts = \texttt{Some}(y, j)$). There are again two cases

Case a ($f\ y\ j = \texttt{None}$). Starting from the lhs we get

$$\begin{aligned}((p \texttt{ >>= } f) \texttt{ >>= } g)\ ts &= \texttt{match } (p \texttt{ >>= } f)\ ts \texttt{ with } \ldots \\ &= \texttt{match } f\ y\ j \texttt{ with } \ldots \\ &= \texttt{None}\end{aligned}$$

And starting from the rhs we also get

$$\begin{aligned}(p \texttt{ >>= } (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g))\ ts &= (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g)\ y\ j \\ &= (f\ y \texttt{ >>= } g)\ j \\ &= \texttt{None}\end{aligned}$$

Case b ($f\ y\ j = \texttt{Some}(z, k)$). Starting from the lhs we get

$$\begin{aligned}((p \texttt{ >>= } f) \texttt{ >>= } g)\ ts &= \texttt{match } (p \texttt{ >>= } f)\ ts \texttt{ with } \ldots \\ &= \texttt{match } f\ y\ j \texttt{ with } \ldots \\ &= g\ z\ k\end{aligned}$$

And starting from the rhs we also get

$$\begin{aligned}(p \texttt{ >>= } (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g))\ ts &= (\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g)\ y\ j \\ &= (f\ y \texttt{ >>= } g)\ j \\ &= \texttt{match } f\ y\ j \texttt{ with } \ldots \\ &= g\ z\ k\end{aligned}$$

□

[4] Because of types we cannot write $p \texttt{ >>= } (f \texttt{ >>= } g)$ since f is not a parser (which is required by the second bind). Hence we write $\texttt{fun } x \texttt{ -> } f\ x \texttt{ >>= } g$, being a function that yields a parser. Note that $f\ x$ is a parser.

Example 8.2. A possible application of *bind* would be to scan the first two characters of the input. This is easily achieved by combining two `any` parsers using `(>>=)`.

```
let any_pair = any >>= fun c -> any >>= fun d -> return(c,d)
```

Note that no parentheses are needed since `(>>=)` is associative and **`fun`** associates to the right. Hence the above code does the same as

```
let any_pair = (any >>= (fun c -> (any >>= (fun d -> return(c,d)))))
```

Somehow similar to *bind* is the combinator *then*

```
let (>>) p q ts = (>>=) p (fun _ -> q) ts
```

only that the output of the first parser is ignored.

Example 8.3. The following parser accepts an opening parenthesis followed by a closing parenthesis (i.e., `"()"`).

```
let open_close = char '(' >> char ')'
```

This parser is successful on any input that starts with `"()"`. E.g.,

```
test open_close "(()())"
```

returns the pair `(')',"(())")` (the result of `char ')'`), whereas

```
test open_close ")("
```

fails.

The parser `return : 'a -> ('a,'t) t` is used to turn an arbitrary value into a parser returning that value and leaving the input unchanged.

```
let return x = fun ts -> Some (x,ts)
```

There are several more parser combinators. Consider `(<|>)`—called *choice*—taking two parsers `p` and `q`, and creating a new one that returns the result of `p` if it was successful and otherwise applies `q`.

```
let (<|>) p q ts = match p ts with None        -> q ts
                                 | Some _ as r -> r
```

Example 8.4. Using `(<|>)` the example from above can be modified to accept an arbitrary string of balanced parentheses given by the BNF:

$$p ::= (p)p \mid \epsilon$$

For the one to one translation of the grammar

```
let rec parens =
 (char '(' >> parens >> char ')' >> parens) <|> return()
```

the OCaml compiler detects that this statement would never terminate (since the recursive call to `parens` would be executed immediately; recall that OCaml has an eager evaluation strategy).

Introducing a dummy argument, and demanding the evaluation of `char '('` (and `char')'`) before the recursive calls to `parens` fixes this problem:

```
let rec parens() = (
  char '(' >>= fun _ ->
  parens() >>
  char ')' >>= fun _ ->
  parens()
 ) <|> return()
```

Then `parens` (apparently) works

```
# test (parens ()) "(()(()))"
- : unit * char list = ((), "")
```

but does always succeed, i.e., unbalanced parentheses are not parsed but just remain in the list of tokens. To see this consider the call:

```
# test (parens ()) "()((())(";;
- : unit * char list = ((), "((())(")
```

Therefore a parser would be handy that only accepts, if the end of the input is reached. This can be implemented by

```
let eoi = function [] -> Some ((),[])
                 | _  -> None
```

Now `(parens() >> eoi)` can be used to create a parser that only accepts if the full input is consumed and corresponds to the above grammar.

Finally we discuss the *option*

```
let (?>) p d = p <|> return d
```

which returns the result of applying `p` (if successful) and otherwise the default value `d` is returned.

8.1.5. Giving Parsers Work

In the examples so far, the purpose of the parsers was just to succeed if the input corresponded to some grammar and to fail otherwise. Actually most of the time parsers are used to generate some result from the information parsed. This is achieved using the `return` function.

Example 8.5. We want to write a parser that computes the maximum nesting depth of some input fitting the grammar of Example 8.4. The solution that yields the desired behaviour is

```
let rec nesting() =
 (char '('  >>= fun _ ->
  nesting() >>= fun i ->
  char ')'  >>= fun _ ->
  nesting() >>= fun j ->
  return(max (i+1) j)
 ) <|> return 0
```

We remove the dummy argument by

```
let nesting = nesting ()
```

and test the parser

```
# test nesting "()"
-: int * char list = (1, "")
```

which looks promising. Fixing the (open) issue that this parser never fails (it should fail if the input does not correspond to the grammar in Example 8.4, is left as an exercise, cf. Exercise 8.3).

8.2. The Parser Module

Additionally to the primitive parsers, character parsers, and parser combinators that have been presented above, there are many others implemented in the module `Parser`. To use the module, nothing has to be known about the implementation of these functions. For the user just the interface is interesting, which can be found in Listing 8.1. A short description of each function that has not already been described follows:

`between`. The parser resulting from `between o p c` accepts the same as the parser `p` but enclosed in `o` and `c`, i.e., the accepted input is the accepted input of the parser `o` (open), followed by the accepted input of `p`, followed by the accepted input of `c` (close). The result is the result of `p`.

Example 8.6. A parser accepting any character enclosed in braces is

```
let braced_char = between (char '{') any (char '}')
```

`digit`. This is a character parser that only accepts a single digit, i.e., exactly one of ‘`'0'`’, ‘`'1'`’, . . . , ‘`'9'`’, and returns it as character.

`letter`. This is a character parser that accepts any (lower or uppercase) letter, i.e., ‘`'a'`’, . . . , ‘`'z'`’, ‘`'A'`’, . . . , ‘`'Z'`’.

`many`. The parser resulting from `many p` applies the parser `p` *zero* or more times and returns a list of the returned values of `p`. Note that `many` is greedy, i.e., as many applications of `p` as possible are performed.

Example 8.7. Usually a *word* is defined to consist of letters. Hence a parser for arbitrary words could be implemented and used as follows:

```
# let word = many letter;;
val word : (char list, char) Parser.t = <abstr>
# test word "hello, world!";;
- : char list * char list = ("hello", ",␣world!")
# test word "123hello";;
- : char list * char list = ([], "123word")
```

The only problem is that also the empty input is accepted (it mainly depends on the exact application of the parser whether this really is a problem or not). To avoid this, the next combinator is useful.

`many1`. This combinator works exactly like `many`, only that the empty input is not accepted, i.e., for `many p` to accept, `p` has to accept at least once.

Example 8.8. By redefining the `word` parser as follows

```
# let word = many1 letter;;
```

only non empty words are accepted. Here are two example runs:

```
# test word "hello";;
- : char list = ("hello", "")
# test word "123hello";;
Exception: Failure "parse␣error".
```

`noneof`. A character parser that accepts any character except those specified in the given string, e.g., `noneof "hello"` would accept any single character except ‘`'h'`’, ‘`'e'`’, ‘`'l'`’, or ‘`'o'`’. The result is the accepted character.

```
(* type t *)
type ('a,'t)t
(* *)

(** Primitive Parsers *)
val eoi : (unit,'t)t
val return : 'a -> ('a,'t)t
val token : ('t -> 'a option) -> ('a,'t)t

(** Parsers *)
val sat : ('t -> bool) -> ('t,'t)t

(** Primitive Combinators *)
(** Combinators *)
val (?>) : ('a,'t)t -> 'a -> ('a,'t)t
val (>>=) : ('a,'t)t -> ('a -> ('b,'t)t) -> ('b,'t)t
val (>>) : ('a,'t)t -> ('b,'t)t -> ('b,'t)t
val (<|>) : ('a,'t)t -> ('a,'t)t -> ('a,'t)t
val between : ('a,'t)t -> ('b,'t)t -> ('c,'t)t -> ('b,'t)t
val many1 : ('a,'t)t -> ('a list,'t)t
val many : ('a,'t)t -> ('a list,'t)t
val sep_by1 : ('a,'t)t -> ('b,'t)t -> ('b list,'t)t
val sep_by : ('a,'t)t -> ('b,'t)t -> ('b list,'t)t

(** Character Parsers *)
val any : (char,char)t
val char : char -> (char,char)t
val digit : (char,char)t
val letter : (char,char)t
val noneof : string -> (char,char)t
val oneof : string -> (char,char)t
val space : (char,char)t
val spaces : (int,char)t
val string : string -> (char list,char)t

(** Running Parsers on Input *)
val parse : ('a,'t)t -> 't list -> 'a option
val test : ('a,char)t -> string -> ('a * char list)
(* *)
```

Listing 8.1: `Parser.mli`

`oneof`. Behaves exactly like `noneof`, only that the set of *accepted* characters is specified.

`sep_by`. This combinator takes two parsers `s` and `p`. Then `sep_by s p` parses *zero* or more occurrences of `p`, separated by `s` and returns the list of values returned by `p`.

Example 8.9. For example a parser for a comma separated list of characters could be used as follows:

```
# let comma_chars = sep_by (char ',') any;;
val comma_chars : (char list, char) Parser.t = <abstr>
# test comma_chars "";;
- : char list * char list = ("", "")
# test comma_chars "a";;
- : char list * char list ("a", "");;
# test comma_chars "h,e,l,l,o";;
- : char list * char list = ("hello", "")
```

Again also the empty input is accepted.

`sep_by1`. This is the obvious restriction of `sep_by`.

`space`. A character parser that consumes an arbitrary white space character, i.e., '`' '`', '`'\n'`', or '`'\t'`' and returns nothing.

`spaces`. A parser that accepts an arbitrary (possibly empty) sequence of white spaces.

`string`. The parser `string "test"` accepts if and only if the input starts with `"test"`.

8.3. A Parser for Simplified Arithmetic Expressions

In the following an example parser is given that transforms a string into an abstract syntax tree for arithmetic expressions. The grammar for (the somehow simplified) arithmetic expressions is

$$
\begin{aligned}
e &::= e \texttt{ + } t \mid t \\
t &::= t \texttt{ * } f \mid f \\
f &::= \texttt{(} e \texttt{)} \mid n \\
n &::= d\ n \mid d \\
d &::= \texttt{0} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

where the hierarchy of the grammar corresponds to the operator priority. The type for the abstract syntax tree is given by

```
type arith = Num of int
           | Add of arith * arith
           | Mul of arith * arith
```

A first approach for a parser could be

```
let rec e() =
 (e() >>= fun e1 -> char '+' >> t() >>= fun e2 -> return(Add(e1,e2)))
 <|> (t())
and t() =
 (t() >>= fun t1 -> char '*' >> f() >>= fun t2 -> return(Mul(t1,t2)))
 <|> (f())
and f() = (
  char '(' >>= fun _ ->
```

```
  e()      >>= fun e1 ->
  char ')' >>
  return e1
) <|> n
and n = many1 digit >>= fun r -> return(Num r)
```

There is a "slight" problem however. Since the first thing that `e()` does, is recursively calling itself (before checking any break condition) this parser does *always* loop forever. The problem is caused by the fact that above grammar is *left recursive*. Gladly, every left recursive grammar can be transformed into a non left recursive one. For the above grammar, the result after eliminating left recursion is

$$
\begin{array}{lll}
e ::= t\ e' & e' ::= \texttt{+}\ t\ e' \mid \epsilon & \\
t ::= f\ t' & t' ::= \texttt{*}\ f\ t' \mid \epsilon & \\
f ::= \texttt{(}\ e\ \texttt{)} \mid n & n ::= d\ n \mid d & d ::= \texttt{0} \mid \ldots \mid \texttt{9}
\end{array}
$$

where the straightforward implementation (okay at first sight it looks complex, but by *straightforward* it is meant that once you are used to this kind of transformations it is very easy to apply them) is:

```
let rec e() = t() >>= e'
and e' term = (
  char '+' >>= fun _ ->
  t()      >>=
  e'       >>= fun t2 ->
  return(Add(term,t2))
) <|> return term
and t() = f() >>= t'
and t' factor = (
  char '*' >>= fun _ ->
  f()      >>=
  t'       >>= fun f2 ->
  return(Mul(factor,f2))
) <|> return factor
and f() = (
  char '(' >>= fun _ ->
  e()      >>= fun e1 ->
  char ')' >>
  return e1
) <|> n
and n = many1 digit >>= fun r ->
 return (Num (int_of_string (Strng.to_string r)))
```

In the example two things can be seen: Firstly one should think about a grammar before starting to implement it and secondly there is much more to learn in the field of formal languages that is not part of this lecture (e.g., how to eliminate left recursion).

8.4. Chapter Notes

In the literature, combinator parsers are not new. However, efficient combinator parsers are rare. In [8] and [7] the implementation of an efficient combinator parser library is discussed. The module `Parser` is based on the PARSEC library for Haskell and many of the above examples originate from [7].

Also note that in realistic applications, parsing is usually split into two phases: The first is usually called *lexing* and it converts the raw input into so called *tokens*. (It is of course possible to implement a *lexer* using combinator parsing.) The second phase—the

actual *parsing*—then operates on the stream of tokens generated by the lexer. All our example parsers work as if single characters would be the tokens (which is possible but tends to require parsers that take care about low-level stuff like separating white spaces).

8.5. Exercises

Exercise 8.1. Write a parser

```
uibk_mail : (Strng.t * Strng.t, char)Parser.t
```

that accepts an email address as used for students at the university of Innsbruck, i.e.,

$$l^+.l^+\texttt{@student.uibk.ac.at}$$

where l is a letter. The result should be the forename and the surname as a pair, e.g.,

```
# test uibk_mail "christian.sternagel@student.uibk.ac.at";;
```

should give the result `(("christian", "sternagel"),"")`. The functions of `Parser` may (and should) be used freely.

Exercise 8.2. Use the module `Parser` together with the type

```
type role = Employee | Student
```

to implement a function

```
mail : string -> (role * string * string)option
```

that parses a university e-mail address and returns the role, the first name, and the last name corresponding to the address, e.g.,

```
mail "christian.sternagel@uibk.ac.at"
 = Some(Employee,"Christian","Sternagel")
mail "some.student@student.uibk.ac.at"
 = Some(Student,"Some","Student")
mail "some.student@gmail.com"
 = None
```

Exercise 8.3. Extend the parser `nesting` from Example 8.5 such that it is successful if and only if the input corresponds to the grammar given in Example 8.4.

Exercise 8.4. Implement a parser `int : (int, char)Parser.t` for (decimal) integers where an integer is given by the (simplified) grammar

$$\begin{aligned} i &::= n \mid \texttt{+}n \mid \texttt{-}n \\ n &::= d^+ \\ d &::= \texttt{0} \mid \ldots \mid \texttt{9} \end{aligned}$$

Exercise 8.5. Using the parser `int : (int, char)Parser.t` of the previous exercise, implement a parser `int_list: (int list) t` that accepts any integer list where elements are enclosed in brackets (‘`[`’ and ‘`]`’) and separated by semicolons (‘`;`’). Your parser should drop any white spaces.

Exercise 8.6. Write a parser `words : (Strng.t list, char)Parser.t` that accepts arbitrarily many sentences and returns all the words that are contained as a list of l-strings. Here a sentence is a sequence of words (i.e., lowercase or uppercase letters) that are separated by white spaces and/or commas and terminated by a full stop (`.`), question mark (`?`), or exclamation mark (`!`).

Exercise 8.7. Implement a parser `btree: (ctree, char)Parser.t` that accepts the grammar

$$b ::= l \mid \texttt{(}\ b\ \texttt{,}\ l\ \texttt{,}\ b\ \texttt{)}$$
$$l ::= \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}$$

where the type `ctree` is the following:

```
type ctree = Leaf of char | Node of ctree * char * ctree
```

Exercise 8.8. Write a parser `tag : ((Strng.t * Strng.t), char) t` that accepts a simplified version of XML tags. For this purpose let a tag be of the form

$$\texttt{<}\textit{tagname}\texttt{>}\textit{content}\texttt{</}\textit{tagname}\texttt{>},$$

where *tagname* is an arbitrary (non empty) sequence of letters, *content* is an arbitrary sequence of characters except '`<`', and the first and second occurrence of *tagname* have to be identical. The result of the parser should be a pair of l-strings, where the first is the name of the tag and the second its content. E.g., `<a>bla</a>` should be accepted with result `(['a'], ['b'; 'l'; 'a'])`, whereas `<a>bla</A>` and `<a>bla</b>` should both fail.

Hint: You need not consider nested XML tags.

Exercise 8.9. Write a parser accepting only strings that are valid floating-point literals for OCaml.

Hint: See http://caml.inria.fr/pub/distrib/ocaml-3.10/ocaml-3.10-refman.pdf, page 92.

Exercise 8.10. Write a function

```
regexp : string -> (Strng.t, char)Parser.t}
```

that takes a string describing a regular expression and gives back a parser accepting any input that matches this expression. A regular expression e is built using the following (non left recursive) grammar:

$$e ::= \texttt{.}e' \mid x\ e' \mid \texttt{(}e\texttt{)}e'$$
$$e' ::= e\ e' \mid \texttt{*}e' \mid \epsilon$$
$$x ::= c \mid \texttt{\textbackslash}s$$
$$s ::= \texttt{.} \mid \texttt{(} \mid \texttt{)} \mid \texttt{*} \mid \texttt{\textbackslash}$$

where ϵ denotes the empty string, '`.`' stands for an arbitrary character, parentheses are used for grouping, '`*`' says that the previous pattern may match zero or more times, and c can be any character except '`.`', '`(`', '`)`', '`\`', and '`*`'. It should be possible to enter the above excluded characters by *escaping* them with a prefixed backslash. Hence '`\\`' is used to match a single backslash and '`\.`' to match a dot. E.g.,

```
test (regexp "(ab)*a") "ababac" = ['a';'b';'a';'b';'a']
test (regexp "\\..") ".z" = ['.';'z']
```

Hint: You have to write `"\\"` within a string to get a single backslash.

Exercise 8.11. Write a lexer `tokenize : (token list,char)Parser.t` (i.e., a parser working on `char`s and returning a list of tokens) for the following grammar:

$$\begin{aligned}
\langle \textit{rules} \rangle &\stackrel{\text{def}}{=} \langle \textit{rule} \rangle (\texttt{,} \langle \textit{rule} \rangle)^* \\
\langle \textit{rule} \rangle &\stackrel{\text{def}}{=} \langle \textit{string} \rangle \texttt{->} \langle \textit{string} \rangle \\
\langle \textit{string} \rangle &\stackrel{\text{def}}{=} \langle \textit{ident} \rangle (\sqcup \langle \textit{ident} \rangle)^* \mid \epsilon \\
\langle \textit{ident} \rangle &\stackrel{\text{def}}{=} \langle \textit{letter} \rangle^+ \\
\langle \textit{letter} \rangle &\stackrel{\text{def}}{=} \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{aligned}$$

Here ϵ denotes the empty string and the data type for tokens is:

```
type token = Ident of Strng.t | Arrow | Comma
```

Exercise 8.12. Write a parser `parse : (srs,token)Parser.t` (i.e., a parser working on `token`s and returning an abstract syntax tree), using the type `token` and the grammar of Exercise 8.11. The AST is represented by the data type `srs` (standing for *string rewrite system*):

```
type ident  = Strng.t list
type string = ident list
type rule   = (string * string)
type srs    = rule list
```

Exercise 8.13. Write a function

```
split_prefix : 'a list -> 'a list -> 'a list option
```

where `split_prefix l s` checks whether `l` is a prefix of `s`, and if so, returns `Some u` such that `l @ u` = `s`—otherwise `None` is returned.

Exercise 8.14. Write a function

```
split_sublist : 'a list -> 'a list -> ('a list * 'a list)option
```

where `split_sublist l s` checks whether `l` does occur somewhere inside `s`, and if so, returns `Some(u,v)` such that `u @ l @ v` = `s`—otherwise `None` is returned.

Exercise 8.15. Write a function `step : srs -> string -> string option` (here the type `string` refers to the type of Exercise 8.12) that for a given SRS $\mathcal{S}$ and a given string s, searches for a rule l `->` $r \in \mathcal{S}$ such that $s = ulv$ for some strings u and v, and returns as result urv, i.e., it replaces an occurrence of the left-hand side l in s by the corresponding right-hand side r. This function computes a so-called *$\mathcal{S}$-rewrite step* on the string s.

Exercise 8.16. Combine Exercise 8.11–Exercise 8.15 to an *interpreter* for SRSs, i.e., a command line program `srs` that takes two arguments, a file containing the textual description of an SRS (as given by the grammar above), as well as a string from where to start and applies `step` as often as possible printing the intermediate results. E.g., for a file `an.srs` containing

```
   a b  ->  b a,
   b b  ->  a,
 a a a  ->
```

and the starting string `a b a b` we get

```
> srs an.srs "a␣b␣a␣b"
a b a b
b a a b
b a b a
b b a a
a a a
[e]
```

Hint: The function `File.read : string -> Strng.t` (here `string` refers to OCaml strings) from the archive of week 9 may be useful to read the contents from a file.

Exercise 8.17. Write a parser `list: (Strng.t list,char)Parser.t` for lists, e.g.,

```
# test list "[Hello;World]"
 - : Strng.t list * char list = (["Hello"; "World"], "")
```

Test your parser on the following calls:

```
# test list "[Hello;World;]"
# test list "[Hello;World]blabla"
# test list "[Hello;;;;World]"
# test list "[[1]]"
```

Hint: Your parser need not give the desired output on the test cases.

Exercise 8.18. Consider the following grammar for propositional formulas:

$$\phi ::= p \mid (!\ \phi) \mid (\phi\ \&\ \phi)$$

a) Write a `lexer` for this grammar such that e.g.

```
# Parser.test lexer "(a & (!a) )";;
- : token list * char list =
([LPAR; ID "a"; AND; LPAR; NOT; ID "a"; RPAR; RPAR], "")
```

and

```
# Parser.test lexer "(a a)";;
- : token list * char list =
([LPAR; ID "a"; ID "ab"; RPAR], "")
```

b) Write a `parser` for this grammar such that e.g.

```
# Parser.parse parser [LPAR; ID "a"; AND; LPAR; NOT; ID "a"; RPAR; RPAR];;
- : t option = Some (And (Atom "a", Not (Atom "a")))
```

and

```
# Parser.parse parser [LPAR; ID "a"; AND; LPAR; NOT; ID "a"; RPAR; RPAR];;
- : t option = Some (And (Atom "a", Not (Atom "a")))
```

c) Give a non-left recursive grammar such that ! binds stronger than &.

Exercise 8.19. Consider the parsers `digit : (char,char) t` and `digit_int : (int,char) t`. The calls

```
# test digit "1234";;
- : char * char list = ('1', "234")
# test int_digit "1234";;
- : int * char list = (1, "234")
```

show the difference of the two functions.

a) Implement `digit_int` with the help of `digit`.

b) Implement `digit_int` without using `digit`.

Hint: For item b) the function `token` is useful.

Exercise 8.20. For each of the following parsers explain why the call

`test (count_spaces 0) "␣␣hello␣world␣"`

terminates or not.

a)

```
count_spaces i =
      (eoi >> return i)
  <|> (noneof "␣\t\r" >>= fun _ -> count_spaces i)
  <|> (spaces >>= fun j -> count_spaces (i+j))
```

b)

```
let rec count_spaces i =
      (eoi >> return i)
  <|> (spaces >>= fun j -> count_spaces (i+j))
  <|> (noneof "␣\t\r" >>= fun _ -> count_spaces i)
```

c)

```
let rec count_spaces i =
      (eoi >> return i)
  <|> (noneof "␣\t\r" >> count_spaces i)
  <|> (spaces >>= fun j -> count_spaces (i+j))
```

d)

```
let rec count_spaces i =
      (eoi >>= fun _ -> return i)
  <|> (noneof "␣\t\r" >> count_spaces i)
  <|> (spaces >>= fun j -> count_spaces (i+j))
```

e)

```
let rec count_spaces i =
      (noneof "␣\t\r" >> count_spaces i)
  <|> (spaces >>= fun j -> count_spaces (i+j))
  <|> (eoi >>= fun _ -> return i)
```

9. Types

There are two important tasks concerning types in a functional language: type checking and type inference.

The former is the process of verifying given constraints on types and may either occur at compile-time (static type checking) or at run-time (dynamic type checking). The advantage of static type checking is that no type information has to be stored after the compilation process, since type correctness has already been proved. OCaml uses static type checking, hence this is also what is discussed in the following.

The latter is the process of computing a (most general) type for a given expression. A language where types are inferred automatically (like OCaml) makes some programming tasks easier. E.g., types of variables need not to be declared explicitly. But still type safety is maintained.[1]

Before giving the details of type checking and type inference, some typed language is needed. Two obvious choices would be λ-calculus extended by types (also called *simply typed lambda-calculus*) and OCaml itself. Since the former is inconvenient to use and the latter is more complex than necessary, a mixture of both is considered.

9.1. Core ML

The combination of the λ-calculus and OCaml used to demonstrate type checking and inference is called *core ML*. Its expressions (e) are defined by the following BNF grammar

$$\begin{array}{lll} e ::= x \mid e\ e \mid \lambda x.e & & (\lambda\text{-calculus}) \\ \quad \mid c & & (\text{for primitives}) \\ \quad \mid \textbf{let}\ x = e\ \textbf{in}\ e & & (\text{let binding}) \\ \quad \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & & (\text{conditional}) \end{array}$$

where x is a *variable* and c a *constant* denoting any of the primitives true, false, $+$, $-$, $\times$, $>$, $<$, $=$, ...

9.2. Type Checking

Before introducing how to check types some formal definitions are needed. In the following a type τ is of the form

$$\tau ::= \alpha \mid \tau \to \tau \mid g(\tau, \ldots, \tau)$$

where α is a *type variable*, '$\to$' is the *arrow type* constructor (in the end just a special case of the following construct), and g an arbitrary type constructor (e.g., for lists or tuples). Every type constructor has a fixed *arity*, i.e., number of arguments it takes. E.g., list is unary, hence applications of the type constructor for lists are of the form $\mathsf{list}(\alpha)$, $\mathsf{list}(\mathsf{int})$, $\mathsf{list}(\mathsf{bool})$, etc. Note that the base types (bool, int, ...) are just a special case of type constructors, namely those of arity 0 (i.e., without arguments). Instead of

[1] A program is *type safe* if a certain class of errors—namely *type errors*—is prevented by the compiler. An example of a type error would be the application of a list length function to an integer.

$$\frac{e : \tau \in E}{E \vdash e : \tau}\ (\mathsf{ref}) \qquad \frac{E \vdash e_1 : \tau_2 \to \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash e_1\ e_2 : \tau_1}\ (\mathsf{app})$$

$$\frac{E, x : \tau_1 \vdash e : \tau_2}{E \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\mathsf{abs}) \qquad \frac{E \vdash e_1 : \tau_1 \quad E, x : \tau_1 \vdash e_2 : \tau_2}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ (\mathsf{let})$$

$$\frac{E \vdash e_1 : \mathsf{bool} \quad E \vdash e_2 : \tau \quad E \vdash e_3 : \tau}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ (\mathsf{ite}).$$

Table 9.1.: The inference system $\mathcal{C}$ for type checking.

$\mathsf{bool}()$ or $\mathsf{int}()$, as indicated by the BNF grammar, such *nullary* type constructors are written without (), i.e., bool, int.

A *typing environment* is a set of pairs, mapping variables and constants to types. Instead of (e, τ) these pairs are written $e : \tau$, denoting "e *has type* τ". E.g., the typing environment where the variable x is of type bool and the variable y of type $\mathsf{list}(\alpha)$, is written as

$$\{x : \mathsf{bool}, y : \mathsf{list}(\alpha)\}.$$

In the following let

$$P = \{\mathsf{true} : \mathsf{bool}, \mathsf{false} : \mathsf{bool}, + : \mathsf{int} \to \mathsf{int} \to \mathsf{int}, 0 : \mathsf{int}, 1 : \mathsf{int}, \dots\}$$

denote the *primitive typing environment* (containing type information for every primitive constant).

The *domain* of a typing environment E is defined by

$$\mathcal{D}\mathsf{om}(E) = \{e \mid (e : \tau) \in E\}.$$

It consists of all variables or constants that have a type assigned in the environment E.

A *typing judgment* is written $E \vdash_{\mathcal{C}} e : \tau$ for some typing environment E, core ML expression e, type τ and the type checking system $\mathcal{C}$. Such a typing judgment reads: "From the typing environment E the type τ can be derived for the expression e using the type checking system $\mathcal{C}$." The system $\mathcal{C}$ is given by the rules of Table 9.1. Usually it is clear from the context that the system $\mathcal{C}$ is used. Hence in the following $E \vdash e : \tau$ is written instead of the longer form from above.

In such a system a single rule is called an *inference rule*. The part of an inference rule above the line consists of some so called *premises*. An inference rule where the premises do not contain new type judgements but just a condition when the rule can be applied is called axiom.

Hence type checking of a typing judgment $E \vdash e : \tau$ corresponds to building a tree (this tree is called a *proof*), where the root is the judgment, branching denotes applications of inference rules from $\mathcal{C}$ and the leaves are applications of ref (since this rule is the only axiom).

When constructing proof trees we will pursue a goal-directed approach, i.e., start with the typing judgement under consideration and construct the proof from the root to the leaves. Consequently we apply the inference rules from bottom to top. Stated differently in order to prove a type judgement we apply an inference rule and prove its premises.

Now, let us have a closer look at the different inference rules where $E, e : \tau$ abbreviates $E \cup \{e : \tau\}$.

(ref) The *reflexivity* rule states that we can prove the typing judgement $E \vdash e : \tau$ if $e : \tau$ is contained in the type environment E.

(app) The *application* rule states that in order to prove that the type τ_1 can be derived for the application $e_1\ e_2$ we have to show that it is the case that (1) the type $\tau_2 \to \tau_1$ can be derived for e_1 and (2) the type τ_2 can be derived for e_2 (both from the same environment E). This rule captures the intuition that function application does only make sense for functions, i.e., expressions of some *arrow type* $\tau \to \tau'$, and further, the argument of a function has to have the correct type.

(abs) The *abstraction* rule states that a function $\lambda x.e$ has type $\tau_1 \to \tau_2$ if (1) the variable x has type τ_1 and the function body consists of an expression e of type τ_2.

(let) The *let(-binding)* rule states that the expression **let** $x = e_1$ **in** e_2 has type τ_2 if (1) the type τ_1 can be derived for e_1 and (2) assuming type τ_1 for a variable x, the type τ_2 can be derived for e_2.

(ite) The *if-then-else* rule captures the intuition that the conditional expression e_1 of **if** e_1 **then** e_2 **else** e_3 has to be of type bool and the *then*-branch as well as the *else*-branch have to be of the same type.

From the above description we see that rules (app) and (let) require more thoughts to be applied since the type τ_2 of the argument must be *guessed* in rule (app) and the same holds for the type τ_1 in rule (let).

Example 9.1. Consider the typing environment $E = \{\mathsf{true} : \mathsf{bool}, + : \mathsf{int} \to \mathsf{int} \to \mathsf{int}\}$. Then the judgment $E \vdash (\lambda x.x)\ \mathsf{true} : \mathsf{bool}$ can be proved by

$$\dfrac{\dfrac{\dfrac{}{E, x : \mathsf{bool} \vdash x : \mathsf{bool}}\,(\mathsf{ref})}{E \vdash \lambda x.x : \mathsf{bool} \to \mathsf{bool}}\,(\mathsf{abs}) \qquad \dfrac{}{E \vdash \mathsf{true} : \mathsf{bool}}\,(\mathsf{ref})}{E \vdash (\lambda x.x)\ \mathsf{true} : \mathsf{bool}}\,(\mathsf{app})$$

and the judgment $E \vdash \lambda x.x + x : \mathsf{int} \to \mathsf{int}$ by

$$\dfrac{\dfrac{\star \qquad \dfrac{}{E, x : \mathsf{int} \vdash x : \mathsf{int}}\,(\mathsf{ref})}{E, x : \mathsf{int} \vdash x + x : \mathsf{int}}\,(\mathsf{app})}{E \vdash \lambda x.x + x : \mathsf{int} \to \mathsf{int}.}\,(\mathsf{abs})$$

where $\star$ is

$$\dfrac{\dfrac{}{E, x : \mathsf{int} \vdash + : \mathsf{int} \to \mathsf{int} \to \mathsf{int}}\,(\mathsf{ref}) \qquad \dfrac{}{E, x : \mathsf{int} \vdash x : \mathsf{int}}\,(\mathsf{ref})}{E, x : \mathsf{int} \vdash (+)\ x : \mathsf{int} \to \mathsf{int}}\,(\mathsf{app})$$

In the second example the '+' is used infix. This is just for convenience. By the grammar for core ML expressions it would be prefix, which is used as in OCaml, i.e., $(+)\ x\ y$ instead of $x + y$.

9.3. Type Inference

Inferring the most general type of a given expression is known as *type inference*. It is a bit more complicated than type checking. Hence some further definitions are needed.

A *type substitution* σ is very similar to a substitution for λ-terms (see Chapter 5). Here the mapping is from the set of type variables (α, β, γ, ...) into the set of types. The application of a substitution σ to a type τ (written as $\tau\sigma$) is defined by

$$\tau\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \tau = \alpha \\ \tau_1\sigma \to \tau_2\sigma & \text{if } \tau = \tau_1 \to \tau_2 \\ g(\tau_1\sigma, \ldots, \tau_n\sigma) & \text{if } \tau = g(\tau_1, \ldots, \tau_n). \end{cases}$$

A type substitution can also be applied to a typing environment. This is defined by

$$E\sigma \stackrel{\text{def}}{=} \{e : \tau\sigma \mid e : \tau \in E\}.$$

The application of a type substitution to another type substitution is their functional composition

$$\sigma_1\sigma_2 \stackrel{\text{def}}{=} \sigma_2 \circ \sigma_1.$$

Note: Recall that $\sigma_2 \circ \sigma_1$ is the function defined by $x \mapsto \sigma_2(\sigma_1(x))$.

The set of *type variables* of a type τ is given by

$$\mathcal{TV}\mathsf{ar}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TV}\mathsf{ar}(\tau_1) \cup \mathcal{TV}\mathsf{ar}(\tau_2) & \text{if } \tau = \tau_1 \to \tau_2 \\ \bigcup_{1 \le i \le n} \mathcal{TV}\mathsf{ar}(\tau_i) & \text{if } \tau = g(\tau_1, \ldots, \tau_n). \end{cases}$$

The next example familiarizes the reader with type substitutions.

Example 9.2. Consider the type τ and the type substitutions σ and σ_2:

$$\begin{aligned} \tau &= \alpha \to (\alpha_1 \to \alpha_3) \\ \sigma &= \{\alpha/\mathsf{int} \to \mathsf{int}, \alpha_1/\mathsf{list}(\alpha_2)\} \\ \sigma_2 &= \{\alpha_3/\alpha_4, \alpha_2/\alpha, \alpha/\alpha_1\} \end{aligned}$$

Then we have

$$\begin{aligned} \tau\sigma &= (\mathsf{int} \to \mathsf{int}) \to (\mathsf{list}(\alpha_2) \to \alpha_3) \\ \mathcal{TV}\mathsf{ar}(\tau) &= \{\alpha, \alpha_1, \alpha_3\} \\ \mathcal{TV}\mathsf{ar}(\tau\sigma) &= \{\alpha_2, \alpha_3\} \\ \sigma\sigma_2 &= \{\alpha/\mathsf{int} \to \mathsf{int}, \alpha_1/\mathsf{list}(\alpha), \alpha_3/\alpha_4, \alpha_2/\alpha\} \end{aligned}$$

9.3.1. Unification Problems

A *unification problem* is represented by a (finite) sequence of equations between types $\tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'$. Here, an empty sequence is represented by $\square$. *Unification* is the process of finding a substitution σ such that the types in each equation become syntactically identical, i.e., $\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$. Such a substitution is then called *a solution to the unification problem* or just a *unifier*. If a unification problem admits a solution then it can be found by arbitrary applications of rules from the inference system $\mathcal{U}$ (see Table 9.2).

In contrast to the inference rules of the system $\mathcal{C}$ we read the rules from system $\mathcal{U}$ top to bottom.

(d) The *decomposition* rules capture the facts that (d_1) two applications of type constructors are equal if and only if the type constructors are equal *and* their respective parameters are unifiable, and (d_2) two arrow types are equal if and only if their respective components are unifiable.

(v) The *variable* rules state that as soon as either the lhs (for v_1) or the rhs (for v_2) of an equation is a type variable α, the extracted information can be used (in form of a substitution) to refine the remaining problem, but only if the type variable does not occur in the other side of the equation. This is called the *occur*-check.

(t) The *trivial equations removal* rule does exactly that, it removes trivial equations, i.e., equations where the lhs is the same as the rhs.

$$\frac{E_1; g(\tau_1, \ldots, \tau_n) \approx g(\tau_1', \ldots, \tau_n'); E_2}{E_1; \tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'; E_2} \ (\mathsf{d}_1)$$

$$\frac{E_1; \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'; E_2}{E_1; \tau_1 \approx \tau_1'; \tau_2 \approx \tau_2'; E_2} \ (\mathsf{d}_2)$$

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \ (\mathsf{v}_1)$$

$$\frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \ (\mathsf{v}_2)$$

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \ (\mathsf{t})$$

Table 9.2.: The inference system $\mathcal{U}$ for unification.

Also for constructing proofs we apply the rules from system $\mathcal{U}$ top to bottom. Since rules v_1 and v_2 modify all equations we do not depict unification proofs as trees but rather as sequences. If E is the premise and E' the conclusion of the above inference rule r (where $r \in \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{v}_1, \mathsf{v}_2, \mathsf{t}\}$), the application of r is written as $E \Rightarrow_\sigma^{(r)} E'$, where σ indicates a substitution (for $r \in \{\mathsf{d}_1, \mathsf{d}_2, \mathsf{t}\}$ the substitution ι, i.e., the *empty substitution*, with $\iota(\alpha) = \alpha$ for all type variables α, is used). To solve a given unification problem E_1 the inference rules are applied repeatedly. The inference rules are designed such that this process stops after finitely many, say n, steps:

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n.$$

If $E_n = \square$ then E_1 has the solution $\sigma = \sigma_1\sigma_2\cdots\sigma_{n-1}$.[2] If $E_n \neq \square$ then E_1 does not have a solution.

Example 9.3. The types $\mathsf{list}(\mathsf{bool})$ and $\mathsf{list}(\alpha)$ are unifiable as can be seen by the derivation

$$\begin{array}{lll} \mathsf{list}(\mathsf{bool}) \approx \mathsf{list}(\alpha) & \Rightarrow_{\iota}^{(\mathsf{d}_1)} & \mathsf{bool} \approx \alpha \\ & \Rightarrow_{\{\alpha/\mathsf{bool}\}}^{(\mathsf{v}_2)} & \square. \end{array}$$

The unifier is $\{\alpha/\mathsf{bool}\}$.

9.3.2. Typing Constraints

A type inference problem is given by $E \rhd_{\mathcal{I}}^{\mathcal{U}} e : \alpha$. This reads: "Transform the given problem into a unification problem using the system $\mathcal{I}$. Afterwards solve the resulting unification problem (if possible) using the system $\mathcal{U}$." The corresponding result is a substitution σ such that $E\sigma \vdash e : \alpha\sigma$.

Before unification can be used to implement type inference, a translation from type inference problems to unification problems is needed. This can be done using the inference rules of $\mathcal{I}$ (which should be easily understandable since they are very close to those of $\mathcal{C}$) to generate typing constraints. The rules of $\mathcal{I}$ can be found in Table 9.3. Usually $\mathcal{U}$ as well as $\mathcal{I}$ are omitted from $E \rhd_{\mathcal{I}}^{\mathcal{U}} e : \alpha$, resulting in $E \rhd e : \alpha$.

[2] The order of applying the inference rules from $\mathcal{U}$ to equations in E may have an effect on the unifier. However, every unifier σ computed by system $\mathcal{U}$ is *most general*. This means that any other unifier τ can be obtained from σ. Formally this means that there exists a substitution μ such that $\tau = \sigma\mu$.

$$\frac{E, e : \tau_0 \rhd e : \tau_1}{\tau_0 \approx \tau_1}\ (\mathsf{con}) \qquad \frac{E \rhd e_1\ e_2 : \tau}{E \rhd e_1 : \alpha \to \tau; E \rhd e_2 : \alpha}\ (\mathsf{app})$$

$$\frac{E \rhd \lambda x.e : \tau}{E, x : \alpha_1 \rhd e : \alpha_2; \tau \approx \alpha_1 \to \alpha_2}\ (\mathsf{abs}) \qquad \frac{E \rhd \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}{E \rhd e_1 : \alpha; E, x : \alpha \rhd e_2 : \tau}\ (\mathsf{let})$$

$$\frac{E \rhd \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}{E \rhd e_1 : \mathsf{bool}; E \rhd e_2 : \tau; E \rhd e_3 : \tau}\ (\mathsf{ite})$$

Table 9.3.: The inference system $\mathcal{I}$ for type inference.

Note that in the rules of $\mathcal{I}$, the type variables α, α_1 and α_2 are required to be *fresh*, i.e., they do not occur in the preceding derivations.

To solve the type inference problem $E \rhd e : \alpha$ we apply the rules of system $\mathcal{I}$ from top to bottom, resulting in a proof sequence. If this proof sequence stops before having a unification problem—none of the rules is applicable but some type inference constraints are still left—then statement e cannot be typed w.r.t. the type environment E. Otherwise at some point the given type inference problem is translated into a unification problem. If the resulting unification problem has a solution, this represents the most general type of the original type inference problem, otherwise the original type inference problem is *not typable* (w.r.t. $\mathcal{I}$ and $\mathcal{U}$).

Example 9.4. Consider the primitive environment P as defined above and application of the identity function as given by $\mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ 1$. The resulting type inference problem is

$$P \rhd \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ 1 : \alpha_0$$

where α_0 is a fresh type variable. Using $\mathcal{I}$ this is transformed into the unification problem:

$$\underline{P \rhd \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ 1 : \alpha_0}$$
$$\overset{\mathsf{let}}{\Rightarrow}$$
$$\underline{P \rhd \lambda x.x : \alpha_1}; P, id : \alpha_1 \rhd id\ 1 : \alpha_0$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{P, x : \alpha_2 \rhd x : \alpha_3}; \alpha_1 \approx \alpha_2 \to \alpha_3; P, id : \alpha_1 \rhd id\ 1 : \alpha_0$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \underline{P, id : \alpha_1 \rhd id\ 1 : \alpha_0}$$
$$\overset{\mathsf{app}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \underline{P, id : \alpha_1 \rhd id : \alpha_4 \to \alpha_0}; P, id : \alpha_1 \rhd 1 : \alpha_4$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \underline{P, id : \alpha_1 \rhd 1 : \alpha_4}$$
$$\overset{\mathsf{con}}{\Rightarrow}$$
$$\alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{int} \approx \alpha_4.$$

Afterwards $\mathcal{U}$ is used to get a solution:

$$\begin{array}{ll}
 & \alpha_2 \approx \alpha_3; \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_2/\alpha_3\}} & \alpha_1 \approx \alpha_3 \to \alpha_3; \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_1/\alpha_3 \to \alpha_3\}} & \alpha_3 \to \alpha_3 \approx \alpha_4 \to \alpha_0; \mathsf{int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{d}_2)}_{\iota} & \alpha_3 \approx \alpha_4; \alpha_3 \approx \alpha_0; \mathsf{int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_3/\alpha_4\}} & \alpha_4 \approx \alpha_0; \mathsf{int} \approx \alpha_4 \\
\Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_4/\alpha_0\}} & \mathsf{int} \approx \alpha_0 \\
\Rightarrow^{(\mathsf{v}_2)}_{\{\alpha_0/\mathsf{int}\}} & \square.
\end{array}$$

The resulting unifier is

$$\sigma = \{\alpha_0/\mathsf{int}, \alpha_1/\mathsf{int} \to \mathsf{int}, \alpha_2/\mathsf{int}, \alpha_3/\mathsf{int}, \alpha_4/\mathsf{int}\}.$$

Since the type variable α_0 was used for the initial type inference problem and $\sigma(\alpha_0) = \mathsf{int}$, the most general type for **let** $id = \lambda x.x$ **in** $id\ 1$ is int.

Note: To compute the unifier σ we look how the type variables change by applying all substitutions appearing in the unification proof sequence. For α_0 this is easy since α_0 is mapped to int. For α_1 this is a bit more involved since α_1 is first mapped to $\alpha_3 \to \alpha_3$, but afterwards α_3 is mapped to α_4 and finally α_4 is mapped to int. Hence $\sigma(\alpha_1) = \mathsf{int} \to \mathsf{int}$.

Example 9.5. As a second example consider the expression $\lambda x.x\ x$ and the primitive environment P. The resulting type inference problem is

$$P \rhd \lambda x.x\ x : \alpha_0$$

where α_0 is a fresh type variable. This is transformed into the unification problem:

$$\frac{\underline{P \rhd \lambda x.x\ x : \alpha_0}}{\overset{\mathsf{abs}}{\Rightarrow}}$$

$$\frac{\underline{P, x : \alpha_1 \rhd x\ x : \alpha_2}; \alpha_0 \approx \alpha_1 \to \alpha_2}{\overset{\mathsf{app}}{\Rightarrow}}$$

$$\frac{\underline{P, x : \alpha_1 \rhd x : \alpha_3 \to \alpha_2}; P, x : \alpha_1 \rhd x : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2}{\overset{\mathsf{con}}{\Rightarrow}}$$

$$\frac{\alpha_1 \approx \alpha_3 \to \alpha_2; \underline{P, x : \alpha_1 \rhd x : \alpha_3}; \alpha_0 \approx \alpha_1 \to \alpha_2}{\overset{\mathsf{con}}{\Rightarrow}}$$

$$\alpha_1 \approx \alpha_3 \to \alpha_2; \alpha_1 \approx \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

Afterwards use $\mathcal{U}$ as follows:

$$\begin{array}{ll}
\alpha_1 \approx \alpha_3 \to \alpha_2; \alpha_1 \approx \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2 & \Rightarrow^{(\mathsf{v}_1)}_{\{\alpha_1/\alpha_3 \to \alpha_2\}} \\
\alpha_3 \to \alpha_2 \approx \alpha_3; \alpha_0 \approx (\alpha_3 \to \alpha_2) \to \alpha_2 &
\end{array}$$

where after the first step the occur-check fails and hence the given unification problem is not unifiable. This means that $\lambda x.x\ x$ is not typable.

Example 9.6. As a further example consider the Y-combinator and the empty environment $\varnothing$. The type inference problem $\varnothing \rhd \mathsf{Y} : \alpha_0$ is transformed into a unification problem as follows:

$$\underline{\varnothing \rhd \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) : \alpha_0}$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{f : \alpha_1 \rhd (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) : \alpha_2}; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow}$$
$$\underline{f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3 \to \alpha_2}; f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{abs}}{\Rightarrow}$$
$$\underline{\{f : \alpha_1, x : \alpha_4\} \rhd f\ (x\ x) : \alpha_5}; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\underline{\{f : \alpha_1, x : \alpha_4\} \rhd f : \alpha_6 \to \alpha_5}; \{f : \alpha_1, x : \alpha_4\} \rhd x\ x : \alpha_6;$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \underline{\{f : \alpha_1, x : \alpha_4\} \rhd x\ x : \alpha_6;}$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \underline{\{f : \alpha_1, x : \alpha_4\} \rhd x : \alpha_7 \to \alpha_6}; \{f : \alpha_1, x : \alpha_4\} \rhd x : \alpha_7;$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \underline{\{f : \alpha_1, x : \alpha_4\} \rhd x : \alpha_7};$$
$$\alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5; f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{f : \alpha_1 \rhd \lambda x.f\ (x\ x) : \alpha_3}; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{abs}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{\{f : \alpha_1, x : \alpha_8\} \rhd f\ (x\ x) : \alpha_9}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{app}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\underline{\{f : \alpha_1, x : \alpha_8\} \rhd f : \alpha_{10} \to \alpha_9}; \{f : \alpha_1, x : \alpha_8\} \rhd x\ x : \alpha_{10}; \alpha_3 \approx \alpha_8 \to \alpha_9;$$
$$\alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{con}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \underline{\{f : \alpha_1, x : \alpha_8\} \rhd x\ x : \alpha_{10}}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$
$$\overset{\mathsf{app}}{\Rightarrow} \alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \underline{\{f : \alpha_1, x : \alpha_8\} \rhd x : \alpha_{11} \to \alpha_{10}};$$
$$\{f : \alpha_1, x : \alpha_8\} \rhd x : \alpha_{11}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{cons}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \alpha_8 \approx \alpha_{11} \to \alpha_{10}; \underline{\{f : \alpha_1, x : \alpha_8\} \rhd x : \alpha_{11}};$$
$$\alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

$$\overset{\mathsf{cons}}{\Rightarrow}$$

$$\alpha_1 \approx \alpha_6 \to \alpha_5; \alpha_4 \approx \alpha_7 \to \alpha_6; \alpha_4 \approx \alpha_7; \alpha_3 \to \alpha_2 \approx \alpha_4 \to \alpha_5;$$
$$\alpha_1 \approx \alpha_{10} \to \alpha_9; \alpha_8 \approx \alpha_{11} \to \alpha_{10};$$
$$\alpha_8 \approx \alpha_{11}; \alpha_3 \approx \alpha_8 \to \alpha_9; \alpha_0 \approx \alpha_1 \to \alpha_2$$

It is left as an exercise to show that this unification problem is not solvable (cf. Exercise 9.4). Hence Y is not typable.

9.4. Recursion

An interesting result for the simply typed lambda calculus (that will not be proved in this course, however), is that every typable λ-term is guaranteed to terminate. This is also true for core ML. As has been seen in the last section, Y is not typable. That is, of course, due to the mentioned result. But if Y is not typable, there is no possibility to define recursive functions (since all other thinkable fixed point combinators are also not typable). The trick is, to include Y as a primitive constant and just assign a type that suffices to make applications of Y well-typed.

The idea of the fixed point combinator was that given some function t (expecting itself as first argument), it replicates this function and applies it to the computed fixed point $\mathsf{Y}\ t$, i.e., $t\ (Y\ t)$.

Example 9.7. Again, consider the function length. As a first approach for its implementation consider

$$\mathsf{length} \stackrel{\text{def}}{=} \lambda x.\mathbf{if}\ \mathit{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + \mathsf{length}\ (\mathit{tl}\ x)$$

where $\mathit{null} : \mathsf{list}(\alpha) \to \mathsf{bool}$, $\mathit{tl} : \mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$, $0 : \mathsf{int}$, $1 : \mathsf{int}$, and $+ : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ are constants contained in P. As already mentioned for the corresponding λ-term, this definition is not well-defined, due to the recursive reference to length.

Again the problem is solved by introducing an additional argument and applying Y to the resulting expression, i.e.,

$$\mathsf{length} \stackrel{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\mathbf{if}\ \mathit{null}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + f\ (\mathit{tl}\ x)).$$

It can be seen that Y expects some function (i.e., an expression having an arrow type) as its argument. The function that is supplied to Y in turn expects another function (namely the one that is about to be defined). This process should be generally applicable, hence Y does need some not too restricted type. The result after some investigation will be similar to

$$\mathsf{Y} : (\alpha \to \alpha) \to \alpha.$$

Now let P_μ denote the primitive typing environment P and the type assignment for Y, i.e.,

$$P_\mu \stackrel{\text{def}}{=} P \cup \{\mathsf{Y} : (\alpha \to \alpha) \to \alpha\}.$$

9.5. Chapter Notes

The type inference algorithm presented in this chapter follows the *Hindley-Milner type inference* algorithm for the simply typed lambda calculus. It was first presented by Hindley [6] and independently conceived by Milner [11]. Sometimes it is also referred to as *Algorithm W*.

9.6. Exercises

Exercise 9.1. Prove the judgment $E \vdash \mathbf{if}\ \mathsf{true}\ \mathbf{then}\ 1\ \mathbf{else}\ 0 : \mathsf{int}$ for the typing environment

$$E = \{\mathsf{true} : \mathsf{bool}, 0 : \mathsf{int}, 1 : \mathsf{int}\}.$$

Exercise 9.2. Prove the judgment $E \vdash e : \mathsf{int}$ for the typing environment

$$\begin{aligned} E = \{ & \\ & \mathsf{pair} : \mathsf{int} \to \mathsf{int} \to \mathsf{pair}(\mathsf{int}, \mathsf{int}), \\ & \mathsf{fst} : \mathsf{pair}(\mathsf{int}, \mathsf{int}) \to \mathsf{int}, \\ & 1 : \mathsf{int}, \\ & 3 : \mathsf{int} \\ \} & \end{aligned}$$

and the core ML expression

$$e \equiv \textbf{let } f = \lambda p.\mathsf{fst}\ p \textbf{ in } f\ (\mathsf{pair}\ 1\ 3).$$

Exercise 9.3. Transform the type inference problem $E \rhd \mathsf{hd}\ (\mathsf{cons}\ 1\ \mathsf{nil}) : \alpha_0$ with the typing environment

$$\begin{aligned} E = \{ & \\ & \mathsf{hd} : \mathsf{list}(\beta_1) \to \beta_1, \\ & \mathsf{cons} : \beta_2 \to \mathsf{list}(\beta_2) \to \mathsf{list}(\beta_2), \\ & \mathsf{nil} : \mathsf{list}(\beta_3), \\ & 1 : \mathsf{int} \\ \} & \end{aligned}$$

into a unification problem.

Exercise 9.4. Show that the unification problem given by the list of equations

$$\begin{aligned} \alpha_1 &\approx \alpha_6 \to \alpha_5; \\ \alpha_4 &\approx \alpha_7 \to \alpha_6; \\ \alpha_4 &\approx \alpha_7; \\ \alpha_3 \to \alpha_2 &\approx \alpha_4 \to \alpha_5; \\ \alpha_1 &\approx \alpha_{10} \to \alpha_9; \\ \alpha_8 &\approx \alpha_{11} \to \alpha_{10}; \\ \alpha_8 &\approx \alpha_{11}; \\ \alpha_3 &\approx \alpha_8 \to \alpha_9; \\ \alpha_0 &\approx \alpha_1 \to \alpha_2 \end{aligned}$$

is unsolvable.

Exercise 9.5. Solve the unification problem resulting from Exercise 9.3 and give the unifier.

Exercise 9.6. Solve the type inference problem $P_\mu \rhd e : \alpha_0$ for the core ML expression

$$e \equiv \textbf{let } o = \mathsf{Y}\ (\lambda f.\mathsf{cons}\ 1\ f) \textbf{ in } o.$$

Exercise 9.7. Solve the type inference problem $E \rhd e : \alpha_0$ for the core ML expression

$$e \equiv \textbf{let } x = \mathsf{cons}\ 0\ \mathsf{nil} \textbf{ in } \mathsf{cons}\ \mathsf{true}\ \mathsf{nil}.$$

and the environment

$$E = \{\mathsf{nil} : \mathsf{list}(\beta_1), \mathsf{cons} : \beta_2 \to \mathsf{list}(\beta_2) \to \mathsf{list}(\beta_2), 0 : \mathsf{int}, \mathsf{true} : \mathsf{bool}\}$$

Can you explain the (surprising) result?

10. Lazyness

As discussed in Chapter 5, in principle any OCaml program is equivalent to some λ-term. Further, computing the result of such a program amounts to the same as β-reducing the corresponding λ-term to some kind of normal form. However, in general the order in which to contract redexes is not unique. Hence a certain strategy is used that uniquely determines the redex at each reduction step.

Two well known evaluation strategies are call-by-value (or strict/eager evaluation) and call-by-name (or lazy evaluation). Whereas OCaml adopts eager evaluation (by default), Haskell [1] would be an example of a *lazy* language.

One nice application of lazy evaluation is the usage of infinite data structures (e.g., infinite lists). The advantage compared to strict evaluation is that data can be separated from control. In the following we give two implementations of lazy lists and some examples of programs that use them.

10.1. Motivation

Recall the built-in cons operator '`(::)`' of OCaml. Since OCaml adopts eager evaluation, in an expression like e_1 `::` e_2, the arguments e_1 and e_2 are evaluated—to values x and xs say—before the resulting list x `::` xs is constructed and returned.[1] Now if e_2 contains some expensive computations (or even not terminate at all), then e_1 `::` e_2 will take at least as long to compute as e_2; even if the only place in the remaining program referring to e_1 `::` e_2 is `hd`$(e_1$ `::` $e_2)$, i.e., the value of e_2 is not used.

The idea behind lazy evaluation is that only those parts of a program that are really needed to compute the final result, should be evaluated at all. Hence, in the above example only e_1 should be reduced. This is enough to give the result of `hd`$(e_1$ `::` $e_2)$, namely x. In this case we will even get the result, if e_2 does not terminate at all, since e_2 is never reduced.

In the following we give two possible implementations of lists that only reduce their tails as soon as they are needed. Usually, such lists are called *lazy lists*.

10.2. Custom Lazy Lists

Before we give some examples on how to use lazy lists, we need an appropriate type. Let us start with a type that corresponds to the standard list type and refine it step by step.

```
type 'a llist = Nil | Cons of ('a * 'a llist)
```

Next we want to have some expression that has an `llist` type but does not terminate. Consider for example

```
let rec from n = Cons(n,from(n+1))
```

which, given an integer n, constructs the (infinite) list of all integers starting with n. A call to `from` does not terminate, as can be seen by the example:

[1] Note that OCaml evaluates parameters in functions from right to left. Hence first e_2 is evaluated to xs, then e_1 is evaluated to x, and finally the list x `::` xs is constructed.

```
# from 0;;
Stack overflow during evaluation (looping recursion?).
```

Consider a *head* function for our custom `llist` type.

```
let hd xs = match xs with Nil       -> failwith "empty list"
                        | Cons(x,_) -> x
```

Also if the call to `from` is the argument of a call to `hd`, there is no result, e.g.,

```
# hd(from 0);;
Stack overflow during evaluation (looping recursion?).
```

The desired behavior is that `hd(from 0)` should result in `0`. Hence, we have to modify our list type, such that the second argument of a `Cons`-cell is only evaluated if explicitly requested to do so. A common trick to achieve this, is to use a function, taking one (dummy) argument of type `unit`. The body of this function is only evaluated, if `()` is supplied. The first attempt to integrate this trick in our list type could be:

```
type 'a llist = Nil | Cons of ('a * (unit -> 'a llist))
```

The next example familiarizes the reader with some values of this new type of lazy lists.

Example 10.1. In the following table the left column gives some values for the type `'a llist`. The right column shows which list is represented by the respective value in the left column.

`Nil`	`[]`
`Cons(1, fun () -> Nil)`	`[1]`
`Cons(2, fun () -> Cons(1,fun () -> Nil))`	`[2;1]`

Note that we also have to redefine `from` in order to obtain the correct type.

```
let rec from n = Cons(n,fun() -> from(n+1))
```

Now applying `hd` works:

```
# hd(from 0);;
- : int = 0
```

At this point we are in the strange situation that the tail of an `llist` is not an `llist` itself (but rather a function from `unit` to `llist`). We solve this by using a mutually recursive type definition.

```
type 'a cell = Nil | Cons of ('a * 'a llist)
and 'a llist = (unit -> 'a cell)
```

Again we give some values of this new type of lazy lists.

Example 10.2.

`fun () -> Nil`	`[]`
`fun () -> Cons(1,fun () -> Nil)`	`[1]`
`fun () -> Cons(2, fun () -> Cons(1,fun () -> Nil))`	`[2;1]`

The new definition of `'a llist` requires a further modification of `from` (and this time also of `hd`).

```
let hd xs = match xs() with Nil       -> failwith "empty"
                          | Cons(x,_) -> x

let rec from n = fun() -> Cons(n,from(n+1))
```

The implementation of the corresponding `tl` function is left as an exercise (cf. Exercise 10.1). Further consider the version of `zip_with` for lazy lists

```
let rec zip_with f xs ys = fun() -> match (xs(),ys()) with
  | (Cons(x,xs),Cons(y,ys)) -> Cons(f x y,zip_with f xs ys)
  | _                       -> Nil
```

Note how a call to `zip_with f xs ys` immediately returns a function waiting for a `()` argument, instead of computing something. Only after this `()` was supplied, any computation starts.

It is about time for the first example on how to use lazy lists. But before that we need a function that transforms (part of) a lazy list into a list.

```
let rec to_list n xs = if n < 1 then [] else match xs() with
  | Nil        -> []
  | Cons(x,xs) -> x :: to_list (n-1) xs
```

10.2.1. The Fibonacci Numbers

As a first application of lazy lists we consider Fibonacci numbers. Remember that the ith Fibonacci number F_i is given by the equation

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise} \end{cases}$$

(Note that in contrast to Chapter 7, we do start the sequence with 0 here. Both definitions are used in the literature.) Hence for all but the first two Fibonacci numbers, the value is determined by its predecessors. This insight can be turned into the following definition:

```
let rec fibs =
  fun() -> Cons(0,fun() -> Cons(1, zip_with (+) fibs (tl fibs)))
```

Consider the two occurrences of `fibs` in the function body. The first one just returns a list starting with 0, 1, whereas the second one returns a list starting with 1 (since the call to `tl` removes the first element). So the call to `zip_with` is of the form

`zip_with (+) [0;1;...] [1;...]`

This is just enough information to compute the first element of the resulting list, leading to

`1 :: zip_with (+) [1;...] [...]`

Now it is clear that the third element of `fibs` is 1 and hence the '`...`' in above expression can be replaced by '`1;...`' yielding:

`1 :: zip_with (+) [1;1;...] [1;...]`

Which is equivalent to

`1 :: 2 :: zip_with (+) [1;...] [...]`

This can be continued ad infinitum, thereby computing all the Fibonacci Numbers. To see that this really works, we use `to_list` to get the first few Fibonacci numbers

```
# to_list 10 fibs;;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

You have seen that it is possible to define your own lazy lists. One problem with above approach however, is that unnecessary recomputations are done. By measuring the time of computing '`to_list` n `fibs`' for bigger and bigger n, it should be possible to convince yourself that above implementation has an exponential time-complexity in n. The problem is that our implementation somehow uses call-by-name evaluation for lists, but another essential ingredient for *lazy evaluation* is missing: *memoization*. It has already been stated (on page 39; in the footnote) that lazy evaluation corresponds to call-by-name evaluation together with memoization.

Therefore, OCaml provides means for proper lazy evaluation (i.e., using memoization) that prohibits unnecessary recomputations. This is discussed in the next section.

10.3. Lazyness in OCaml

The keyword **`lazy`** is reserved to indicate that some expression should be evaluated lazily. It works as a function that lifts an arbitrary expression to its lazy equivalent, i.e., of type `'a -> 'a Lazy.t`. Consider for example the function

```
let one _ = 1
```

that ignores its argument and returns 1. When called eagerly, as for example in the call `one(print_string "test")`, first the argument `print_string "test"` is evaluated (printing "test"), and then 1 is returned. If the same function call is done lazily however, the result differs. Namely, the function call `one(lazy(print_string "test"))` just returns 1 without printing "test", since the argument is never used within the function body.

Together with **`lazy`** comes the function `force : 'a Lazy.t -> 'a` from the module `Lazy` that provides a way to force the evaluation of some lazy expression, i.e., a lazy expression will not evaluate until forced to do so.

The above mechanism can also be used to implement lazy lists. Let us first have a look at the type definition:

```
type 'a t   = 'a cell Lazy.t
and 'a cell = Nil | Cons of ('a * 'a t)
```

Hence a lazy list is a lazy `cell` that is either empty (`Nil`) or contains an element together with a lazy list (`Cons of 'a * 'a t`).

The next example lists some further values of this new type.

Example 10.3.

```
lazy Nil                                  []
lazy (Cons(1,lazy Nil))                   [1]
lazy (Cons(2,lazy (Cons(1,lazy Nil))))    [2;1]
```

For convenience the function `fc` is installed as an abbreviation for `Lazy.force`.

```
let fc = Lazy.force
```

Consider `from: int -> int t` as a first example of a function on this kind of lazy lists.

```
let rec from n = lazy(Cons(n,from(n+1)))
```

Next consider `filter : ('a -> bool) -> 'a t -> 'a t` which removes all elements from a lazy list that do not satisfy a given predicate.

```
let rec filter p xs = lazy(match fc xs with
  | Nil        -> Nil
  | Cons(x,xs) -> if p x then Cons(x,filter p xs)
                         else fc(filter p xs)
)
```

The function `to_list : int -> 'a t -> 'a list` is used to translate a lazy list into a 'normal' list:

```
let rec to_list n xs = if n < 1 then [] else match fc xs with
 | Nil        -> []
 | Cons(x,xs) -> x :: to_list (n-1) xs
```

10.3.1. The Sieve of Eratosthenes

The *Sieve of Eratosthenes* is a simple algorithm to compute all prime numbers. The idea is to start at the sequence of *all* natural numbers from 2 on, i.e.,

$$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad \ldots.$$

Then repeatedly apply the following steps to the sequence:

a) mark the first element h of the sequence as prime

b) remove all multiples of h (including h itself) from the sequence

c) goto step 1

Here are the first few iterations:

$$\begin{array}{rccccccccccccccc}
\boxed{2} & 3 & \underline{4} & 5 & \underline{6} & 7 & \underline{8} & 9 & \underline{10} & 11 & \underline{12} & 13 & \underline{14} & 15 & \underline{16} & \ldots \\
\boxed{3} & 5 & 7 & \underline{9} & 11 & 13 & \underline{15} & 17 & 19 & \underline{21} & 23 & 25 & \underline{27} & 29 & 31 & \ldots \\
\boxed{5} & 7 & 11 & 13 & 17 & 19 & 23 & \underline{25} & 29 & 31 & \underline{35} & 37 & 41 & 43 & 47 & \ldots \\
\boxed{7} & 11 & 13 & 17 & 19 & 23 & 29 & 31 & 37 & 41 & 43 & 47 & \underline{49} & 53 & 59 & \ldots \\
\boxed{11} & 13 & 17 & 19 & 23 & 29 & 31 & 37 & 41 & 43 & 47 & 53 & 59 & 61 & 67 & \ldots \\
\vdots & & & & & & & & & & & & & & & \ddots
\end{array}$$

The Sieve in OCaml Using lazy lists the Sieve of Eratosthenes can easily be implemented in OCaml. First some code is needed that generates the (infinite) list of all natural numbers from 2 on. This is done by

```
from 2
```

using the function `from` from page 88. Then the sieve itself is specified by:

```
let rec sieve xs = lazy(match fc xs with
 | Nil        -> Nil
 | Cons(x,xs) ->
  Cons(x,sieve(filter (fun y -> y mod x <> 0) xs))
)
```

Using this, the list of all primes can be computed by

```
let primes = sieve(from 2)
```

To get concrete results the function `to_list` from above is used. E.g., the first 100 prime numbers are computed by:

```
# to_list 100 primes;;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43;
 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97; 101; 103;
 107; 109; 113; 127; 131; 137; 139; 149; 151; 157; 163;
 167; 173; 179; 181; 191; 193; 197; 199; 211; 223; 227;
```

```
229; 233; 239; 241; 251; 257; 263; 269; 271; 277; 281;
283; 293; 307; 311; 313; 317; 331; 337; 347; 349; 353;
359; 367; 373; 379; 383; 389; 397; 401; 409; 419; 421;
431; 433; 439; 443; 449; 457; 461; 463; 467; 479; 487;
491; 499; 503; 509; 521; 523; 541]
```

These are 18 lines in total (including the type of lazy lists) to compute arbitrarily many prime numbers! (In a language that has built-in lazy lists, this would be a two-liner.)

10.4. Exercises

Exercise 10.1. Consider the following type for lazy lists (from Section 10.2)

```
type 'a cell = Nil | Cons of ('a * 'a llist)
and 'a llist = (unit -> 'a cell)
```

Implement the functions

- `tl: 'a llist -> 'a llist` that returns the tail of a lazy list
- `append : 'a llist -> 'a llist -> 'a llist` that concatenates two lazy lists
- and `map : ('a -> 'b) -> 'a llist -> 'b llist` that maps a function over a lazy list.

Exercise 10.2. Implement the *Sieve of Eratosthenes* using the type of Exercise 10.1.

Exercise 10.3. Use the type of Section 10.3 to implement a function `fibs : int t`, which computes the infinite list of Fibonacci numbers.

Exercise 10.4. Use the type of Section 10.3 to implement a function `merge` that combines two sorted lazy lists, i.e.,

`merge [1;4;6;7;8;...] [1;2;3;4;10;...] = [1;1;2;3;4;4;6;7;8;10;...]`

Exercise 10.5. Use the type of Section 10.3 to implement a function `sunique` that drops duplicates in a *sorted* lazy list, i.e.,

`sunique [1;1;2;3;4;4;6;7;8;10;...] = [1;2;3;4;6;7;8;10;...]`

Exercise 10.6. Use the type of Section 10.3 to implement a function `hamming : int t` that lazily computes the infinite sequence of *Hamming numbers* (i.e., all natural numbers whose only prime factors are 2, 3, and 5), e.g.,

`hamming = [1;2;3;4;5;6;8;9;10;12;15;16;18;20;...]`

Hint: Use `map`, `merge`, and `sunique`.

Exercise 10.7. Consider the function `filter` from Section 10.3, defined as

```
let rec filter p xs = lazy(match fc xs with
  | Nil        -> Nil
  | Cons(x,xs) -> if p x then Cons(x,filter p xs)
                         else fc(filter p xs)
)
```

Does it differ (and if how) from the following function?

```
let rec filter2 p xs = match fc xs with
  | Nil        -> lazy Nil
  | Cons(x,xs) -> if p x then lazy(Cons(x,filter2 p xs))
                         else filter2 p xs
```

Hint: Via `#remove_printer LazyList.toplevel_printer_int;;` you can remove the pretty printer.

Exercise 10.8. Consider the function `zip_with` from Section 10.2, defined as

```
let rec zip_with f xs ys = fun() -> match (xs(),ys()) with
  | (Cons(x,xs),Cons(y,ys)) -> Cons(f x y,zip_with f xs ys)
  | _                       -> Nil
```

Does it differ (and if how) from the following function?

```
let rec zip_with2 f xs ys = match (xs (),ys ()) with
  | Cons(x,xs),Cons(y,ys) -> fun () -> Cons(f x y,zip_with2 f xs ys)
  | _                     -> fun () -> Nil
```

Hint: Try to implement the Fibonacci numbers via `zip_with2`.

Exercise 10.9. Read the following article (available from UIBK network):
Melissa E. O'Neill, The Genuine Sieve of Eratosthenes, Journal of Functional Programming, 19(1), 95–106 (2009), doi:10.1017/S0956796808007004.

The whole is of necessity prior to the part.

Aristotle

11. Divide and Conquer

11.1. Divide and Conquer

Many recursive algorithms follow the *divide and conquer* philosophy. They **divide** the problem into smaller subproblems (of the same shape) and then **conquer** these subproblems (either because they are trivial or they are further divided). Finally the solutions of the subproblems are **combined** into a solution for the original problem.[1]

In the sequel we will demonstrate the *divide and conquer* principle by discussing *mergesort*, which works as follows: To sort a list `zs` we distinguish two cases. In the base case `zs` contains at most one element and is already sorted. In the step case we *divide* `zs` into two sublists `xs` and `ys` (such that `zs` = `xs@ys`). After sorting `xs` and `ys` we *merge* the result, such that the obtained list is also sorted. In OCaml, mergesort can be implemented as follows:

```
let rec merge xs ys = match (xs,ys) with
  | ([],ys) -> ys
  | (xs,[]) -> xs
  | (x::xs,y::ys) -> if x < y then x::(merge xs (y::ys))
                             else y::(merge (x::xs) ys)

let rec msort = function
  | [] -> []
  | [z] -> [z]
  | zs -> let (xs,ys) = Lst.split_at (Lst.length zs / 2) zs in
          merge (msort xs) (msort ys)
```

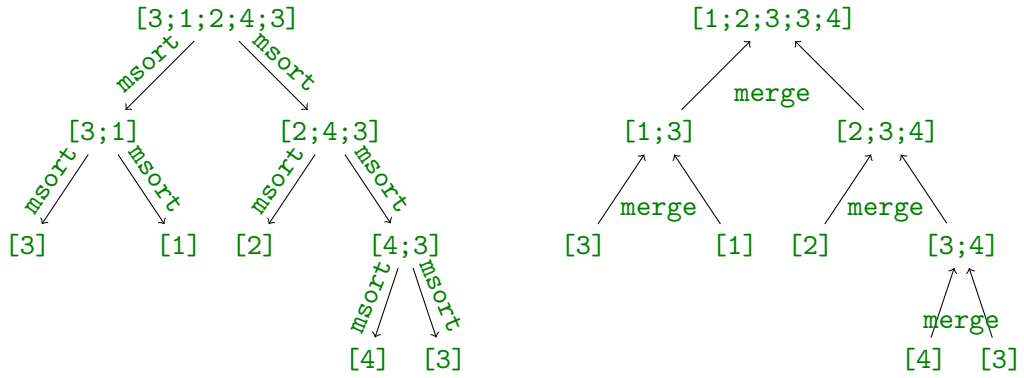
The above implementation divides the list which should be sorted (approximately) at the middle (using `split_at`) and sorts the first half and the second half separately. Finally the obtained lists are *merged* into the resulting list. The execution of `msort` can be visualized by the trees in Figure 11.1. First the recursive calls to `msort` decompose the list until all lists are singleton. This phase is top to bottom (cf. Figure 11.1(a)). In a second phase the resulting list is built by merging the already sorted ones. This phase is executed bottom to top (cf. Figure 11.1(b)).

Next we will investigate the runtime of `msort`. Note that the runtime of `msort` is mainly depending on the length of the list to sort. The actual list contents are of minor importance. Hence the runtime of `msort` is a function $T\colon \mathbb{N} \to \mathbb{N}$ where n is the length of the list zs we want to sort and $T(n)$ is the number of instructions that are executed when calling `msort` zs. To simplify matters we will consider a worst case analysis and study the *asymptotic* runtime only.[2] We will use standard O-notation for this. Hence we are looking for a function $f\colon \mathbb{N} \to \mathbb{N}$ such that $T \in O(f)$. Sometimes we are interested in tight upper bounds on $T(n)$ and then write $T \in \Theta(f)$.

The runtime of a divide and conquer algorithm can always be computed by summing up the runtime for each of the steps to *divide* the problem, to *conquer* the subproblems, and to *combine* the solutions. We investigate each of the steps below for our implementation of `msort`:

[1] Consequently the concept should be named *divide and conquer and combine* but since this sounds worse (and is more work to write) we prefer the shorter name.

[2] Here *asymptotic* means that we are not interested in the exact number of operations `msort` executes but in the *order* of the performed operations.



(a) Divide (b) Combine

Figure 11.1.: Evaluation of `msort [3;1;2;4;3]`.

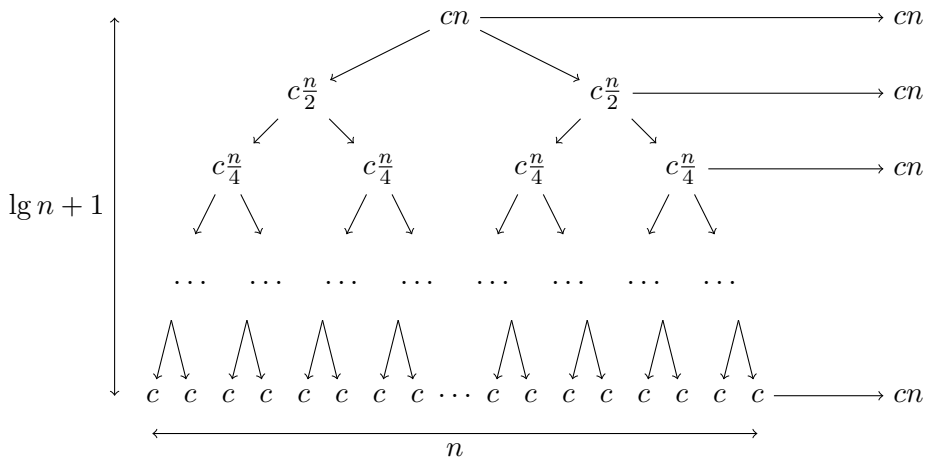


Figure 11.2.: A tree for the recurrence (11.1).

- *divide*: We divide the original list of length n into two lists. We used the function `split_at` for this purpose. Looking at the implementation of `split_at` we see that this part can be done in $O(n)$ time. Hence the runtime of *divide* is $O(n)$.
- *conquer*: Here we must consider two cases. If we have an empty or a singleton list, then this list is already sorted and we can just return it. Hence in this case we have constant runtime, i.e., $O(1)$. In the other case we *conquer* each of the two lists (obtained from *divide*). Note that `split_at` produced two sublists, each of length (approximately) $\frac{n}{2}$. To sort one of them we need time $T(\frac{n}{2})$ and hence for both of them we obtain a runtime of $2T(\frac{n}{2})$. Hence in this case *conquer* runs in time $2T(\frac{n}{2})$.
- *combine*: Finally we need to *combine* the two sorted lists into a single sorted list (using `merge`). Since both of these lists are of length $\frac{n}{2}$ and one element of either list is removed in each recursive call, `merge` runs in time $O(n)$.

Hence for `msort` the function T can be given as follows (for some constant c):

$$T(n) = \begin{cases} c & \text{if } n \leqslant 1 \\ 2T(\frac{n}{2}) + cn & \text{otherwise} \end{cases} \tag{11.1}$$

The case $n \leqslant 1$ will be called the base case (in contrast to the step case). Note that $2T(\frac{n}{2})$ is the time needed for *conquer* while cn is the accumulated time for *divide* and

combine. Equations of the form (11.1) are called *recurrence equations* or just *recurrences*. Note that the recurrence (11.1) does not yet give a bound on the runtime of `msort`. However, we can use it to compute such a bound. We unfold recurrence (11.1) recursively and obtain a binary tree (see Figure 11.2). For ease of discussion we assume that n is a power of 2, i.e., $n = 2^k$ for some $k \in \mathbb{N}$, and hence the tree is perfect (see Section 6.1.2). Each node corresponds to a call to `msort` and we label the node by the runtime needed for the phases *divide* and *combine*. At the root node we split a list of length n into two sublists (*divide*) and then merge the sorted lists, each of length (approximately) $\frac{n}{2}$ (*combine*). Hence this node gives runtime cn. The two recursive calls (on lists of length $\frac{n}{2}$) give runtime $c\frac{n}{2}$ each, so together also yield cn. This actually holds for each *level* of the tree. Note that the bottom level of the tree has n nodes since we started with a list of length n. So how many levels are there? Or stated differently: What is the height of the tree? From Exercise 11.1 we obtain that there are $\lg n + 1$ levels and hence $T(n) \leqslant cn \times (\lg n + 1) \in O(n \lg n)$.[3]

There are several ways to solve recurrence equations, but for the ones we meet in this course unfolding the tree will suffice. In general the runtime of a divide and conquer algorithm can be given as

$$T(n) = \begin{cases} c & \text{base case} \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{step case} \end{cases} \tag{11.2}$$

Here a is the number of subproblems that must be conquered in the step case and $\frac{n}{b}$ is the size of each of these problems. Furthermore the time needed to divide ($D(n)$) the problems and to combine ($C(n)$) the solutions must be added. For `msort` above we have $a = b = 2$, but there exist many other problems where a and b are different from two (and even different from each other). Note that $D(n) + C(n)$ tells us how costly a single function call is, whereas $aT(\frac{n}{b})$ gives us the number of recursive function calls we have to consider. However, the problems treated recursively are smaller ($\frac{n}{b}$), which we must take into account.

To get an even simpler form of recurrence equations we often collapse the cases for dividing problems, combining solutions, and the base case. Then a recurrence equation reads as follows:

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{11.3}$$

Here $a, b \in \mathbb{N}$ and $f\colon \mathbb{N} \to \mathbb{N}$ is an asymptotically positive function. Often we do not need to solve recurrence equations on our own, but we can use the *Master Theorem*, which tells us how the solutions look like in many cases. Basically, we compare the cost for a single function call ($f(n)$) with the number of nodes in the last level of the tree ($n^{\log_b a}$, which estimates the number of recursive calls needed if $b > 1$). If the cost for a single call is significantly smaller than the number of recursive calls ($f(n) \in O(n^{\log_b a - \epsilon})$, then the latter determines the overall complexity. This corresponds to the first case in the theorem. The reasoning for the third case is analogous. In the second case the cost of one function call approximately equals the number of recursive calls and hence the reasoning is similar as in Figure 11.2, explaining the additional factor $\lg n$.

Theorem 11.1 (Master Theorem). *Let $a \geqslant 1$, $b > 1$, and $T(n)$ as in (11.3). Then*

a) $T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$.

b) $T(n) \in \Theta(n^{\log_b a} \lg n)$ if $f(n) \in \Theta(n^{\log_b a})$.

c) $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ and $af(\frac{n}{b}) \leqslant cf(n)$ for some $c < 1$ and sufficiently large n.

[3] By $\lg n$ we abbreviate $\log_2 n$.

Since for `msort` we have $a = b = 2$ and $f(n) \in \Theta(n)$, the Master Theorem (case 2) applies and yields $T(n) = \Theta(n \lg n)$.

We have seen that for many divide and conquer algorithms we can easily conclude their runtime by just checking if the Master Theorem applies. But this is far from being the only benefit. Another nice fact about recursive algorithms is that proving their correctness is usually easier than when considering loops since induction can be applied. In the exercises you are asked to prove the correctness of `msort` (see Exercise 11.3). Have you ever tried to prove *bubblesort* correct?

11.2. Dynamic Programming

Dynamic programming is a technique that prefers *recalling* over *recomputing*. To this end it stores the results of subproblems and just looks the result up instead of recomputing it. Clearly, storing the results for subproblems will need additional memory. Not all divide and conquer problems are equally well-suited for dynamic programming. Those where it is necessary to solve many (identical) subproblems frequently will benefit more than problems where all subproblems look (fairly) different. Another important issue is that the *lookup* of a result should be drastically cheaper than the *recomputation* of a result. In practice one typically uses data structures that allow a lookup in constant or logarithmic time. In the imperative world this property holds for hash tables (lookup is possible in constant time if there is at most one element in each bucket) whereas in the functional setting often binary search trees (lookup is possible in logarithmic time if the tree is balanced) are used. Since theory is pretty independent from such implementation matters we will refer to it in the sequel just as *(lookup) table* and assume that operations such as insertion (`add`) and lookup (`find`) are sufficiently efficient. We will use the interface to a lookup table shown in Listing 11.1.

```
(** Lookup Tables *)
type ('a,'b) t
(** the type of a lookup table *)
val empty: ('a,'b) t
(** [empty] constructs empty [t] *)
val mem: ('a,'b) t -> 'a -> bool
(** [mem t k] tests if there is a value associated with key [k] in [t] *)
val find : ('a,'b) t -> 'a -> 'b
(** [find t k] returns value [v] associated with key [k] in [t]
 @raise Not_Found if [t] has no binding for [k] *)
val add : ('a,'b) t -> 'a -> 'b -> ('a,'b) t
(** [add t k v] adds key-value pair [(k,v)] to [t] if [k] is not in [t] *)
val size : ('a,'b) t -> int
(** [size t] returns the size, i.e., number of bindings in [t] *)
```

Listing 11.1: `Lookup.mli`

The type of a lookup table is `('a,'b) t` where `'a` is the type for the *keys* and `'b` the type for the *values* associated with the keys. The constant `empty` returns an empty table. The function `mem t k` checks if table `t` contains a binding for the key `k`. If so, then `find t k` returns the value `v` associated to `k` in table `t`. Finally, `add t k v` adds a new binding `(k,v)` to table `t` and returns the new table.

11.2.1. Fibonacci Numbers

We have already seen that the straightforward implementation of the Fibonacci function is inefficient, since `fib n` results in approximately 2^n recursive calls. To overcome this exponential growth we will dynamically program (hence the name) a table, where for each m (here $1 \leqslant m \leqslant n$) we have two cases. If we did not yet consider m we compute `fib` m and add as a binding for m the value of `fib` m to the table. If m has already been considered, then we just look up the binding, i.e., `fib` m, in the table. To store the table linear space is needed but now it is possible to compute `fib` m with linearly many recursive calls. An implementation of the Fibonacci numbers using dynamic programming can be done as follows:

```
let fib_dp n =
 let rec fib t n =
  if Lookup.mem t n then t
  else if n < 2 then Lookup.add t n 1
  else
   let t = fib t (n-1) in
   let t = fib t (n-2) in
   let r = Lookup.find t (n-1) + Lookup.find t (n-2) in
   let t = Lookup.add t n r in
   t
 in Lookup.find (fib Lookup.empty n) n
```

The differences to the original implementation are as follows: The (inner) `fib` function now has an additional parameter `t`, which is a lookup table that will finally contain all Fibonacci numbers as bindings. Hence `fib` now returns a lookup table instead of a single Fibonacci number. The first thing `fib` checks is if it has already computed the Fibonacci number for the current parameter n. In this case the binding is already in the table and it can be returned without changes. If there is no binding yet for n then we have to add it. This is easy in the base case ($n < 2$). In the step case we first get the (updated) tables for the recursive calls to $n-1$ and $n-2$. Now the table contains bindings for the $(n-1)$-st and the $(n-2)$-nd Fibonacci numbers which we can just lookup in the table to compute the n-th Fibonacci number, which we finally add to the table before we return it. Since `fib Lookup.empty n` returns a table containing the first n Fibonacci numbers the last line then looks up the n-th Fibonacci number.

We have seen that dynamic programming allows to reduce exponential runtime to polynomial runtime while the additional memory needed is only linear (in the size of the input).

11.2.2. Beans and Bowls

Consider a bowl containing black and white beans. We may replace beans by the following laws:

a) Replace two black beans by a single white bean.

b) Replace two white beans by a single black bean.

c) Replace a black and a white bean by a single white bean.

These laws can be written more concisely as *rewrite rules*[4]

●● → ○ ○○ → ● ●○ → ○ ○● → ○

[4]Here we use two rewrite rules for the last law to show that the order of the beans does not matter.

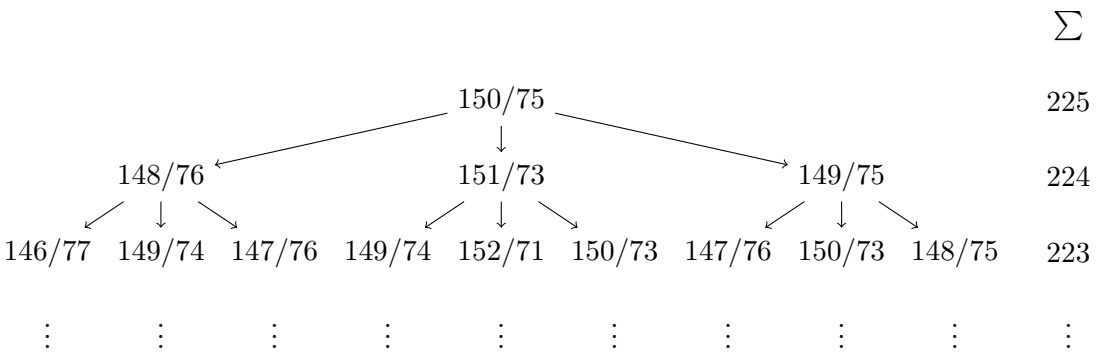


Figure 11.3.: Naive algorithm starting with 150 black and 75 white beans.

Now we face the following question: Starting with a bowl containing 150 black and 75 white beans can the color of the last bean be predicted? In other words the question is if we always end up with either a white or a black bean (independent from *how* we replace the beans).

First we tackle the problem with a naive algorithm that just tries all possibilities. Starting with 150 black and 75 white beans we consider each of the three rules separately. Figure 11.3 visualizes this strategy. Note that the naive algorithm terminates since the (total) number of beans decreases by one in each step (see the right column in Figure 11.3). To compute the number of recursive calls we recall that a (non-empty) ternary tree of height n has (more than) 3^{n-1} nodes. For $n = 225$ we get $3^{224} = (3^2)^{112} = 9^{112} \approx 10^{100}$, which is intractable for a computer (which we will see soon).

For the naive approach we write a function `beans : int -> int -> (bool * bool)` which takes two integers (number of black and white beans, respectively) and returns a pair of booleans. Here the first component is true if and only if it is possible to end up with a black bean and the second component indicates if we can end up with a white bean (it might be that both possibilities come true). The function can then be implemented as follows:

```
let rec beans b w =
 if b = 1 && w = 0 then (true,false)
 else if b = 0 && w = 1 then (false,true)
 else if b < 0 || w < 0 then (false, false)
 else
  (beans (b-2) (w+1)) <||>
  (beans (b+1) (w-2)) <||>
  (beans (b-1) w)
```

Here the operator `<||>` is a logical or on pairs, defined as

```
let (<||>) (b1,w1) (b2,w2) = (b1 || b2,w1 || w2)
```

However, while e.g., `beans 5 5` shows that for *some* starting configurations it is possible to end up with a black or a white bean the call `beans 150 75` does not terminate within reasonable time. The reason is that similar instances of the problem are considered over and over again (see Figure 11.3 and Exercise 11.11). To overcome this bottleneck we use dynamic programming. In the function `beans_dp` we first fill the table (with the inner `beans` function) and then we can access the result for a concrete configuration.

```
module L = Lookup

let beans_dp b w =
 let rec beans t b w =
```

```
  if L.mem t (b,w) then t else
  if b = 1 && w = 0 then L.add t (b,w) (true,false)
  else if b = 0 && w = 1 then L.add t (b,w) (false,true)
  else if b < 0 || w < 0 then L.add t (b,w) (false, false)
  else
   let t = beans t (b-2) (w+1) in
   let t = beans t (b+1) (w-2) in
   let t = beans t (b-1) w in
   let r = L.find t (b-2,w+1) <||> L.find t (b+1,w-2) <||> L.find t (b-1,w) in
   L.add t (b,w) r
in L.find (beans L.empty b w) (b,w)
```

And indeed, if we start with 150 black and 75 white beans the outcome can either be a black or a white bean. Note that the call `beans_dp 150 75` does not return immediately (On a contemporary laptop it takes about a second). To understand this (shockingly slow) execution we first take a look at the search space. The call `beans_dp 150 75` requires 19,138 entries in the lookup table and from Exercise 11.13 we obtain that in general the lookup table (computed for the call `beans_dp` m n) is quadratic in $m + n$. Hence the runtime of `beans_dp` will be (at least) quadratic. The overall runtime also depends on the implementation of the `Lookup` module. Ours is based on binary search trees, where `mem` and `find` have logarithmic runtime (if the tree is balanced, otherwise the runtime is linear). Hence the overall runtime of `beans_dp` will be between $O(n^2 \cdot \log n)$ and $O(n^3)$.

11.2.3. Optimal Rod Cutting

Another example demonstrating the benefit of dynamic programming is the *Optimal Rod Cutting* problem. This problem comes as an *optimization* problem. We are given a rod of length n and a table of prices p_i for $1 \leqslant i \leqslant n$ where p_i is the price for a rod of length i. The question is to maximize the profit when cutting a rod of length n into pieces. Consider the following example.

Example 11.1. Given the following table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	2	3	5	5	8	12	12	15	15	17

What is the maximum price that can be obtained when cutting a rod of length 10? If we do not cut at all the price is 17. If we cut the rod in two pieces of lengthes 8 and 2 the price is $15 + 3 = 18$, etc. Later we will see that the maximal price is 20 for this instance. Note that for simplicity we are currently not interested in *how to cut the rods* to obtain this answer.

We will solve the optimal rod cutting problem recursively. Let r_i (for $0 \leqslant i < n$) be the optimal solution for a rod of length i. Then

$$r_n = \max_{0 \leqslant i < n} \{p_n, p_i + r_{n-i}\} \tag{11.4}$$

This formula considers a rod of length n and says that for an optimal solution (r_n) we either do not cut at all (p_n) or take the maximum sum of p_i (which is not cut further) and r_{n-i} (which is an optimal solution for a smaller problem). We can implement the above formula as follows:[5]

```
let price ps i = if i <= Lst.length ps then Lst.nth ps (i-1) else 0
```

[5] Note that the original problem is restricted to rods for which a price is specified. The implementation is slightly more general by assigning a price of zero to rods which are too long, i.e., ones that we cannot sell.

```
let rec cut ps n =
 if n = 0 then 0
 else
  let f i q = max q (price ps i + cut ps (n-i)) in
  Lst.foldr f (price ps n) (IntLst.range 1 n)

let ps = [2;3;5;5;8;12;12;15;15;17]
```

Here `price ps i` returns the price for a rod of length `i` and `cut` implements the formula (11.4) for a price list `ps`. Now `cut ps 10 = 20` can be computed in almost no time but the call `cut ps 30` already seems to take forever. The reason is that the same problems are solved again and again. We can remove the bottleneck similar as for the Fibonacci numbers by dynamic programming. The result looks as follows:

```
let cut_dp ps n =
 let rec cut t ps n =
  if Lookup.mem t n then t
  else if n = 0 then Lookup.add t n 0
  else
   let t = Lst.foldr (fun i t -> cut t ps (n-i)) t (IntLst.range 1 n) in
   let f i q = max q (price ps i + Lookup.find t (n-i)) in
   let r = Lst.foldr f (price ps n) (IntLst.range 1 n) in
   Lookup.add t n r
in Lookup.find (cut Lookup.empty ps n) n
```

The (inner) `cut` function is almost the same as before. The main difference is that we first update the lookup table before computing the optimal value r for a rod of length n.

We observe that `rod_dp 30 = 60` is now computed almost instantaneously. While `rod_dp` is significantly faster than `rod` it might not be satisfactory because an optimal solution does not yet tell us *how* the rod should be cut. Adding this information is the task of Exercise 11.10.

11.3. Chapter Notes

This chapter builds on Chapter 4 (divide and conquer) and Chapter 15 (dynamic programming) from [4].

While *divide and conquer* techniques are not restricted to functional programming they often appear in this programming paradigm due to the heavy use of recursion. The principle itself is much older, however. Already in 1805 Carl Friedrich Gauss presented a fast Fourier transformation algorithm where the problem is divided into smaller subproblems whose solutions are combined. Recurrence equations have already been studied by Fibonacci in the 13th century.

While *dynamic programming* is a simple idea itself, it took until the 1950's to properly study the underlying mathematics. The idea of dynamic programming sometimes is also referred to by the name *memoization*. Note that some other (functional) programming languages have memoization already built in (e.g. Haskell).

There are many other real world problems where an efficient implementation is possible using dynamic programming. We mention an important one which is finding longest common subsequences of two strings. This is used in DNA analysis to determine the similarity of two genes. Other problems that benefit from dynamic programming are discussed in the exercises.

11.4. Exercises

Exercise 11.1. Consider a perfect binary tree where the last level has n nodes. Show that the tree has $\lg n + 1$ levels, i.e., height $\lg n + 1$.

Hint: How many nodes does the tree have? Lemma 6.5 might be helpful.

Exercise 11.2. Prove the following claim by induction:

If xs and ys are sorted lists then `merge` xs ys is a sorted list.

Hint: Which kind of induction is useful?

Exercise 11.3. Show by structural induction on lists that for all lists zs the result of `msort` zs is a sorted list.

Hint: You can assume the claim in Exercise 11.2.

Exercise 11.4. Consider a different implementation of `msort` where the list `zs` is split differently, i.e., `(xs,ys) = (Lst.hd zs, Lst.tl zs)`. Is the (worst case) runtime affected by this change?

Exercise 11.5. Write a function `qsort`, which implements *quicksort*.

Hint: For a non-empty list select the head element as pivot.

Exercise 11.6. Consider the Fibonacci numbers (see Definition 7.1).

a) Compute a recurrence equation for the Fibonacci numbers.

b) Does the Master Theorem apply to the recurrence from item 1?

Exercise 11.7. Consider the following implementation of *insert* sort.

```
let rec insert x = function
 | [] -> [x]
 | y::ys -> if x < y then x::y::ys else y::(insert x ys)

let rec isort = function
 | [] -> []
 | x::xs -> insert x (isort xs)
```

a) Compute the recurrence equation for `isort`.

b) Solve the recurrence by unfolding it into a tree. Conclude an upper bound for the runtime of `isort`.

c) Does the Master Theorem apply to the recurrence from item 1?

Exercise 11.8. Give a recurrence where the Master Theorem does not apply.

Hint: Find (un)suitable values for a, b, and $f(n)$.

Exercise 11.9. Consider the Optimal Rod Cutting problem. A *greedy* strategy chooses a j such that $p_j + p_{n-j}$ is maximal. Then it performs the recursive calls on the shorter rods (of lengths j and $n - j$) such as in the following function:

```
let rec cut_greedy ps n =
 if n = 0 then 0
 else
  let f i (l,q) =
   let p = price ps i + price ps (n-i) in
   if p > q then (i,p) else (l,q)
  in
  let (j,_) = Lst.foldr f (n,price ps n) (IntLst.range 1 n) in
  if j = n then price ps n else cut_greedy ps j + cut_greedy ps (n-j)
```

a) Show that the greedy strategy does not necessarily yield an optimal solution.

b) What is the runtime of `cut_greedy`?

Exercise 11.10. Extend the function `cut_rod` such that it also returns *how* an optimal solution can be obtained by indicating how the rod must be cut.

Hint: Add this information to the lookup table (it might now contain triples (r_n, l, r) where r_n is the optimal solution to a rod of length n and l and r are the lengths of the left and right rod, respectively).

Exercise 11.11. Consider beans & bowls.

a) Give an upper bound on the number of recursive calls emerging from `beans` m n (in terms of m and n).

b) Draw the recursive calls emerging from `beans 3 3` as a tree. The nodes are labeled `beans` m n for different values of m and n and there is an edge from node `beans` m n to node `beans` m' n' if the former recursively calls the latter.

Hint: Share identical nodes in the tree.

c) Use the tree from item 2 to compute the number of (recursive) calls to `beans` (starting from `beans 3 3`). Check your computation by adding a counter to `beans`.

d) Use the tree from 2 to compute the number of (recursive) calls to `beans` (starting from `beans_dp 3 3`). Check your computation by adding a counter to `beans` (inside `beans_dp`).

Exercise 11.12. Extend `beans_dp` such that it returns the sequence of choices that yield a single black/white bean.

Exercise 11.13. Give an upper bound (in terms of m and n) on the size of the lookup table generated for the call `beans_dp` m n. Conclude that the size of the lookup table is quadratic in $m + n$. Compare your upper bound with the exact size of the lookup table for $m = 150$ and $n = 75$.

Hint: Note that $m \times n$ is not a correct upper bound. Why?

Exercise 11.14. Consider Post's Correspondence Problem (PCP). Here PCPs are given as a list of pairs where each pair contains the corresponding words, i.e., the words at the same indices.

The following implementation tries to determine if a PCP has a solution or not by testing all possibilities in a breadth first search. To save memory, common prefixes of two words are removed using the function `trim`.

```
let rec trim = function
 | (x::xs,y::ys) when x = y -> trim (xs,ys)
 | d -> d

let extend (w1,w2) (d1,d2) = trim (w1@d1,w2@d2)

let solve ds =
 let rec solve = function
  | [] -> false
  | ([],[])::_ -> true
  | (x::_,y::_)::ws when x <> y -> solve ws
  | w::ws -> solve (ws@Lst.map (extend w) ds)
 in solve (Lst.map trim ds)
```

a) Write a function `solve_dp`, which uses dynamic programming to avoid considering the same problems again.

b) Can you give an upper bound on the additional memory needed for the lookup table?

c) Write a function `solve_dps`, which returns (the indices of) a solution.

Exercise 11.15. Implement the *Towers of Hanoi* game using a naive approach which just tries all (allowed) moves. Towers of which size does a depth-first/breadth-first traversal manage? How does performance change when using dynamic programming?

Hint: Implement a tower as a list of integers where the value of the integer encodes the size of the disc.

> Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.
>
> John Carmack

A. OCaml in a Nutshell

As the title of this document suggests, the emphasis is on *(purely) functional* programming. Hence only a fragment of OCaml is presented. Imperative features are solely used for I/O, and object orientation is not covered at all. Even of OCaml's purely functional fragment, only constructs that are needed during the lecture, are introduced. Consequently this is by no means a full description of OCaml (for such see [9]). The focus is on techniques that can be used in any functional language (e.g., Erlang, Haskell, Standard ML etc.).

A.1. Availability

The complete OCaml distribution (either source code or binaries for several platforms) can be downloaded from the website `http://caml.inria.fr`. (A lot of additional information on OCaml can be found there too.)

A.2. The Obligatory "Hello, world!"

```
let main() = Printf.printf "Hello, world!\n" in main()
```

Listing A.1: `helloWorld.ml`

The program in Listing A.1 can be described as follows: first a function by the name `main` is defined. Here '`()`' indicates that the function does not take any arguments and the *body* of the function (i.e., everything to the right of '`=`') is just writing the string `"Hello, world!"` on the standard output channel by using the function `printf` of the module `Printf` from the OCaml standard library. This alone would not lead to any output, since the function was just defined but never used. That is exactly what the part after '`in`' takes care of. The function `main` is called on input '`()`' (i.e., empty input).

To get a stand-alone program from the source code of Listing A.1, write it to a file called `helloWorld.ml`. A bytecode executable is compiled with `ocamlc` (the OCaml bytecode compiler) as follows:

```
> ocamlc -o hello helloWorld.ml
```

This tells the compiler to produce an executable called `hello` from the source file `helloWorld.ml`. Running this program yields:

```
> ./hello
Hello, world!
```

The first line of the executable starts with something like

```
#!/usr/bin/ocamlrun
```

indicating that the program `ocamlrun` is used to interpret the contents of this file. What follows is platform independent (and human unreadable) bytecode.

There also is a native-code compiler (`ocamlopt`) for OCaml which is not used in this lecture. The main differences are that native-code is often much faster than bytecode but in return not platform independent.

A.3. Types

The first thing to notice is that there are no statements without return value in OCaml. Every value has a type and hence every single statement in a source file has a type (at least if the code is syntactically correct). Variables do never change their values (i.e., they are mere identifiers/abbreviations of their values and hence the term *variable* is a bit misleading). This sounds restrictive. Indeed it is not (as will be seen in the course of the lecture). In order to build arbitrary types, three ingredients are needed: A set of *basic types*, a set of *type variables*, and a set of *type constructors*.

A.3.1. Basic Types

OCaml offers the following basic types (sometimes also called *primitive types*): Boolean, characters, floating point numbers, integers, strings, and the empty type. These are denoted by the *type constants* `bool`, `char`, `float`, `int`, `string`, and `unit`, respectively.

A.3.2. Type Variables

In OCaml every identifier preceded by a single quote (‘`'`’) is a *type variable* (e.g., `'a`, `'b`, `'c`, ... are often used). A type variable is a placeholder for an arbitrary type. Thus having type `'a` (often pronounced “alpha”) for an expression means that it can have any type. Type variables enable the use of so called *polymorphic types*, i.e., types whose structure is free to some extent.

A.3.3. Type Constructors

In addition to some predefined *type constructors*, OCaml allows (and indeed encourages) a user to define her own type constructors. The only predefined type constructors that will be mentioned here are ‘`*`’ which is used to build *tuple types* and ‘`->`’ which denotes *function types*.

A *tuple* is a mathematical structure with a fixed number of components. A tuple with n components is called an n-tuple. Some special cases are *pairs* (2-tuples), *triples* (3-tuples), *quadruples* (4-tuples), etc.

A function type denotes the type of a function. Does not help much this description, does it? Consider the type `int -> int`. This is the type of all functions that take a single integer as argument and return a single integer as result.

In Section A.3.5 it is shown how new type constructors can be introduced by the programmer.

A.3.4. Examples

The best way to get a feeling for this, is to look at some examples.

```
int
```

the type of integers (e.g., $\ldots, -1, 0, 1, \ldots$).

```
int * int
```

the type of pairs of integers (e.g., $(0,0), (0,1), \ldots$).

```
int -> int -> int
```

the type of functions that take two integers and return a single integer as result. Some prominent members having that type are: Addition ($+$), subtraction ($-$), multiplication ($\cdot$), etc. An important thing to note here, is that ‘`->`’ associates to the right—meaning that one starts to insert parentheses from the right—and hence the same type

can be written as `int -> (int -> int)` but *not* as `(int -> int) -> int` (which would be the type of functions taking a function of type `int -> int` as argument and returning an integer.

And now some polymorphic types:

```
'a * 'b
```

the type of pairs over arbitrary types. The only thing that is known, is that there is a tuple with exactly two components, nothing is known about the types of those components. Very similar but still different is the type

```
'a * 'a
```

which denotes pairs having components of the *same* type.

A.3.5. User-Defined Types

The **`type`** declaration can be used to define new type constructors at will. Type constructors can be parametrized by type variables. The general form is

$$\texttt{type ('a}_1\texttt{,...,'a}_n\texttt{) name = ...}$$

where `name` is the name of the newly defined type constructor and its *type parameters* $\texttt{'a}_1$ to $\texttt{'a}_n$ can be used after '`=`'. Depending on what stands on the right of '`=`', two kinds of user-defined types are distinguished: *Type abbreviations* and *algebraic data types*.

Type Abbreviations

Type abbreviations are used to give existing types shorter and/or more descriptive names. Some examples are

```
type coord = int * int
```

which merely installs `coord` as an abbreviation for the type of integer-pairs,

```
type ('a) eqpair = 'a * 'a
```

which defines the type of all pairs whose components are of the same type (we can also drop the parentheses and write **`type`** `'a eqpair = 'a * 'a` in case of a single type parameter), and

```
type ('a,'b) funs = 'a -> 'a -> 'b
```

which defines `('a,'b) funs` to be the type of functions taking two arguments of the same type and returning a value of some other type.

Algebraic Data Types

Algebraic data types (sometimes called *variant types*) are defined by listing all possible shapes of values of that type where each case is identified by a so called *(data) constructor* (beginning with an uppercase letter). For example

```
type direction = East | North | South | West
```

defines a new type `direction` which consists of just the four values `East`, `North`, `South`, and `West`.

In the example above a variant type was used like an enumeration in other languages (i.e., listing all possible values a certain type can adopt). Indeed variants can be used in a more general way since each of the constructors can itself have an argument of arbitrary type (defined with the keyword **`of`**) as in

```
type number = Int of int | Float of float
```

introducing a new type combining the existing types of integers and floating point numbers. But there are still more general applications of variants. Consider the type declaration

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```

stating that a `mylist` parametrized by an arbitrary type `'a` is either empty (`Nil`) or a list consisting of an element of type `'a` paired with another `mylist` parametrized with the same type `'a`. The type `mylist` represents lists of arbitrary type. Note that each element in such a list must be of the same type.

A.4. Values

The *instances* of a type are called *values*. Values are also called *members* of the specific type. Since every expression has a type in OCaml, in most cases expressions are annotated with their type in the following. This is done in the form

$$e : \tau$$

where e is an expression (for example an instance of some type) and τ is its type.

The available values of type `bool` are **`true`** and **`false`**. For characters a single quote notation is used. Hence the letter 'A' is denoted by `'A'`. Some examples of floating point numbers are: `1.`, `0.0`, `1e1`, `1E-3`. Integers are written as an arbitrary (finite) sequence of digits with a leading '`~-`' for negative numbers (no leading '`+`' is allowed however). Strings finally, are written in *double quote notation* (see Listing A.1 for an example).

Note: Everything is a member of type `'a`.

A.4.1. Tuples

Values having a tuple type are called tuples. There are two special cases of tuples. Firstly 0-tuples which have their own type in OCaml (namely `unit`) and consist of the single value '`()`', denoting "nothing" (comparable to the type `void` of Java). Secondly 1-tuples which coincide with their single component (e.g., `(1)` is the same as `1`). Some examples are

```
() : unit
```

denoting 'nothing',

```
(1,2,3,4) : int * int * int * int
```

a 4-tuple where every component is an integer,

```
("Akira",23) : string * int
```

a *pair* consisting of a string as first component and an integer as second one.

As has already been seen, the type constructor for tuples is '`*`'. Concrete tuples are built using parentheses—'`(`' and '`)`'—to enclose all components of a tuple and commas ('`,`') to separate consecutive components.

A.4.2. Functions

In OCaml functions are just values. A value denoting a function is called a *functional value*. Hence it is possible to write down a function without a name (a so called *anonymous function* or *lambda term*). This is done using '**`fun`**' and '`->`' as in

```
fun x -> x + 1
```

denoting a function which adds `1` to its argument `x`. The general form is

$$\texttt{fun}\ x_1\ x_2\ \ldots\ x_n\ \texttt{->}\ b$$

where x_1 to x_n are the arguments of the function and b is the so called body (i.e., an expression describing what the function does). If $n = 0$, i.e., the function does not take any arguments, then the special notation `fun() ->` e is used.

The call of a function on an argument is called *function application* and denoted by juxtaposition (i.e., writing next to each other, separated by white spaces). Function application is left associative. Hence `f` a_1 a_2 is the same as `(f` a_1`)` a_2 but different from `f (`a_1 a_2`)`. As an example consider the expression

```
(fun x -> x + 1) 1
```

i.e., application of the anonymous function `fun x -> x + 1` to the argument `1`. This results in `1 + 1` which yields `2`.

A.4.3. Variants

The instances of algebraic data types are called *variant values* (or shorter: *variants*). Such values are constructed by using the names of constructors plus parentheses and commas if the specific constructor has an argument. For example consider an instance of the type `int mylist`:

```
Cons(1,Cons(2,Cons(3,Cons(4,Nil)))) : int mylist
```

representing the list of integers

$$1, 2, 3, 4.$$

A.5. Values and Types

Since OCaml uses *type inference* (see Chapter 9 for more information on that) in most cases the user does not have to explicitly assign the type by herself. However it is always possible to specify a type by hand (where the compiler will issue an error if it is not compatible with the automatically inferred type). To think about the type of a function is a good idea since it improves the understanding of this function.

A.5.1. Declaring Values

Almost surely the first thing to do when starting to program, is to give a name to some value. In OCaml this is done via the `let ... = ... in ...` declaration. For instance

$$\texttt{let x = 1 in}\ e$$

binds the name `x` to the value `1` within the expression e. A special case of this—that can only be used at the top level—is `let ... = ...`. It means that from this point in the program on the bound value is accessible. It is of course possible to name a function (i.e., bind a function to an identifier) using `let`. Consider for example the successor function which just adds one to its argument:

```
let s = fun x -> x + 1
```

Because it is tedious to always write the '`fun`' and '`->`' there is also a shorter form available in OCaml. The above function for instance could equivalently be written as

```
let s x = x + 1
```

```
1 let w = 1
2
3 let x =
4  let y = w in
5  let w = 2 in
6  let z = w in
7  y + z
```

Listing A.2: Scoping

However, the possibility to write down anonymous functions is sometimes very useful.

Another special case is when you want to refer to a bound name directly in its body. Writing

```
let f x = if x < 1 then 0 else x + f(x-1)
```

where the goal is to define a function that adds the integers $x, x-1, \ldots, 0$, will result in an error message complaining about an 'unbound value f'. Even worse, if the name `f` was already bound to another value there will be no error message but the result might not be as expected.

```
let f x = x
let f x = if x < 1 then 0 else x + f(x-1)
```

will result in `f` being defined as if one had written

```
let f x = if x < 1 then 0 else x + (x-1)
```

To achieve the desired goal the modifier **`rec`** has to be added after **`let`**.

```
let rec f x = if x < 1 then 0 else x + f(x-1)
```

A.5.2. Scoping

For the sake of demonstration, consider the rather stupid example of Listing A.2. In line 1, `w` is bound to the value `1`. In line 3, an expression is started that will finally give the value for `x`. In line 4, `y` is bound to `w` (and since `w` is the same as its value to `1`). In line 5, a *new* variable which has coincidentally the same name as the first declared variable is bound to the value `2`. Hence, in line 6, `w` refers to the last variable by that name and therefore `z` is bound to `2`. The `y + z` in line 7 has value `3`. From line 9 on, `x` is bound to `3`. Notice however that the old `w` has neither been erased nor changed its value. As soon as the scope of the new `w` is left (as would be in line 8) every occurrence of `w` corresponds to a value of `1`.

A.5.3. Infix Operators

The usual infix (i.e., written between their two arguments) operators are provided by OCaml. Some examples are '`&&`' for the logical conjunction of two values of type `bool`, '`>`' to check for arbitrary values (of the same type) whether the left one is greater than the right one, '**`mod`**' for computing the remainder of integer division, etc. Every infix operator can also be used in prefix notation (i.e., in front of its two arguments) provided that it is enclosed in parentheses. For instance `(+) 1 2` can be written instead of `1 + 2` (it has the same effect as if first defining a new function **`let`** `f x y = x + y` and then calling `f 1 2`). This notation is also used when presenting the type of an infix operator. E.g., the type of addition would be given as

```
(+) : int -> int -> int
```

Note that in the special case of '`*`' (multiplication) it is necessary to put a space between the opening '`(`' and the '`*`' since '`(*`' starts a comment in OCaml.

A.5.4. Patterns

A very convenient feature in functional programming is *pattern matching*, i.e., checking whether a given value *matches* a certain *pattern*. A pattern p is defined by the following BNF:

$$
\begin{aligned}
p ::= \quad & x && \text{(A.1)}\\
\mid \quad & \mathit{const} && \text{(A.2)}\\
\mid \quad & C\ \texttt{(}p\texttt{,}\ldots\texttt{,}p\texttt{)} && \text{(A.3)}\\
\mid \quad & p\ \texttt{as}\ x && \text{(A.4)}\\
\mid \quad & p\texttt{,}\ldots\texttt{,}p && \text{(A.5)}\\
\mid \quad & \texttt{(}p\texttt{)} && \text{(A.6)}\\
\mid \quad & p\ \texttt{|}\ p && \text{(A.7)}
\end{aligned}
$$

where (A.1) is called a *variable pattern* (matching anything and binding it to the name x), (A.2) is called a *constant pattern* (matching the given constant, e.g., `true`, `42`, `3.1415`, `()`, etc.), (A.3) is called a *constructor pattern* or *variant pattern* (matching every constructor C if the argument matches the given pattern), (A.4) is called an *alias pattern* (matching everything that p would match and binding it to the name x), (A.5) is called a *tuple pattern* (matching a tuple where each entry matches p), (A.6) is called a *parentheses pattern* (matching exactly the same as p), and (A.7) finally is called a *choice pattern* (matching *either* the first *or* the second pattern, where the first is tried first). Patterns can be used in different places of OCaml programs which will be mentioned later.

A.5.5. Control Structures

Several *control structures* are available in OCaml. The first to mention is the *semicolon*

```
(;) : unit -> 'a -> 'a
```

where e_1 `;` e_2 first evaluates e_1 (which has to be of type `unit`), then evaluates e_2, and returns the value of e_2.

The construct `if` ... `then` ... `else` ... (sometimes called *conditional*) can be seen as a function

```
ite : bool -> 'a -> 'a -> 'a
```

i.e., taking three arguments, the first having type `bool` and the other two of type `'a`, and returning a value of type `'a`. However there is one thing special about the evaluation order. In `ite` e_1 e_2 e_3 the expression e_2 is only considered if e_1 evaluates to `true` whereas the expression e_3 is only considered if e_1 is `false`. In every user-defined function of the same shape all three expressions would be reduced to a concrete value before checking whether e_1 is `true` or `false`. Particularly useful along with conditionals are Boolean operators. Those are

```
(&&) : bool -> bool -> bool
```

which is `true` if both arguments are `true` and `false` otherwise,

```
(||) : bool -> bool -> bool
```

which is `true` if at least one of the arguments is `true` and `false` otherwise, and

```
not : bool -> bool
```

which is `true` if its argument is `false` and `false` otherwise.

Case expressions lastly choose different possibilities in programs by pattern matching. The syntax is

```
match e with
 | p1 [when cond1] -> e1
 ⋮
 | pn [when cond2] -> en
```

meaning that if the expression e matches pattern p_1 (and satisfies condition $cond_1$) then e_1 should be chosen, if it matches pattern p_2 (and satisfies condition $cond_2$), e_2 should be chosen, etc. At most one branch of a case expression is chosen. OCamls will always choose the first matching pattern where the condition is satisfied. Note that every of the expressions e_1 to e_n has to be of same type. The conditions $cond_1$ to $cond_n$ are called *guards*. Most of the time we will not need guards; hence it is allowed to skip them (as indicated by the square brackets).

The function `f` from above could alternatively be defined as (except that the new definition behaves differently on negative input; do you see why?)

```
let rec f x = match x with 0 -> 0
                         | n -> n + f(n-1)
```

and since similar patterns occur that often in function definitions there is even a shorter possibility, namely:

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

where `function` tells the compiler that there is one argument (which does not need a name since it is only used in pattern matching) that should be matched against the given patterns.

A.6. The Standard Library

The standard library of OCaml consists of a bunch of *modules* that are available in every OCaml program. A function of a module is identified by preceding its name with the module name followed by a dot. E.g., if you want to use the `length` function of the `List` module, you have to write `List.length`. A description of the standard library is contained in [9].

A.7. The Core Library

The OCaml core library consists of declarations for basic types and exceptions, plus the module `Pervasives` providing operations on them. The special thing about this 'core' is that it is available in every source file as if `open Pervasives` would have been the first line (i.e., unqualified names can be used, e.g., `fst p` rather than `Pervasives.fst p` to extract the first component of the pair `p`).

A.8. Exercises

Exercise A.1. Give at least one value of each basic type and the types `coord`, `direction`, and `number`.

Exercise A.2. Give at least five examples of mathematical functions that would have the type `int -> int -> int` in OCaml.

Exercise A.3. Write down the type of '`let f x = x - 1`' and justify your answer.

Exercise A.4. Explain (in words) what the type `string -> char -> bool` means. Can you imagine any meaningful function having this type?

Exercise A.5. Give a function which has type `'a * 'b -> 'a`. Give a function which has type `'a * 'b -> 'b`.

Exercise A.6. Write down an arbitrary recursive function in OCaml. (Note that any function that calls itself in its body is recursive.)

Exercise A.7. Consider the user-defined type for lists

```
type 'a mylist = Nil | Cons of ('a * 'a mylist)
```

Which of the following items are patterns?

a) `x`

b) `_`

c) `3.14`

d) `Nil`

e) `Cons(x,xs)`

f) `Cons(x,Cons(y,xs))`

g) `n,Cons(x,xs) as ys`

h) `(n,Cons(x,xs)) as (_,ys)`

i) `(n,Cons(x,xs)) when n < 0`

j) `(n,Cons(x,xs)) | (_,Nil)`

k) `(n,Cons(x,xs)) when x < 0 | (_,Nil)`

Hint: Also try them in the interpreter by replacing the dots in:

```
let foo = function | ... -> ()
```

Exercise A.8. Define the functions

```
first  : 'a * 'b * 'c -> 'a
second : 'a * 'b * 'c -> 'b
third  : 'a * 'b * 'c -> 'c
```

that extract the first, second, and third component of a triple, respectively.

Exercise A.9. Use pattern matching to define a function `turn` that changes a given `direction` to its opposite.

Exercise A.10. Using pattern matching, define a function `square` that computes the square of a `number`.

Exercise A.11. Use pattern matching to define a function `add` that adds two `number`s.
Hint: The function `float_of_int : int -> float` (of module `Pervasives`) might be useful.

Exercise A.12. Consider the algebraic datatype

```
type nat = Zero | Succ of nat
```

for natural numbers. Use pattern matching on the first argument to define a function `add` that adds two `nat`s.

Hint: The equations `Zero + m = m` and `(Succ n) + m = Succ (n + m)` might be helpful.

Exercise A.13. Consider the algebraic datatype

```
type shape = Rect of float * float | Square of float | Circle of float
```

where `Rect(a,b)` represents a rectangle of length a and width b, `Square` a is a square of side length a, and `Circle` r denotes a circle with radius r.

Use pattern matching to define the functions `circum : shape -> float` computing the circumference and `area : shape -> float` computing the area of the given geometric shape.

Hint: You may use $\pi = 3.14$.

Exercise A.14. Use recursion to define a function `gcd : int -> int -> int` computing the greatest common divisor of two integers.

Hint: Use Euclid's algorithm exploiting that for non-negative integers

`gcd m 0` $=$ `m`, `gcd m m` $=$ `m`, and `gcd m n` $=$ `gcd (m - n) n` (if m $>$ n).

Exercise A.15. Consider the following recursive definition for computing binomial coefficients:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n; \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise;} \end{cases}$$

for $n \geq k \in \mathbb{N}$. Define a recursive function that computes binomial coefficients. Make sure your function terminates for all possible arguments.

Exercise A.16. List five claims (positive or negative) the following article makes about OCaml:

Yaron Minsky, OCaml for the masses, Comm. of the ACM 54:11, 2011.

Exercise A.17. Implement a function `c : int -> int` which computes the following:

$$c(n) = \begin{cases} n & \text{if } n \leqslant 1 \\ c(3n+1) & \text{if } n > 1 \text{ and } n \text{ is odd} \\ c(\frac{n}{2}) & \text{if } n > 1 \text{ and } n \text{ is even} \end{cases}$$

At each recursive call print the value of `n`.

Trivia: This function is known as *Collatz'* or *Syracuse* function.

Bonus: Prove that for `n` greater 2 this function always ends with the values $4, 2, 1$.

B. Automatic Compilation of OCaml Projects

This is a very basic introduction to `ocamlbuild` which should however suffice for the needs of this lecture. In general `ocamlbuild` is called as follows

```
$ ocamlbuild <target>
```

where the identifier `<target>` tells `ocamlbuild` what to do.

B.1. Targets

In the following some of the most common *targets* are shortly described. Notice that `ocamlbuild` creates the directory `_build` and the file `_log` in the current directory, where the former contains all generated files (object files, executables, etc.) and the latter a transcript of all actions that have been taken.

B.1.1. Bytecode Executables

To generate a bytecode executable whose main function is in the file `<main>.ml`, the following command line does the job:

```
$ ocamlbuild <main>.byte
```

The result is a link named `<main>.byte` that points to the bytecode executable by the same name residing in the `_build` directory. E.g., the program of Listing A.1 could be compiled by

```
$ ocamlbuild helloWorld.byte
```

By adding `.d` before `.byte` (i.e., `.d.byte` the compiler can be informed to put additional debugging information into the bytecode executable (to use, e.g., with `ocamldebug`). Debugging is also necessary to enable stack backtraces. Additionally the environment variable `OCAMLRUNPARAM` has to be set to `b` (for backtrace). E.g., add the line

```
export OCAMLRUNPARAM=b
```

to your `.bashrc`. To compile above program for debugging use

```
$ ocamlbuild helloWorld.d.byte
```

Bibliography

[1] Haskell – A purely functional programming language. The Haskell site is available online at http://www.haskell.org; visited on January 14th 2008.

[2] Henk P. Barendregt and Erik Barendsen. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[3] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, second edition, 1998. ISBN 0-13-484346-0.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 978-0-262-03384-8.

[5] Anthony J. Field and Peter Harrison. *Functional Programming*. Addison-Wesley, 1988. ISBN 0201192497.

[6] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[7] Daan Leijen. *Parsec, a fast combinator parser*, October 2001.

[8] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[9] Xavier Leroy. *The Objective Caml system release 3.12*, July 2011. http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12-refman.pdf.

[10] Bruce J. MacLennan. *Functional programming: practice and theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-13744-5.

[11] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science (JCSS)*, 17:348–374, 1978.

[12] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN 0521663504.

[13] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996. ISBN 9780521565431.

[14] Fethi Rabhi and Guy Lapalme. *Algorithms: A functional programming approach*. Addison-Wesley, 1999. ISBN 0201596040.

[15] Ivan Stojmenovic. Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. 2000.

[16] Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley, 1996. ISBN 0-201-40357-9.

Index