Functioneel Programmeren

2015–2016

J.D. Fokker
J. Hage (J.Hage@uu.nl)
S. Holdermans
A. Löh
S.D. Swierstra

August 14, 2015

Contactadres: Jurriaan Hage, Informatica-instituut,
Postbus 80089, 3508 TB Utrecht, e-mail `J.Hage@uu.nl`.

1e druk (informatica-versie) september 1992
2e druk (informatica-versie) februari 1993
3e druk (informatica-versie) september 1993
4e druk (informatica-versie) september 1994
5e druk (informatica-versie) september 1995
6e druk (CKI-versie) oktober 1995
7e druk (informatica-versie) september 1996
8e druk (CKI-versie) oktober 1996
9e druk (CKI-versie) oktober 1997
10e druk (CKI-versie) oktober 1998
11e druk (CKI-versie) november 2002

12e druk (informatica-versie) februari 2003
13e druk 2004
14e druk (informatica+CKI) februari 2005
15e druk (informatica) januari 2006
16e druk (informatica) januari 2007
17e druk (informatica) februari 2008
18e druk (informatica) januari 2009
19e druk (informatica) oktober 2009
20e druk (informatica) september 2010
21e druk (informatica) september 2011
22e druk (informatica) augustus 2012
23e druk (informatica) augustus 2013
24e druk (informatica) september 2014
25e druk (informatica) september 2015

Contents

1 Preface and reading guide

1.1 A new language; a new paradigm

Programming language technology has been an active research area from the beginning of the use of computers; the very first programming languages such as FORTRAN (Formula Translator, nowadays called "Fortran") had a very simple structure, and their mapping to machine code (a language that computers directly understand) was straightforward so that compilers could be simple, small and fast. Also the programmers using these languages had written machine code before and had a good idea of how this mapping worked. Since machines were relatively small and slow with respect to current standards, and besides that incredibly expensive, emphasis was put on creating small and efficient code.

Over the years however the so-called *semantic gap* between the expressivity of the programming language and the generated machine code has widened, and most programmers nowadays have no idea what such machine code may look like. The process of programming has changed over the years too: the focus of programming has shifted from the production of highly efficient code at any cost to expressing one's thoughts as effectively as possible, and leaving the generation of efficient code to clever compilers. Although efficiency still plays an important role, many other aspects have gained importance. We mention a few:

- How much effort is involved to get a reasonably efficient program?
- How easy will it be to change the program once its original specification is changed?
- How much can be re-used from library code?
- How well can the program be statically checked for inconsistencies so that programming errors can be detected by a compiler before it goes in production?
- How easy can a program be tested?
- How hard is it to formally verify the correctness of the code?
- How well is the programming language and its compiler supported by the operating system and programming environment?
- Will my company be able to hire programmers in sufficient numbers once the product becomes a success?

As a consequence of programs getting larger and larger, the emphasis on being able to re-use code has increased, and various questions relating to how well re-usability is supported by a given programming language have gained in importance:

- Can we easily re-use existing code?
- How can we write code such that it can be easily re-used
- How can we ensure that the result of the composition of applications out of components makes sense?
- How can we avoid to pay a too high price for the overhead such composability always carries?
- And, how can we be sure that code written by someone else is indeed doing what it is supposed to do?.

Programming languages are often categorized based on the programming paradigm that they most adhere to. When we talk of *structured programming* we refer to the ability to write programs that are divided up into smaller pieces (calling these pieces functions, definitions, procedures, methods, or whatever name you fancy). Almost all programming languages in use today follow the structured programming paradigm.

The *imperative paradigm* allows (maybe we should say "expects" or "demands") that programmers specify in detail the order in which operations in the program should be performed. Most languages you have heard of are considered imperative languages, including C, C#, Java, PHP, Cobol, FORTRAN. In the *object-oriented paradigm*, data and operations are grouped together, and typically support concepts such as inheritance, subtyping and overloading; Java, C++, C#, Ruby and Python are well-known examples. As you may have already decided for yourself, paradigms need not be exclusive: an object-oriented language may also be an imperative one. In fact, most languages in use today are a mix of concepts that arise from multiple, sometimes many, paradigms.

In the course on Imperative Programming (or a course much like it) you have been taught the basics of the imperative and object-oriented paradigm. The essential aspects of these paradigms are illustrated within the language C#. In this course, we shall be looking at concepts that have historically arisen within the *functional programming paradigm*, and we shall illustrate these concepts in the context of the purely functional programming language called Haskell.

1.2 The pure, functional programming paradigm

Imperative languages have as common property that a program consists of a sequence of commands, and that the execution of such commands changes the global state, e.g., the values of variables, and the contents of files on disk. The primitive command underlying all other commands is the *assignment statement* which changes the state of a variable by

assigning a new value to it by overwriting the old value. It is precisely this assignment statement which makes it more difficult to reason about programs, to abstract from a specific part of the program, and to compose a new program out of a collection of fragments.

Let us take as an example the following pseudo-code:

```
let x = getChar
in if x == x then print "True" else print "False"
```

where getChar is a function that returns the next character typed in by the user.

Now everyone, including the authors of these lecture notes, would expect this program to print `True`. Let us now take the definition of x and expand it literally at its use sites:

```
if getChar == getChar then print "True" else print "False"
```

For this program one would in general expect it to print `False`: each occurrence of getChar is executed once and this execution has, besides returning the next character from the input, as *side-effect* that the global state changes, i.e. the input pointer is advanced by one. We thus observe that the presence of side-effects disallows us to replace an expression by another even though they are equal by definition (here we replace x by getChar). Languages in which we may safely replace the uses of x by its definition are called *referentially transparent*. Or, the other way around, referential transparency allows us to give a single name to identical subprograms which occur at various places in the program. It is the absence of this property which makes it difficult to abstract from an arbitrary piece of program text (i.e. to give a name to any expression), without running the risk of changing the meaning of the program.

In functional languages we do not express ourselves using commands, but – as the name says – only with functions. Sometimes we encounter the phrase *pure functions* to emphasise that such functions *do not have side-effects*, i.e. do not change the global state. Now you may ask yourself the question "But if we do not have assignments, how can we then have loops?". The answer lies in the use of *recursion*, which is the term used for functions calling (either directly or indirectly) themselves again.

Let us take a look at the following imperative program :

```
function sum_upto (n) = {
  i := n;
  sum: = 0;
  while i > 0 do {
    sum := sum + i;
    i := i − 1
  };
  return sum
}
```

This function can be transformed directly into a recursive, assignment-free equivalent:

```
function sum_upto (n) =
  let function loop (i, sum) = if i == 0 then return sum
                                         else return (loop (i − 1, sum + i))
  in return (loop (n, 0))
```

So how does this last function deal with the absence of assignments? In both programs, assuming that n is at least 0, i takes on all values between 0 and n. But in the imperative program the location for i in memory is a single location that is updated after every loop. In the recursive program, the effect of calling the recursive loop function is to allocate a new piece of memory for storing the updated values of i and sum. So the values in memory locations are never updated, new ones are allocated instead. By doing so, you do not run the risk of inadvertently changing the value of a variable of which you still needed the old value (making this a source of subtle bugs in imperative programs), but, on the other hand, allocating many memory cells for all the different values of sum and i is a potential source of inefficiency. This is why functional languages depend strongly on a powerful optimizing compiler to reduce this overhead whenever possible.

Not all functional languages are pure. Many, including LISP, Scheme and ML, take a hybrid approach: they are heavily based on functions, but nevertheless allow assignments in programs too, thereby sacrificing full referential transparency. What is the same for all functional languages, pure or not, is the focus on writing programs as a collection of functions being passed around to, and returned from other functions, and a general discomfort with the use of assignments and other implicit side effects.

Over the last few decades, languages in the functional programming paradigm have received much attention in the academic world. This has resulted in the development of new concepts for programming, and techniques for effectively implementing them. Many of these concepts have found their way into mainstream languages such as Java and C#. For example, *parametric polymorphism* (often called *generics* in the context of object oriented languages) was added to Java in version 1.5. And *anonymous functions* (or *lambda*'s) are part of Java since version 1.8.

The development of purely functional languages dates back to even before the development of imperative languages by Von Neumann. The mother of all functional languages, the λ-calculus, was introduced as part of the research into the concepts of *computability* and *provability*, in the beginning of the previous century. For a precise characterisation of the collection of all possible computations we find on the one hand the abstract imperative machines (called Turing-machines in honor of its inventor) as proposed by Alan Turing, and on the other hand the functional models for computation a developed by M. Schönfinkel (in Germany and Russia), Haskell Curry (in England) and Alonzo Church (in the USA); it has been proven that in terms of expressive power these formalisms are equivalent, in the sense that every program for a Turing-machine can be transformed into the λ-calculus and vice versa. The *Church-Turing* hypothesis states that every computable function can be computed within either of these models, and that the functions

which can be computed by these models are precisely the computable functions. However, having universal expressive power is not sufficient to become a good programming language.

1.3 A short tour of Haskell

In this course, we have chosen Haskell as the main programming language. There are various reasons. The main reason is that it has many features that are worthy of study. It is also the most widely used pure functional language, has an active research community and a rapidly growing collection of users. Over the years a lot of work has been invested into the development of its main compiler, the Glasgow Haskell Compiler (*GHC*), extending the language with quite a lot of new features. In recent years over 2000 library packages have become available though the *hackageDB* site (see `http://hackage.haskell.org`), containing implementations in Haskell for a wide class of problems.

Haskell is also used by more theoretically oriented (mathematically inclined) people to provide an operational basis to the abstractions they have invented and work with. As such Haskell has become the language of discourse in many papers about programming language research. This can be witnessed by taking a look at the proceedings of the major ACM-conferences such as the *Principles of Programming Languages* (POPL) series of conferences.

We now provide a quick overview of the main concepts of Haskell. All will be considered in much more detail in the remainder of these lecture notes.

1.3.1 Haskell is purely functional

In 1977 John Backus, who headed the team which designed Fortran and constructed its first compiler at IBM and who gave his name to the Backus-Naur Form (BNF), delivered his ACM Turing award lecture [1] titled *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*, which can be seen as one of the first points in the history of Computing Science where it was made clear to a wider audience that there might be more than imperative programming alone, and that programs could be looked upon as formal objects, which should have a nice notation, and which could be manipulated like we manipulate mathematical formulae when we are doing calculus. As the title says the lecture tries to answer the question how to get rid of the assignment statement.

As we have said before the concept of an assignment is absent from Haskell. As a result, the style of programming is completely different from what you may be used to, and this may at first be hard. But, it may well be that once you become accustomed to the fact that you can freely abstract from almost everything, you will also want to do this when

writing programs in other languages that you know. In your everyday way of writing imperative programs you are likely to be much more aware of the fact that every now and then state changes.

In the object-oriented paradigm, the program manipulates stateful objects, which often correspond to real-world objects. This approach can be quite natural, since the objects which are modelled also change state in the real world. Since even Haskell programs want to have some effect on the world (printing to screen, writing to disc), we shall show how to obtain these effects in a purely functional language through the use of monads.

When we ask an everyday programmer to describe the precise effect of an assignment statement the answer will often be that once the assignment statement has been executed the variable has gotten a new value. It is then easy to gloss over another important effect: *the old value is no longer accessible.* In fact by updating a variable the programmer has indicated explicitly that the old value is no longer needed; thus he has explicitly encoded part of the garbage collection process in his program.

When we schedule values which mimic values in the real world this is not such an issue, since we have a real world model equivalent to think about our state changes. For more algorithmic parts the mental overhead associated with keeping track of which values are still needed and which ones can be discarded, can make programming extremely difficult and error prone. As a result, much time may be wasted debugging imperative programs in which this explicit scheduling is not sufficiently well thought out.

By insisting on referential transparency, Haskell takes away a large class of programming errors. Fortunately modern compilers are vey good at discovering when values are no longer needed, and the speed of modern machines makes that we can, more than in the past, afford to pay for the extra security and programming convenience.

1.3.2 Lazy evaluation

Since we do not have assignments, we do not have side effects. So for a functional call `f arg1 arg2` the order in which the arguments are evaluated cannot make any difference to the outcome of the program. This has made it possible to take a very radical approach here, i.e. in Haskell the arguments *are not evaluated at all at the place of the function call*; instead they are evaluated when their value is needed for the first time (this is sometimes referred to as call-by-need) during the evaluation of the function body, or bodies of functions called from this body. Once the argument has been evaluated however the resulting value is recorded so that the next time the parameter is referred to, the result of the first and only evaluation is still available. This sharing of values, together with the call-by-need strategy, is jointly often referred to as *lazy evaluation.*

Although this evaluation strategy seems only to differ minimally from what is known from more conventional languages, having lazy evaluation has deep implications. We will see that it can provide very elegant formulations for otherwise complicated algorithmic

problems. Of course, lazy evaluation comes with its own set of problems, which we have to become aware of. For example, lazy evaluation induces a certain amount of overhead, since whenever an expression is evaluated it has to be checked whether it has been evaluated before or whether we are touching it for the first time. Research in (static) program analysis has made a large contribution to making such dynamic tests largely superfluous. It does however not solve all problems.

To give an impression of the new possibilities lazy evaluation brings us we take a look at the following Haskell definition:

$$\mathsf{ones} = 1 : \mathsf{ones}$$

In this expression the : stands for the operator which constructs a new list, using its left hand operand as the head of the new list and its right hand operand as its tail. Without lazy evaluation the call to the : operator would start by evaluating the right hand side operand (ones), which would lead to a call to itself, thus leading to a non-terminating evaluation. With lazy evaluation, the result however is for the time being a list of which we know that it is not empty, and that its head equals 1. As long as we do not ask for its tail, nothing happens; only if we ask whether the tail is e.g. non-empty a little bit of further evaluation takes place, exposing a further call to the : operator. Thus the value ones represents the infinite list of 1's, or more precisely, a prefix of this infinite list which is sufficiently long that we cannot see the difference. Of course, it is not wise to try to compute the length of this list, since this computation will not terminate.

1.3.3 Type inference

Haskell is a so-called *type-safe* language: the type system guarantees that no bit pattern in memory can be interpreted in a way which differs from its intended use. This is sometimes phrased with the term *"Well-typed programs do not go wrong"*. There is no way we can write out a string value somewhere, and then read it back and use this bit-pattern as a pointer. As a consequence no pointer arithmetic is possible. (Pointer arithmetic is a programming idiom found in the language C and has been a major source of bugs and security leaks for programs written in that language; for that reason pointer arithmetic has been intentionally omitted from Java and C#.) Thus each variable has a type associated with it, and this type describes what we can do with the value bound to the variable. No other ways of handling the bit pattern bound to the variable are available.

Haskell's type system is also *statically checked*. By this we mean that the compiler checks the type safety of the program before machine code is generated and executed. By contrast, a dynamically checked language (amongst which many scripting languages such as Python and JavaScript) performs these verifications while executing the program, which yields a considerable overhead and makes it very hard to locate type errors.

An advantage of such dynamically typed languages is that one does not have to explicitly assign types to variables, since the values carry their type with them. As a consequence,

programs written in such a language can be much shorter. Fortunately Haskell can be just as short since Haskell has so-called *type inference*, which makes it possible to statically type-check a large class of programs, without having to provide any type explicitly. Static type checking in combination with type inference gives us the best of both worlds: a firm guarantee that the program will not misbehave, while not having to write too much additional code. In a certain sense the compiler provides a partial correctness proof of the program, without too much assistance from the programmer.

Suppose we want to write a program which swaps to integers. A prototypical example in an imperative language may look like:

```
procedure swap (var i, j : Integer);

var h : Integer;
begin
  h := i;
  i := j;
  j := h
end
```

Looking at this code we notice the following. In the first place, the type of this function does not provide us with a great deal of information. There are many, many functions which take two *Integer* variables as parameter. So there is no way in which the type of the function here guarantees that actually values will be swapped. A second observation is that when we want to swap two characters, we have to make a copy of the function, with all the *Integer*'s replaced by *Char*'s.

Now let us take a a look at our first attempt to write an Haskell equivalent. Since we cannot update values, we have to resort to a function which takes a pair of values, and returns a pair with those values swapped:

```
swap :: ∀a . (a, a) -> (a, a)
swap (x, y) = (y, x)
```

Here we have first specified the type of swap, followed by its definition on the next line. One may now wonder how many functions one can write which have this type. A bit of reflection shows that there are four:

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

If we make a random choice out of these definitions we already have a 25% chance that we pick the right one. The type system thus has already made a small contribution in safeguarding us.

The $\forall a$ in the type states that the function works for any type a, be it *Integer*, *Char*, a list of values or even a function. This takes away the need to write a large collection of such swapping functions. We say that Haskell's type system is *parametrically polymorphic* [2]. Because we know that the function has to work for any type a this also implies that we cannot write functions like:

$$\mathsf{swap}\ (\mathsf{x}, \mathsf{y}) = (\mathsf{x} + 1, \mathsf{y} - 1)$$

since operators like + and − are not defined for every type.

Now, what happens if we write the function without specifying its type and then ask for the type inferred by the compiler? When doing so using the interactive Haskell interpreter `ghci`, we get:

```
let swap (x, y) = (y, x) -- define the function swap
:t swap                  -- ask for its type
forall a b . (a, b) -> (b, a)
```

(where `forall` is how `ghci` visualizes ∀)

As it happens, the automatically inferred type is even more general than the one we chose! We can even swap two values of which the types differ; swapping a value of type $(\mathit{Integer}, \mathit{Char})$ returns a value of type $(\mathit{Char}, \mathit{Integer})$; remarkably, the function swap is the only function with this type and thus the type provides a *precise* specification of what the function does.

1.3.4 The type class system

Haskell also supports a way of *overloading* function names, i.e., to have several different definitions co-exist which have the same name. The type system picks the one which applies at a specific place. Unfortunately, the mechanism which takes care of it are called (type) classes in Haskell because they remotely resemble something like classes in object-oriented languages. The supported mechanism however differs radically, and the word **class** generally just increases confusion.

Originally the class system was added to the language with the goal of keeping the language small and to be able to use e.g. + for both the addition of integers and floating point numbers. Over the years the class system has seen many extensions. In this course, we shall focus on the core aspects of type classes.

1.4 What does it entail to learn a new programming language?

1.4.1 Grammar

Learning a new programming language involves a number of activities. In the first place you will have to learn grammar; once you know the grammar of a language you can form correct sentences. Fortunately you do not have to learn the complete grammar before you can start using the language. Haskell was originally defined in the Haskell98 report, and currently we use the Haskell2010 specification as our starting point. For most of what you shall be seeing in this course, the Haskell98 and Haskell2010 specifications are in agreement.

As does this course, many courses teach a new language in a stepwise fashion, gradually introducing new concepts. Keep in mind that Haskell is a large language, with many subtle corners. A dilemma we are often faced with is the following: "Should we tell precisely how things are defined, or is it sufficient for the time being to give an approximation of the truth, without explaining the underlying structures in detail?". We have sometimes chosen for the one and sometimes for the other; anyway, do not be surprised if later in these notes we come back to what we have said before, and add some further detail, or expose the underlying design layer.

Another thing you will see is that many things that are defined as part of the grammar in other languages, are actually defined through libraries in Haskell. We can do so since Haskell is a very powerful language, with extensive features for building advanced libraries. From a program it will not always be clear whether something is part of a library or part of the grammar of the language. In such cases, you should not worry too much. Things will become clearer later on.

1.4.2 Lexicon

Besides grammar you have to build your own vocabulary. In programming languages this means: you have to learn what can be found in libraries, and where to find those libraries. When programming it is a good strategy to look every now and then to see whether there is not a library that actually solves your problem already. Studying the code and the organisation of such libraries is an excellent way to learn how to program in Haskell. Clearly, as you still have to learn to program in Haskell yourself, there is a limit to what you may borrow from the Internet, and Hackage in particular. When in doubt, when working on an assignment, ask the assistants during practice hours.

1.4.3 Idioms

Every programming language has special ways of "how to say things". It is here that you will be most surprised, since the way things are done in a lazily evaluated, strongly

typed, purely functional language may be quite different from what you have been used to thus far. This is also one of the things that are discussed most on the mailing lists, where many treads start like "I tried to do this and that in a such and such a way, but am I right in pursuing this path, or is there a completely different way of doing this". Often the answer to both these questions is an affirmative "yes". Experience will teach you to appreciate the different ways of attacking a problem, and learning how to compare and judge them. In the end you will build up a toolbox for yourself, containing a collection of such idioms, and you will apply them automatically.

1.4.4 Style

As in any other language, there are different styles of programming. To give one example, certain programmers try to avoid the use of names as much as possible. They like to see their programs as mathematical formulae, to which they can apply transformations without having to know why they can do so. Consider the following two versions of the same function mapBoth, which applies first a function f to all elements of a list xs, and then a function g to all elements of the resulting list. We start with the verbose version, which used an important function, map, that takes two arguments, a function and a list, and applies the function to the elements of the list:

```
mapBoth :: ∀a b c . (b -> c) -> (a -> b) -> [a] -> [c]
mapBoth g f xs = let ys = map f xs
                     zs = map g ys
                 in  zs
```

Those who do not like to invent new names for intermediate values, may however prefer the following version:

```
mapBoth f g xs = map g (map f xs)
```

Others may like to use function composition denoted by a ., as in:

```
mapBoth f g xs = (map g . map f) xs
```

or even shorter:

```
mapBoth f g = map g . map f
```

An advantage of the last two formulations is that we immediately see that we can write a more efficient version of this function by applying the law map g . map f == map (g . f), so we get:

```
mapBoth f g = map (g . f)
```

Although the operator . is denoted as an infix operator, we can also write it as a prefix operator, and then write:

```
mapBoth g f =       map   ((.)  g    f)
mapBoth g f = (     map . ((.)  g)) f
mapBoth g =         map . ((.)  g)
mapBoth g =   ( (.) map   ((.)  g)
mapBoth   =   (((.) map) . (.)) g
mapBoth   =   ( (.) map) . (.)
```

Ironically, this last version, which is full of dots, is called a program in the so-called *point-free style* (a term coming from topology, where a variable refers to a point in a space; here we could also have said "variable-free").

The programs you shall be writing and submitting as part of the course will be judged for style. The website has some guidelines on what we expect from you in this regard. Be aware that they are guidelines: you can always come up with situations in which a guideline should not be followed. Part of the challenge in learning to program in Haskell well, is to attain an understanding when to follow a given style, and when not.

1.5 About this reader

As you will have seen from the front page of these lecture notes, quite a few people have contributed to it, and some of the material in the lectures is quite old. The older material was written in Dutch, while newer material is typically written in English, with an eye to internationalization, and the possible presence of master students from abroad that are not familiar with our language.

The contents of the current version of the lecture notes is similar to the one used for the FP course in course year 2013/2014 and before. However, you should be aware that much of the contents has been reorganised: chapters and sections have been moved, chapters have been split up, and merged, and so on. Please do take this into account.

Distinctive changes with respect to the lecture notes of 2013/2014 are:

- The material on monads written by Graham Hutton that was part of the side material in 2013/2014 has been included as a chapter in the current reader. (With his permission, of course. Thank you, Graham!)
- With the exception of writing λ for \ in Haskell code, all code in this reader should be exactly as it should be for a Haskell program: all formatting niceties by lhs2TeX that are normally present in academic papers that involve Haskell have been undone.
- Chapters and sections on I/O, and expression data types, have been moved around to fit the current order of presentation.
- Most material not discussed in the course has been deleted from the reader. This includes some of the appendices.

- New material was added on (embedded) domain specific combinator languages.

Changes with respect to the lecture notes of 2014/2015 are largely cosmetic, correcting typo's and enhancing consistency. You can use the lecture notes of 2014/2015 without any problems.

Happy programming,
Jurriaan Hage

2 Functioneel programmeren

2.1 Functionele talen

2.1.1 Functies

In de jaren veertig werden de eerste computers gebouwd. De allereerste modellen werden nog "geprogrammeerd" met grote stekkerborden. Al snel werd het programma echter in het geheugen van de computer opgeslagen, waardoor de eerste *programmeertalen* de intrede deden.

Omdat destijds het gebruik van een computer vreselijk duur was, lag het voor de hand dat de programmeertaal zo veel mogelijk aansloot bij de architectuur van de computer. Een computer bestaat uit een besturingseenheid en een geheugen. Een programma bestaat daarom voor een flink deel uit instructies die het geheugen veranderen, en die door de besturingseenheid worden uitgevoerd. Daarmee was de *imperatieve programmeerstijl* ontstaan. Imperatieve programmeertalen (en dat zijn de meeste talen zoals Fortran, Cobol, Pascal, C en Java, Perl, Python etc.) worden gekenmerkt door de aanwezigheid van toekenningsopdrachten (*assignments*), die na elkaar worden uitgevoerd.

Wanneer we eens nagaan wat een toekenning (assignment) zoals $\mathsf{x} := \mathsf{y} + 1$ voor effect heeft, dan zullen de meeste mensen die wel eens geprogrammeerd hebben zeggen dat het gevolg van het uitvoeren van dit statement is dat de variabele x na afloop de waarde heeft van de variabele y, vermeerderd met 1. Er is echter meer gebeurd; zo is de oude waarde van x niet langer beschikbaar. De consequentie van deze laatste observatie is dat een programmeur in een imperatieve programmertaal zich dus voortdurend bewust moet zijn van welke waarden hij nog nodig heeft en welke waarden "vergeten" kunnen worden.

Ook voordat er computers bestonden werden er natuurlijk al methoden bedacht om berekeningen te beschrijven. Daarbij is eigenlijk nooit de behoefte opgekomen om te spreken in termen van een geheugen dat verandert door instructies in een programma. In de wiskunde wordt, in ieder geval de laatste vierhonderd jaar, een veel centralere rol gespeeld door *functies*. Functies leggen een verband tussen parameters (de "invoer") en het resultaat (de "uitvoer") van bepaalde processen.

Ook bij een berekening hangt het resultaat op een of andere manier af van parameters. Daarom is het gebruik van een functie een voor de hand liggende manier om een berekening te specificeren. Dit uitgangspunt vormt de basis van de *functionele program-*

meerstijl. Een "programma" bestaat in hoofdzaak uit de definitie van een aantal functies. Bij het uitvoeren van een programma wordt een functie van argumenten voorzien, en moet het resultaat berekend worden. Bij die berekening is nog een zekere mate van vrijheid aanwezig, en die kan door de interpreter (het programma dat het functionele programma daadwerkelijk uitvoert) worden gebruikt om zo handig en efficiënt mogelijk met geheugen en processortijd om te gaan. En waarom zou een programmeur immers moeten voorschrijven in welke volgorde onafhankelijke deelberekeningen moeten worden uitgevoerd? Of welke waarden onthouden moeten worden en welke niet langer nodig zijn? Door daar niet over na te hoeven denken, blijft er meer tijd en aandacht over voor andere zaken.

Met het goedkoper worden van computertijd en het duurder worden van programmeurs wordt het steeds belangrijker om een berekening te beschrijven in een taal die zo dicht mogelijk bij de belevingswereld van de mens aansluit. Een vertaler kan die formulering dan wel omzetten in een efficiënt, maar wellicht voor mensen tamelijk onleesbaar, imperatief programma, dat uiteindelijk door de machine wordt uitgevoerd. Functionele programmeertalen sluiten aan bij de wiskundige traditie, en zijn niet al te sterk beïnvloed door de concrete architectuur van de computer.

Dankzij langdurig onderzoek op het gebied van de vertalerbouw is het niet langer het geval dat programma's geschreven in functionele talen veelal ordes van grootte langzamer zijn dan equivalente programma's in imperatieve talen. Anderzijds gebiedt de eerlijkheid te zeggen dat wil men het uiterste uit een computer persen, het gebruik van een imperatieve taal meer voor de hand ligt. Gelukkig bevatten ook veel functionele talen manieren om in dergelijke noodgevallen, een imperatieve formulering te geven. Voor een voortdurende vergelijking verwijzen we naar een website waar voor een keur aan talen en vertalers programma's worden vergeleken: `http://benchmarksgame.alioth.debian.org/`. Hieruit blijkt dat de uitspraak "functionele talen zijn langzamer dan imperatieve talen" in zijn algemeenheid niet opgaat. Ook blijkt echter dat die versies van functionele programma's die eenzelfde snelheid halen als imperatieve talen wel vaak stevig onder handen zijn genomen, en de facto imperatieve programma's in vermomming zijn. Vaak kan echter met het aanpassen van een klein gedeelte van een programma volstaan worden teneinde een in executietijd vergelijkbare versie te krijgen. In sommige gevallen zijn functionele formuleringen essentieel sneller, en dienen de imperatieve varianten sterk herschreven te worden.

2.1.2 Talen

De theoretische basis voor het imperatief programmeren werd al in de jaren dertig gelegd door Alan Turing (in Engeland) en John von Neuman (in de USA). Ook de theorie van functies als berekeningsmodel stamt uit de twintiger en dertiger jaren. Grondleggers zijn onder andere M. Schönfinkel (in Duitsland en Rusland), Haskell Curry (in Engeland) en Alonzo Church (in de USA).

Het heeft tot het begin van de jaren vijftig geduurd voordat iemand op het idee kwam om deze theorie daadwerkelijk als basis voor een programmeertaal te gebruiken. De taal Lisp van John McCarthy was de eerste functionele programmeertaal, en is ook jarenlang de enige gebleven. Hoewel Lisp nog steeds wordt gebruikt, toont het toch zijn leeftijd.

Met het toenemen van de complexiteit van computerprogramma's deed zich steeds meer de behoefte voelen aan een sterkere controle van het programma door de computer. Het gebruik van *typering* speelt daarbij een grote rol, en de taal Haskell die we in dit dictaat gebruiken is dan ook voorzien van een zeer uitgebreid type systeem.

In de jaren 1980 scheidden zich de wegen van het Amerikaanse en het Europese publiek. In Amerika werden voornamelijk de Common Lisp en Scheme (een directe opvolger van Lisp) populair. Helaas zijn dit nog steeds talen zonder sterke typering; d.w.z. dat sommige categorieën logische fouten veelal pas tijdens het draaien van het programma worden ontdekt, en veelal is dat dus te laat. In Engeland waren de talen Miranda en Gofer invloedrijk. Binnen het bedrijf Ericsson werd de taal Erlang ontworpen die ook buiten dat bedrijf en de Scandinavische industrie belangrijk is geworden; deze taal is vooral bekend vanwege de uitgebreide mogelijkheden voor het bouwen van gedistribueerde en fouttolerante systemen. Een andere taal taal die aan populariteit wint, en ook de nodige functionele kanten heeft is Martin Odersky's Scala. Een Nederlands (Nijmeegs) product is Clean.

Om deze wildgroei in talen enigszins te beteugelen, heeft een aantal onderzoekers begin jaren 90 van de vorige eeuw het initiatief genomen om samen een taal te ontwerpen. Dit heeft in 1992 geresulteerd in de taal *Haskell*, waarvan de definitieve taaldefinite bekend stond als Haskell98. Deze is in 2010 vervangen door wat we Haskell2010 noemen.

Er bestaan inmiddels een aantal verschillende compilers van deze taal, waarvan de Glasgow Haskell Compiler (GHC) (en bijbehorende interpreter `ghci`) de de facto standaard is, en de compiler die we binnen dit vak gebruiken.

Andere bekende compilers zijn Helium (ontwikkeld aan Universiteit Utrecht, implementeert een klein deel van de taal, maar wel met betere foutmeldingen dan de standaardcompiler), en de Utrecht Haskell Compiler (UHC) die ook aan de Universiteit Utrecht wordt onderhouden, en een testbed is voor taal en compilerinnovatie.

Haskell is een omvangrijke taal, met een grote rijkdom aan concepten. Dit heeft het mogelijk gemaakt om veel zaken die in andere talen vast ingebouwd zijn, uit te drukken in de taal zelf; een gevolg hiervan is dat veel "standaardzaken" nu via bibliotheken beschikbaar kunnen worden gesteld.

```
$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
```

Fig. 2.1: Starting up ghci

2.2 De Haskell-interpreter

2.2.1 Expressies uitrekenen

Programma's in een functionele taal bestaan voor een groot deel uit functiedefinities. Deze functies zijn bedoeld om gebruikt te worden in expressies, waarvan de waarde uitgerekend moet worden. Om de waarde van een expressie met een computer te berekenen is een programma nodig dat de functiedefinities begrijpt. Zo'n programma heet een *interpreter*.

Voor de taal Haskell die in dit diktaat gebruikt wordt is een interpreter beschikbaar genaamd `ghci`. Deze interpreter wordt gestart door de naam van het programma, `ghci`, in te tikken, zie Fig. 2.1.

Met het commando :? krijg je een lijst van mogelijke commando's, die je ook veelal via menu's kunt selecteren (Fig. 2.2).

De interpreter `ghci` kent vele commando's (zie Fig. 2.2 tot Fig. 2.5). Deze commando's kun je o.a. gebruiken om een programma in de interpreter te laden; maar dat doen we nog even niet, omdat de interpreter standaard al een hele verzameling definities ingelezen heeft, samen *Prelude* genoemd. Aan de prompt is ook te zien dat dit de omgeving is waarin gebruikte namen opgezocht worden. In de voorbeelden duiden we de prompt echter aan met `?`.

Doordat in de prelude onder andere rekenkundige functies gedefinieerd worden, kan de interpreter direct gebruikt worden als rekenmachine.

De interpreter berekent de waarde van de ingetikte expressie, waarbij $*$ vermenigvuldiging aanduidt.

De van de rekenmachine bekende functies kunnen ook gebruikt worden in een expressie:

```
? sqrt(2.0)
1.41421
```

De functie `sqrt` berekent de "square root", oftewel de vierkantswortel van een getal.

```
<statement>                 evaluate/run <statement>
:                           repeat last command
:{\n ..lines.. \n:}\n       multiline command
:add [*]<module> ...        add module(s) to the current target set
:browse[!] [[*]<mod>]       display the names defined by module <mod>
                            (!: more details; *: all top-level names)
:cd <dir>                   change directory to <dir>
:cmd <expr>                 run the commands returned by <expr>::IO String
:ctags[!] [<file>]          create tags file for Vi (default: "tags")
                            (!: use regex instead of line number)
:def <cmd> <expr>           define command :<cmd> (later defined command has
                            precedence, ::<cmd> is always a builtin command)
:edit <file>                edit file
:edit                       edit last module
:etags [<file>]             create tags file for Emacs (default: "TAGS")
:help, :?                   display this list of commands
:info [<name> ...]          display information about the given names
:issafe [<mod>]             display safe haskell information of module <mod>
:kind <type>                show the kind of <type>
:load [*]<module> ...       load module(s) and their dependents
:main [<arguments> ...]     run the main function with the given arguments
:module [+/-] [*]<mod> ...  set the context for expression evaluation
:quit                       exit GHCi
:reload                     reload the current module set
:run function [<arguments> ...] run the function with the given arguments
:script <filename>          run the script <filename>
:type <expr>                show the type of <expr>
:undef <cmd>                undefine user-defined command :<cmd>
:!<command>                 run the shell command <command>
```

Fig. 2.2: Basiscommando's beschikbaar vanaf de `ghci` prompt

```
:abandon                    at a breakpoint, abandon current computation
:back                       go back in the history (after :trace)
:break [<mod>] <l> [<col>]  set a breakpoint at the specified location
:break <name>               set a breakpoint on the specified function
:continue                   resume after a breakpoint
:delete <number>            delete the specified breakpoint
:delete *                   delete all breakpoints
:force <expr>               print <expr>, forcing unevaluated parts
:forward                    go forward in the history (after :back)
:history [<n>]              after :trace, show the execution history
:list                       show the source code around current breakpoint
:list identifier            show the source code for <identifier>
:list [<module>] <line>     show the source code around line number <line>
:print [<name> ...]         prints a value without forcing its computation
:sprint [<name> ...]        simplifed version of :print
:step                       single-step after stopping at a breakpoint
:step <expr>                single-step into <expr>
:steplocal                  single-step within the current top-level binding
:stepmodule                 single-step restricted to the current module
:trace                      trace after stopping at a breakpoint
:trace <expr>               evaluate <expr> with tracing on (see :history)
```

Fig. 2.3: Commando's voor debuggen

```
 :set <option> ...           set options
 :seti <option> ...          set options for interactive evaluation only
 :set args <arg> ...         set the arguments returned by System.getArgs
 :set prog <progname>        set the value returned by System.getProgName
 :set prompt <prompt>        set the prompt used in GHCi
 :set editor <cmd>           set the command used for :edit
 :set stop [<n>] <cmd>       set the command to run when a breakpoint is hit
 :unset <option> ...         unset options

Options for ':set' and ':unset':

  +m            allow multiline commands
  +r            revert top-level expressions after each evaluation
  +s            print timing/memory stats after each evaluation
  +t            print type after evaluation
  -<flags>      most GHC command line flags can also be set here
                       (eg. -v2, -fglasgow-exts, etc.)
                for GHCi-specific flags, see User's Guide,
                Flag reference, Interactive-mode options
```

Fig. 2.4: Commando's voor het veranderen van de instellingen

```
 :show bindings              show the current bindings made at the prompt
 :show breaks                show the active breakpoints
 :show context               show the breakpoint context
 :show imports               show the current imports
 :show modules               show the currently loaded modules
 :show packages              show the currently active package flags
 :show language              show the currently active language flags
 :show <setting>             show value of <setting>, which is one of
                                [args, prog, prompt, editor, stop]
 :showi language             show language flags for interactive evaluation
```

Fig. 2.5: Commando's voor het opvragen van informatie

Omdat functies in een functionele taal zo veel gebruikt worden, mogen de haakjes worden weggelaten bij de aanroep (gebruik in een expressie) van een functie. In ingewikkelde expressies scheelt dat een heleboel haakjes, en dat maakt zo'n expressie een stuk overzichtelijker. Bovenstaande aanroep van sqrt kan dus ook zo geschreven worden:

```
? sqrt 2.0
1.41421
```

In wiskundeboeken is het gebruikelijk dat "naast elkaar zetten" van expressies betekent dat die expressies vermenigvuldigd moeten worden. Bij aanroep van een functie moeten er dan haakjes worden gezet. In Haskell-expressies komt functieaanroep echter veel vaker voor dan vermenigvuldigen. Daarom wordt "naast elkaar zetten" in Haskell geïnterpreteerd als functieaanroep, en moet vermenigvuldiging expliciet worden genoteerd (met een $*$):

```
? sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1.0
```

Grote hoeveelheden getallen kunnen in Haskell in een *lijst* worden geplaatst. Lijsten worden genoteerd met behulp van vierkante haken. De prelude bevat veel functies die op lijsten werken, zoals:

```
? sum [1..10]
55
```

In bovenstaand voorbeeld is $[1\,..\,10]$ de Haskell-notatie voor de lijst getallen van 1 tot en met 10. De standaardfunctie sum kan op zo'n lijst worden toegepast om de som (55) van de getallen in de lijst te bepalen. Net als bij sqrt en sin zijn (ronde) haakjes overbodig bij de aanroep van de functie sum.

Behalve dat we lijsten als argument aan functies kunnen geven, kunnen functies ook weer lijsten opleveren:

```
? reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

De standaardfunctie reverse zet dus de elementen van een lijst in omgekeerde volgorde.

De namen van de standaardfuncties die lijsten manipuleren spreken vaak voor zich: length bepaalt bijvoorbeeld de lengte van een lijst, en replicate maakt een lijst met een aantal kopieën van een waarde:

```
? length [1,5,9,3]
4
? replicate 10 3
```

```
[3,3,3,3,3,3,3,3,3,3]
```

Merk op dat als een functie meerdere parameters heeft er geen komma's nodig zijn tussen die parameters. In één expressie kunnen meerdere functies gecombineerd worden. Zo is het bijvoorbeeld mogelijk om eerst een lijst te maken met behulp van replicate, en die vervolgens om te draaien

```
? reverse (replicate 5 2)
[2,2,2,2,2]
```

(niet dat het omdraaien veel uithaalt, maar het gebeurt wel!). Zoals in wiskundeboeken ook gebruikelijk is, betekent $f\ (g\ x)$ dat de functie g op x moet worden toegepast, en dat f op het resultaat daarvan moet worden toegepast. De haakjes zijn in dit voorbeeld (zelfs in Haskell!) noodzakelijk, om aan te geven dat het resultaat van $(g\ x)$ dient als argument voor de functie f.

2.2.2 Functies definiëren

Zoals al gemeld bestaan Haskell programma's grotendeels uit functiedefinities die, zoals in vrijwel alle programmeertalen, worden opgeslagen als textfiles met Unicode-karakters. Zulke files kunnen worden gemaakt met een tekstverwerker naar keuze. Voor de meeste editors met syntax-highlighting bestaan plugins voor Haskell.

Gebruikelijk is dat de filenaam van het programma met een hoofdletter begint en de extensie `.hs` heeft (Haskell-script).

Alhoewel voor eenvoudig experimenteren niet strikt noodzakelijk is het verstandig in de eerste regel van de file de naam van de module te vermelden (dit is tevens de filenaam zonder extensie) bijvoorbeeld:

module MijnEersteHaskellModule **where**

Het keyword **where** duidt aan dat er nu functiedefinities volgen.

Het is omslachtig om voor elke verandering in een functiedefinitie de Haskell-interpreter te verlaten, de tekstverwerker te starten om de functiedefinities te veranderen, de tekstverwerker weer te verlaten, de Haskell-interpreter weer te starten, enzovoort. Daarom is het mogelijk gemaakt om de tekstverwerker te starten *zonder* de Haskell-interpreter te verlaten; bij het verlaten van de tekstverwerker staat de interpreter dan meteen weer klaar om de nieuwe definitie te verwerken.

De tekstverwerker wordt gestart door "`:edit`" in te tikken, gevolgd door de naam van een file, bijvoorbeeld:

```
? :edit MijnEersteHaskellModule.hs
```

Door de dubbele punt aan het begin van de regel weet de interpreter dat `edit` geen functie is die uitgerekend moet worden, maar een huishoudelijke mededeling. Er wordt nu een tekstverwerker opgestart.

In de file `MijnEersteHaskellModule.hs` kan nu bijvoorbeeld de definitie worden gezet van de faculteitsfunctie. De faculteit van een getal n (vaak genoteerd als $n!$) is het product van de getallen van 1 tot en met n, bijvoorbeeld $4! = 1 * 2 * 3 * 4 = 24$. In Haskell ziet de definitie van de functie fac er bijvoorbeelt als volgt uit:

```
fac n = product [1..n]
```

Deze definitie maakt gebruik van de notatie voor "lijst van getallen tussen twee waarden" en de standaardfunctie product, die alle getallen uit de lijst met elkaar vermenigvuldigt.

Voordat de nieuwe functie kan worden gebruikt, moet Haskell weten dat de nieuwe file functiedefinities bevat. Dat kan hem meegedeeld worden met het commando `:l` (afkorting van "load") dus:

```
? :l MijnEersteHaskellModule.hs
```

Daarna kan de nieuwe functie gebruikt worden:

```
? fac 6
720
```

Het is mogelijk om later definities aan een file toe te voegen.

Een functie die bijvoorbeeld aan de file kan worden toegevoegd is die voor "n boven k": het aantal manieren waarop k objecten uit een verzameling van n gekozen kunnen worden. Volgens de kansrekeningboeken is dat aantal

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$$

Deze definitie kan, net als die van fac, vrijwel letterlijk in Haskell worden opgeschreven:

```
boven n k = fac n `div` (fac k * fac (n - k))
```

In functiedefinities kunnen zowel functies uit de prelude aangeroepen worden, als andere functies die in de file worden gedefiniëerd: boven maakt bijvoorbeeld gebruik van de functie fac.

Na het veranderen van de file in de tekstverwerker wordt de veranderde file automatisch door Haskell bekeken; het is dus niet nodig om opnieuw een `:load`-opdracht te geven. Er kan meteen bepaald worden op hoeveel manieren uit tien mensen een commissie van drie samengesteld kan worden:

```
? boven 10 3
120
```

2.2.3 Opdrachten aan de interpreter

Naast `:?` en `:l` herhalen we vanwege hun grote belang nog enkele andere opdrachten die direct voor de interpreter zijn bedoeld.

:q (quit) Met deze opdracht wordt een Haskell-sessie afgesloten.

:t expressie (type) Door deze opdracht wordt het type (zie sectie 2.5) van de gegeven expressie bepaald. p. 46

2.3 Standaardfuncties

2.3.1 Ingebouwd/voorgedefinieerd

Behalve functiedefinities kunnen in Haskell-programma's ook constanten en operatoren worden gedefinieerd (*constante* is eigenlijk een functie zonder parameters):

$$\mathsf{pi} = 3.1415927$$

Een *operator* is een functie met twee parameters die *tussen* de parameters wordt geschreven in plaats van er voor. In Haskell is het mogelijk om zelf operatoren te definiëren. De functie boven uit paragraaf 2.2.2 had misschien beter als operator gedefinieerd kunnen worden, en bijvoorbeeld als `!^!` genoteerd kan worden: p. 29

$$\mathsf{n}\ !\hat{}!\ \mathsf{k} = \mathsf{fac\ n} \ / \ (\mathsf{fac\ k} * \mathsf{fac}\ (\mathsf{n} - \mathsf{k}))$$

In de prelude worden ruim tweehonderd standaardfuncties en -operatoren gedefinieerd. Het grootste deel van de prelude bestaat uit gewone functiedefinities, zoals je die ook zelf kunt schrijven. De functie sum bijvoorbeeld zit alleen maar in de prelude omdat hij zo vaak gebruikt wordt; als hij er niet in had gezeten, dan had je er zelf een definitie voor kunnen schrijven.

Je kunt de definitie gewoon bekijken in de file `Prelude.hs`. Dit is direct een handige manier om te weten te komen wat een standaardfunctie doet. Voor sum luidt de definitie bijvoorbeeld

$$\mathsf{sum} = \mathsf{foldl'}\ (+)\ 0$$

Dan moet je natuurlijk wel weten wat de standaardfunctie foldl′ doet, maar ook dat is op te zoeken...

Er bestaan maar een paar functies die je niet zelf had kunnen definiëren, zoals de optellingsoperator. Deze functies worden door de prelude op hun beurt geïmporteerd uit de module `PreludePrim`. Van die module is de source niet te bekijken; deze functies zijn op magische wijze ingebouwd in de interpreter. Deze functies worden *ingebouwde functies* genoemd; hun definitie zit ingebouwd in de interpreter (in het Engels heten ze *primitive functions*). Het aantal ingebouwde functies in de prelude is zo klein mogelijk gehouden. De meeste standaardfuncties zijn gewoon in Haskell gedefinieerd. Deze functies worden *voorgedefinieerde* functies genoemd.

2.3.2 Namen van functies en operatoren

In de functiedefinitie

```
fac n = product [1..n]
```

is fac de naam van een functie die gedefinieerd wordt, en n de naam van zijn parameter.

Namen van functies en parameters moeten met een kleine letter beginnen. Daarna mogen nog meer letters volgen (zowel kleine letters als hoofdletters), maar ook cijfers, het apostrofteken (') en het onderstrepingsteken (_). Kleine letters en hoofdletters worden als verschillende letters beschouwd. Een paar voorbeelden van mogelijke functie- of parameternamen zijn:

```
f
sum
x3
g'
tot_de_macht
extreemLangeNaamWaarvoorGeenAfkortingBedachtKonWorden
```

Het onderstrepingsteken wordt vaak gebruikt om een lange naam gemakkelijk leesbaar te maken. Een andere manier daarvoor is om de woorden die samen één naam vormen (behalve het eerste woord) met een hoofdletter te laten beginnen. Deze conventie, genaamd camel case, is ook in andere programmeertalen gebruikelijk.

Cijfers en apostrofs in een naam kunnen gebruikt worden om te benadrukken dat een aantal functies of parameters met elkaar te maken hebben. Dit is echter alleen bedoeld voor de menselijke lezer; voor de interpreter heeft de naam x3 even weinig met x2 te maken als qX'a_y.

Namen die met een hoofdletter beginnen worden voor speciale functies en constanten gebruikt, de zogenaamde *constructorfuncties*. De definitie daarvan wordt beschreven in hoofdstuk 6. p. 128

Er zijn 22 namen die niet voor functies of variabelen gebruikt mogen worden. Deze *gereserveerde woorden* hebben een speciale betekenis voor de interpreter. Dit zijn de gereserveerde woorden in Haskell:

case of
if then else
let in
where
do
data type newtype deriving
class instance
infix infixl infixr
module import
default
_

De laatste van deze is de zogenaamde underscore. De betekenis van de gereserveerde woorden komt later in dit diktaat aan de orde.

Operatoren bestaan uit één of meer symbolen. Een operator kan uit één symbool bestaan (bijvoorbeeld +), maar ook uit twee (&&) of meer (`!^!`) symbolen. De symbolen waaruit een operator opgebouwd kan worden zijn de volgende:

```
:  #  $  %  &  *  +  -  =  .  /  \  <  >  ?  !  @  ^  |
```

Toegestane operatoren in programmatekst zijn bijvoorbeeld:

```
+   *.    ++   &&    ||   <=   ==   /=   .  $  //
?   @@    -*-  \/    /\   ...  <+>  :->
```

De operatoren op de eerste van deze twee regels worden in de prelude gedefinieerd. Op de tweede regel staan operatoren die je zelf gedefinieerd zou kunnen hebben. Operatoren die met een dubbele punt (:) beginnen kunnen uitsluitend gebruikt worden voor *constructoroperatoren* (net zoals namen die met een hoofdletter beginnen dat zijn voor constructorfuncties); wat dat precies zijn zullen we verderop in dit dictaat vertellen.

Er zijn elf symbolen of symbolencombinaties die niet als operator gebruikt mogen worden, omdat ze een speciale betekenis hebben in Haskell. Wel mogen ze deel uitmaken van een langere symbolencombinatie om een operator te vormen. Het gaat om de volgende combinaties:

```
::   =   ..   --  @   \   ||   <-   ->   ~   =>
```

2.3.3 Functies op getallen

Er zijn twee soorten getallen beschikbaar in Haskell:

- Gehele getallen, zoals `17`, `0` en `-3`;
- "Floating-point getallen", zoals `2.5`, `-7.81`, `0.0`, `1200.0`, `1.2e3`, `4.5e-6` en `0.005`.

De letter `e` in floating-point getallen betekent "maal tien-tot-de". Bijvoorbeeld 1.2e3 is het getal $1.2 \cdot 10^3 = 1200.0$. Het getal `0.5e-2` staat voor $0.5 \cdot 10^{-2} = 0.005$.

De vier rekenkundige operatoren optellen ($+$), aftrekken ($-$), vermenigvuldigen ($*$) en delen ($/$) werken voor alle soorten getallen:

```
? 5-12
-7
? 2.5*3.0
7.5
? 19/4
4.75
```

Hier speelt een subtiliteit. Omdat veel zaken die bij andere programmeertalen zijn ingebouwd in Haskell via definities in de prelude en bibliotheken worden geregeld kunnen we vaak pas goed begrijpen hoe iets precies zit als we de hele taal behandeld hebben. We zullen dus af en toe naar iets moeten verwijzen wat pas in latere hoofdstukken in detail aan de orde komt. Zo bevat Haskell een uitgebreide verzameling definities die het mogelijk maken programma's op een intuïtieve manier te op te schrijven, terwijl er toch iets anders staat dan je op het eerste gezicht zou denken. Omdat zowel $2 + 3$ als $2.0 + 3.5$ beide correcte Haskell expressies zijn zou je in eerste instantie kunnen denken dat $+$ zowel voor *Integer*'s als *Float*'s werkt. Later zullen we zien dat de $+$ in feite voor een hele verzameling verschillende $+$-en staat, waaruit op magische wijze telkens de van toepassing zijnde wordt geselecteerd. Vooreerst volstaat de interpretatie: "$+$ werkt tussen alles wat je kan optellen". Een tipje van de sluier wordt opgelicht als je in de interpreter `ghci` vraagt naar het type van de operator $+$:

```
$ ghci
...
Prelude> Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
Prelude>
```

We komen hier later uitgebreid op terug, maar omdat je deze types misschien in foutmeldingen al onverwacht tegen komt, ben je er vast voor gewaarschuwd. Je kunt dit type voorlopig lezen als: "Voor alle types a die de eigenschap *Num* hebben, bestaat er een functie $+$."

In de prelude wordt daarnaast een Haskell-definitie gegeven voor een aantal standaardfuncties op getallen. Deze functies zijn dus niet "ingebouwd", maar slechts "voorgedefinieerd" en hadden, als ze niet in de prelude zaten, eventueel ook zelf gedefinieerd kunnen worden. Enkele van deze voorgedefinieerde functies zijn:

abs	de absolute waarde van een getal
signum	-1 voor negatieve getallen, 0 voor nul, 1 voor positieve getallen
gcd	de grootste gemene deler van twee getallen
^	de "machtsverheffen"-operator

Een aantal functies die echt ingebouwd zijn, zijn:

sqrt	de vierkanstswortelfunctie
sin	de sinusfunctie
log	de natuurlijke logaritme
exp	de exponentiële functie (e-tot-de-macht)

Er zijn twee ingebouwde functies om tussen gehele getallen en floating-point getallen te converteren:

fromInteger	maakt een geheel getal tot een floating-point getal
truncate	gooit het deel achter de punt weg

Tegenwoordig hoeft het geen probleem te zijn om efficiënt met getallen tot $2^{63} - 1$ (en groter of gelijk aan -2^{63} te rekenen als je je tot *Int*s beperkt. Anders kan er *overflow* optreden:

```
Prelude> 3 * 100000000
300000000
Prelude> 3*10000000000000000000000000 :: Int
-7613292870346211328
Prelude> 3*10000000000000000000000000
30000000000000000000000000
```

In het middelste geval hebben we expliciet aangegeven dat het type van het resultaat *Int* moet zijn; de precisie waarmee de machine het antwoord berekend is kennelijk niet voldoende, en we krijgen een onzinantwoord. In het laatste geval hebben we dat niet aangegeven, en wordt met voldoende grote precisie gerekend. Wat hier gebeurt is dat impliciet de interpreter kiest voor het *Integer* type waarmee willekeurig grote integers kunnen worden gerepresenteerd (zolang het computergeheugen dat toelaat). Je zult in programma's soms ook zelf expliciet aan willen geven dat je met *Integer* wil werken. Dat gaat dan als volgt:

```
Prelude> 3*10000000000000000000000000^14000 :: Integer
```

maar let wel dat het even duurt voor het volledige antwoord op je scherm is weergegeven (^ staat voor machtsverheffen.)

Ook floating-point getallen kennen een maximum waarde (ongeveer 10^{308}), en een kleinste positieve waarde (10^{-308}). Bovendien is de rekennauwkeurigheid beperkt tot 15 significante cijfers. Het is aan te raden om voor discrete grootheden, zoals aantallen,

altijd gehele getallen te gebruiken. Floating-point getallen kunnen worden gebruikt voor continue grootheden zoals afstanden en gewichten.

2.3.4 Boolse functies

De operator < vergelijkt twee getallen. De uitkomst is de constante *True* als het linker argument kleiner is dan het rechter of de constante *False* als dat niet het geval is:

```
? 1<2
True
? 2<1
False
```

De waarden *True* en *False* zijn de enige elementen van de verzameling *waarheidswaarden (ook wel booleans of boolse waarden genoemd)* (genoemd naar de Engelse wiskundige George Boole). Functies (en operatoren) die zo'n waarde opleveren heten *Boolean functions* of *boolse functies*.

Behalve < bestaat er ook een operator > (groter-dan), een operator <= (kleiner-of-gelijk), en een operator >= (groter-of-gelijk). Daarnaast bestaan er operatoren == (gelijk-aan) en operator /= (ongelijk-aan). Voorbeelden:

```
? 2+3 > 1+2
True
? 5 /= 1+4
False
? sqrt 2.0 == 1.5
False
```

Uitkomsten van boolse functies kunnen gecombineerd worden met de operatoren && ("en", `&&`) en || ("of", `||`). De operator && geeft alleen *True* als resultaat als links èn rechts een ware uitspraak staat:

```
? 1<2 && 3<4
True
? 1<2 && 3>4
False
```

Voor de "of"-operator hoeft maar één van de twee uitspraken waar te zijn (maar allebei mag ook):

```
? 1==1 || 2==3
True
```

Er is een functie not die *True* en *False* op elkaar afbeeldt. Verder vinden we in de prelude een functie even die kijkt of een geheel getal een even getal is:

```
? not False
True
? not (1<2)
False
? even 7
False
? even 0
True
```

2.3.5 Functies op lijsten

In de prelude wordt een aantal functies en operatoren op lijsten gedefinieerd. Hiervan is er slechts één ingebouwd (de constructor :), de rest is gedefinieerd met behulp van een Haskell-definitie.

Sommige functies op lijsten zijn al eerder besproken: length bepaalt de lengte van een lijst, en reverse levert de elementen van een lijst op in omgekeerde volgorde.

De operator : zet een extra element op kop van een lijst. Bijvoorbeeld:

```
? 1 : []
[1]
? 1 : (2 : [])
[1,2]
```

Het is misschien wat raar dat de waarde van $1 : (2 : [])$ wordt weergegeven als $[1, 2]$. Dit komt doordat in Haskell de laatste als *syntactische suiker* voor de eerste is geïntroduceerd. Vanwege de leesbaarheid prefereren we $[1, 2]$ boven $1 : (2 : [])$, en vanaf nu zullen we die vorm dan ook zoveel mogelijk gebruiken.

De operator ++ plakt twee lijsten aan elkaar.

```
? [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

De functie null is een boolse functie op lijsten. Deze functie kijkt of een lijst leeg is (geen elementen bevat). De functie and werkt op een lijst waarvan de elementen boolse waardes zijn; and controleert of alle elementen van de lijst *True* zijn:

```
? null [ ]
True
? and  [ 1<2, 2<3, 1==0 ]
False
```

Sommige functies hebben twee parameters. De functie take krijgt bijvoorbeeld een getal en een lijst als parameter. Als het getal n is, levert deze functie de eerste n elementen van de lijst op:

```
? take 3 [2..10]
[2, 3, 4]
```

Merk hier weer op dat de twee argumenten niet tussen haakjes staan, of door een komma worden gescheiden, zoals je wellicht uit andere programmeertalen gewend bent.

2.3.6 Functies op functies

In de functies die tot nu toe besproken zijn, zijn de parameters getallen, boolse waarden of lijsten. Een argument van een functie kan echter zelf ook een functie zijn! Een voorbeeld daarvan is de functie map, die twee parameters heeft: een functie en een lijst. De functie map past zijn eerste argument (wat een functie moet zijn) toe op alle elementen van zijn tweede argument, wat een lijst moet zijn. Bijvoorbeeld:

```
? map fac [1,2,3,4,5]
[1, 2, 6, 24, 120]
? map sqrt [1.0,2.0,3.0,4.0]
[1.0, 1.41421, 1.73205, 2.0]
? map even [1..8]
[False, True, False, True, False, True, False, True]
```

Functies met functies als parameter worden veel gebruikt in Haskell (het heet niet voor niets een "functionele" taal!). In hoofdstuk 3 worden meer van dit soort functies bespro- p. 56
ken.

2.4 Functiedefinities

2.4.1 Definitie door combinatie

De eenvoudigste manier om functies te definiëren is door een aantal andere functies, bijvoorbeeld standaardfuncties uit de prelude, te combineren:

```
fac n                    = product [1 .. n]
oneven           x       = not (even x)
kwadraat         x       = x * x
som_van_kwadraten lijst  = sum (map kwadraat lijst)
```

Functies kunnen ook meer dan één parameter krijgen:

```
boven       n k  = fac n / (fac k * fac (n - k))
abcFormule a b c = [(-b + sqrt (b * b - 4.0 * a * c)) / (2.0 * a)
                   ,(-b - sqrt (b * b - 4.0 * a * c)) / (2.0 * a)
                   ]
```

Functies met nul parameters worden meestal "constanten" genoemd:

```
pi = 3.1415926535
e  = exp 1.0
```

Elke functiedefinitie heeft dus de volgende vorm:

- de naam van de functie
- de namen van eventuele parameters
- een =-teken
- een expressie waar de parameters, standaardfuncties en zelfgedefinieerde functies in mogen voorkomen.

Bij een functie met als resultaat een boolse waarde staat rechts van het =-teken een expressie met een boolse waarde:

```
negatief x = x < 0
positief x = x > 0
isnul x    = x == 0
```

Let in de laatste definitie op het verschil tussen de = en de ==. Een enkel =-teken scheidt in functiedefinities de linkerkant van de rechterkant. Een dubbel =-teken is een operator, net zoals < en >.

In de definitie van de functie abcFormule komen de expressies sqrt (b * b − 4.0 * a * c) en (2.0 * a) twee keer voor. Behalve dat dat veel tikwerk geeft, kost het uitrekenen van zo'n expressie onnodig veel tijd: de identieke deelexpressies worden tweemaal uitgerekend. Om dat te voorkomen, is het mogelijk om deelexpressies een naam te geven. De verbeterde definitie wordt dan als volgt:

```
abcFormule' a b c = [(-b + d) / n
                    ,(-b - d) / n
                    ]
                    where d = sqrt (b * b - 4.0 * a * c)
                          n = 2.0 * a
```

Het woord **where** is niet de naam van een functie: het is één van de "gereserveerde woorden" die in paragraaf 2.3.2 opgesomd zijn. Achter "**where**" staan definities. In p. 32 dit geval definities van de constanten d en n. Deze constanten mogen in de expressie waarachter **where** staat worden gebruikt. Ze kunnen daarbuiten niet gebruikt worden: het zijn *lokale definities*. Het lijkt misschien vreemd om d en n "constanten" te noemen, omdat de waarde bij elke aanroep van abcFormule′ verschillend kan zijn. We noemen ze constanten omdat ze in tegenstelling tot functies geen parameters hebben en door de onveranderbaarheid van variabelen in Haskell tijdens een gegeven aanroep altijd dezelfde waarde zullen bevatten.

2.4.2 Definitie door gevalsonderscheid

Soms is het nodig om in de definitie van een functie meerdere gevallen te onderscheiden. De absolute-waardefunctie abs is hiervan een voorbeeld: voor een negatieve parameter is de definitie anders dan voor een positieve parameter. In Haskell wordt dat als volgt genoteerd:

```
abs x | x < 0    = -x
      | x >= 0 =  x
```

Er kunnen ook meer dan twee gevallen onderscheiden worden. Dat gebeurt bijvoorbeeld in de definitie van de functie signum:

```
signum x | x > 0    =  1
         | x == 0 =  0
         | x < 0    = -1
```

De definities voor de verschillende gevallen worden "bewaakt" door boolse expressies, die dan ook *guards* worden genoemd.

Als een functie die op deze manier is gedefinieerd wordt aangeroepen, worden de guards één voor één geprobeerd. Bij de eerste guard die de waarde *True* heeft, wordt de expressie rechts van het =-teken uitgerekend. De laatste guard kan dus desgewenst vervangen worden door *True* (of de constante otherwise).

De beschrijving van de vorm van een functiedefinitie is dus uitgebreider dan in de vorige paragraaf gesuggereerd werd. Een completere beschrijving van "functiedefinitie" is:

- de naam van de functie;
- de naam van nul of meer parameters;
- een =-teken en een expressie, òf: één of meer "guarded expressies";
- desgewenst het woord **where** gevolgd door lokale definities.

Daarbij bestaat elke "guarded expressie" uit een |-teken, een boolse expressie, een =-teken, en een expressie[1]. Deze beschrijving is echter ook nog niet volledig...

[1] Deze beschrijving lijkt zelf ook wel een definitie, met een lokale definitie voor "guarded expressie"!

2.4.3 Definitie door patroonherkenning

De parameters van een functie in een functiedefinitie, zoals x en y in

$$\mathsf{f\ x\ y} = \mathsf{x} * \mathsf{y}$$

worden de *formele parameters* van die functie genoemd. Bij aanroep wordt de functie voorzien van *actuele parameters*. Vaak zullen we de formele parameters parameters, en de actuele parameters *argumenten* noemen. Wel zo makkelijk. Bijvoorbeeld, in de aanroep

$$\mathsf{f}\ 17\ (1 + \mathsf{g}\ 6)$$

is 17 het argument dat overeenkomt met de parameter x, en $(1 + \mathsf{g}\ 6)$ het argument dat overeenkomt met de parameter y. Bij aanroep van een functie worden de voorkomens van de parameters in de rechterkant van de definitie vervangen door de argumenten. De expressie hierboven is dus equivalent met $17 * (1 + \mathsf{g}\ 6)$.

Argumenten zijn dus *expressies*. Parameters zijn tot nu toe steeds *namen* geweest. In de meeste programmeertalen moet een formele parameter altijd een naam zijn. In Haskell zijn er echter andere mogelijkheden: een parameter mag ook een *patroon* zijn.

Een voorbeeld van een functiedefinitie, waarin een patroon wordt gebruikt als formele parameter is:

$$\mathsf{f}\ [1, \mathsf{x}, \mathsf{y}] = \mathsf{x} + \mathsf{y}$$

Deze functie werkt alleen op lijsten met precies drie elementen, waarvan het eerste element 1 moet zijn. Van zo'n lijst worden dan het tweede en derde element opgeteld. De functie is dus niet gedefinieerd op kortere of langere lijsten, of op lijsten waarvan het eerste element niet 1 is. (Het is niet ongebruikelijk dat functies niet voor alle mogelijke actuele parameters gedefinieerd zijn. Zo is bijvoorbeeld de functie sqrt niet gedefinieerd voor negatieve argumenten, en de operator / niet voor 0 als rechterargument.)

Je kunt een functie definiëren met verschillende patronen als formele parameter:

$$\begin{aligned}
&\mathsf{som}\ [\,] &&= 0\\
&\mathsf{som}\ [\mathsf{x}] &&= \mathsf{x}\\
&\mathsf{som}\ [\mathsf{x}, \mathsf{y}] &&= \mathsf{x} + \mathsf{y}\\
&\mathsf{som}\ [\mathsf{x}, \mathsf{y}, \mathsf{z}] &&= \mathsf{x} + \mathsf{y} + \mathsf{z}
\end{aligned}$$

Deze functie kan worden toegepast op lijsten met nul, een, twee of drie elementen (in de volgende paragraaf wordt de functie gedefinieerd op willekeurig lange lijsten). In alle gevallen worden de elementen opgeteld. Bij aanroep van de functie kijkt de interpreter of de parameter "past" op een van de definities; de aanroep $\mathsf{som}\ [3, 4]$ past bijvoorbeeld op de derde regel van de definitie. De 3 komt daarbij overeen met de x in de definitie en de 4 met de y. Merk op dat een variabele niet twee keer in een patroon mag voorkomen (in een voorganger van Haskell, Miranda, was dit wel toegestaan en betekende dit dat de waarden die hieraan gebonden werden gelijk moesten zijn).

De volgende constructies zijn toegestaan als patroon:

- getallen (bijvoorbeeld 3),
- de constanten *True* en *False*,
- namen (bijvoorbeeld x),
- lijsten, waarvan de elementen ook weer patronen zijn (bijvoorbeeld $[1, \mathsf{x}, \mathsf{y}]$), en
- de operator : met patronen links en rechts (bijvoorbeeld $a : b$).

Met behulp van patronen zijn een aantal belangrijke functies te definiëren. De operator && uit de prelude kan bijvoorbeeld op deze manier gedefinieerd worden:

$$\begin{aligned} \mathit{False} \;\&\&\; \mathit{False} &= \mathit{False} \\ \mathit{False} \;\&\&\; \mathit{True} &= \mathit{False} \\ \mathit{True} \;\&\&\; \mathit{False} &= \mathit{False} \\ \mathit{True} \;\&\&\; \mathit{True} &= \mathit{True} \end{aligned}$$

Met de operator : kunnen lijsten worden opgebouwd. De expressie $\mathsf{x} : \mathsf{y}$ betekent immers "zet element x op kop van de lijst y". Door de operator : in een patroon te zetten, wordt het eerste element van een lijst juist afgesplitst. Daarmee kunnen twee nuttige standaardfuncties geschreven worden:

$$\begin{aligned} \mathsf{head}\ (\mathsf{x} : \mathsf{y}) &= \mathsf{x} \\ \mathsf{tail}\ \ (\mathsf{x} : \mathsf{y}) &= \mathsf{y} \end{aligned}$$

De functie head levert het eerste element van een lijst op (de "kop"); de functie tail levert alles behalve het eerste element op (de "staart"). Gebruik van deze functies in een expressie kan bijvoorbeeld als volgt:

```
? head [3,4,5]
3
? tail [3,4,5]
[4, 5]
```

De functies head en tail kunnen op bijna alle lijsten worden toegepast; ze zijn alleen niet gedefinieerd op de lege lijst (een lijst zonder elementen): die heeft immers geen eerste element, laat staan een "staart".

2.4.4 Definitie door recursie of inductie

In de definitie van een functie mogen standaardfuncties en zelfgedefinieerde functies gebruikt worden. Maar ook de functie die gedefinieerd wordt mag in zijn eigen definitie gebruikt worden! Zo'n definitie heet een *recursieve definitie* (recursie betekent letterlijk "terugkeer": de naam van de functie keert terug in zijn eigen definitie). De volgende functie is een recursieve functie:

$$\mathsf{f}\ \mathsf{x} = \mathsf{f}\ \mathsf{x}$$

De naam van de functie die gedefinieerd wordt (f) staat in de definiërende expressie rechts van het =-teken. Deze definitie is echter weinig zinvol; om bijvoorbeeld de waarde van f 3 te bepalen, moet volgens de definitie eerst de waarde van f 3 bepaald worden, en daarvoor moet eerst de waarde van f 3 bepaald worden, enzovoort, enzovoort...

Recursieve functies zijn echter wèl zinvol onder de volgende twee voorwaarden:

- het argument van de recursieve aanroep is *eenvoudiger* (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan het argument van de te definiëren functie;
- voor een *basisgeval* is er een niet-recursieve definitie.

Een recursieve definitie van de faculteitsfunctie is de volgende:

$$\begin{aligned} &\mathsf{fac\ n} \mid \mathsf{n} == 0 = 1 \\ &\qquad \mid \mathsf{n} > 0 \quad = \mathsf{n} * \mathsf{fac}\ (\mathsf{n} - 1) \end{aligned}$$

Het basisgeval is hier $\mathsf{n} == 0$; in dit geval kan het resultaat direct (zonder recursie) bepaald worden. In het geval $\mathsf{n} > 0$ is er een recursieve aanroep, namelijk $\mathsf{fac}\ (\mathsf{n} - 1)$. De parameter bij deze aanroep $(\mathsf{n} - 1)$ is, zoals vereist, kleiner dan n.

Een andere manier om deze twee gevallen (het basisgeval en het recursieve geval) te onderscheiden, is gebruik te maken van patroonherkenning:

$$\begin{aligned} &\mathsf{fac}\ 0 = 1 \\ &\mathsf{fac\ n} = \mathsf{n} * \mathsf{fac}\ (\mathsf{n} - 1) \end{aligned}$$

Ook in dit geval is het argument van de recursieve aanroep $(\mathsf{n} - 1)$ kleiner dan het argument van de te definiëren functie (n).

Het gebruik van patronen sluit nauw aan bij de wiskundige traditie van "definiëren met inductie". De wiskundige definitie van machtsverheffen kan bijvoorbeeld vrijwel letterlijk als Haskell-functie worden gebruikt:

$$\begin{aligned} &\mathsf{x\ \hat{}\ 0} = 1 \\ &\mathsf{x\ \hat{}\ n} = \mathsf{x} * \mathsf{x\ \hat{}\ (n} - 1) \end{aligned}$$

Een recursieve definitie waarin voor het gevalsonderscheid patronen worden gebruikt (in plaats van boolse expressies) wordt daarom ook wel een *inductieve definitie* genoemd.

Functies op lijsten kunnen ook recursief zijn. Daarbij is een lijst "kleiner" dan een andere als hij minder elementen heeft (korter is). De in de vorige paragraaf beloofde functie som, die de getallen in een lijst van willekeurige lengte optelt, kan op verschillende manieren worden gedefinieerd. Een gewone recursieve definitie (waarin het onderscheid tussen het recursieve en het niet-recursieve geval wordt gemaakt met guards) luidt als volgt:

$$\begin{aligned} &\mathsf{som\ lijst} \mid \mathsf{null\ lijst} = 0 \\ &\qquad \mid \mathsf{otherwise} = \mathsf{head\ lijst} + \mathsf{som\ (tail\ lijst)} \end{aligned}$$

Maar hier is ook een inductieve versie mogelijk (waarin het gevalsonderscheid wordt gemaakt met patronen):

```
som []           = 0
som (kop : staart) = kop + som staart
```

In de meeste gevallen is een definitie met patronen duidelijker, omdat de verschillende onderdelen in het patroon direct een naam kunnen krijgen (zoals kop en staart in de functie som). In de gewone recursieve versie van som zijn de standaardfuncties head en tail nodig om de onderdelen uit de lijst te peuteren. In die functies worden bovendien alsnog patronen gebruikt.

De standaardfunctie length, die het aantal elementen in een lijst bepaalt, kan ook inductief worden gedefinieerd:

```
length []           = 0
length (kop : staart) = 1 + length staart
```

Daarbij is de waarde van het element op kop van de lijst niet van belang (alleen het feit dàt het er is).

In patronen is het toegestaan om in dit soort gevallen het underscore-teken "_" te gebruiken in plaats van een naam:

```
length []           = 0
length (_ : staart) = 1 + length staart
```

2.4.5 Layout en commentaar

Op de meeste plaatsen in een programma mag extra witte ruimte staan, om het programma overzichtelijker te maken. In bovenstaande voorbeelden zijn bijvoorbeeld extra spaties toegevoegd, om de =-tekens van één functiedefinitie netjes onder elkaar te zetten. Natuurlijk mogen er geen spaties worden toegevoegd midden in de naam van een functie of in een getal: len gth is iets anders dan length, en 1 7 iets anders dan 17.

Ook regelovergangen mogen worden toegevoegd om het resultaat overzichtelijker te maken. In de definitie van abcFormule is dat bijvoorbeeld gedaan, omdat de regel anders wel erg lang zou worden.

Anders dan in andere programmeertalen is een regelovergang echter niet helemaal zonder betekenis. Bekijk bijvoorbeeld de volgende twee **where**-constructies:

```
where          where
  a = f x y        a = f x
  b = g z          y b = g z
```

De plaats van de regelovergang (tussen x en y, of tussen y en b) maakt nogal wat uit.

In een rij definities gebruikt Haskell de volgende methode om te bepalen wat bij elkaar hoort:

- een definitie die *precies evenver* is ingesprongen als de vorige, wordt als nieuwe definitie beschouwd;
- is de definitie *verder* ingesprongen, dan hoort hij bij de vorige regel;
- is de definitie *minder ver* ingesprongen, dan hoort hij niet meer bij de huidige lijst definities.

Dat laatste is van belang als een **where**-constructie binnen een andere **where**-constructie voorkomt. Bijvoorbeeld in

```
f x y = g (x + w)
        where g u = u + v
                    where v = u * u
              w =   2 + y
```

is w een lokale declaratie van f, en niet van g. De definitie van w is immers minder ver ingesprongen dan die van v; hij hoort dus niet meer bij de **where**-constructie van g. Hij is evenver ingesprongen als de definitie van g, en hoort dus bij de **where**-constructie van f. Zou hij nog minder ver zijn ingesprongen, dan hoorde hij zelfs daar niet meer bij, en krijg je een foutmelding, omdat de tweede parameter y van f dan niet meer in scope is voor de definitie van w.

Het klinkt allemaal misschien een beetje ingewikkeld, maar in de praktijk gaat alles vanzelf goed als je één ding in het oog houdt:

gelijkwaardige definities moeten even ver worden ingesprongen

Dit betekent ook dat alle globale functiedefinities even ver moeten worden ingesprongen (bijvoorbeeld allemaal nul posities).

Commentaar

Op elke plaats waar spaties mogen staan (bijna overal dus) mag commentaar worden toegevoegd. Commentaar wordt door de interpreter genegeerd, en is bedoeld voor eventuele menselijke lezers van het programma. Er zijn in Haskell twee soorten commentaar:

- met de symbolen `--` begint commentaar dat tot het eind van de regel doorloopt;
- met de symbolen `{-` begint commentaar dat doorloopt tot de symbolen `-}`.

Uitzondering op de eerste regel is het geval dat `--` deel uitmaakt van een operator, bijvoorbeeld `<-->`. Een losse `--` kan echter geen operator zijn: deze combinatie werd in paragraaf 2.3.2 gereserveerd. p. 32

Commentaar met `{-` en `-}` kan worden *genest*, dat wil zeggen weer paren van deze symbolen bevatten. Het commentaar is pas afgelopen bij het bijbehorende sluitsymbool. Bijvoorbeeld in

```
{-  {- hallo -}  f x = 3  -}
```

wordt géén functie f gedefinieerd; het geheel is één stuk commentaar.

2.5 Typering

2.5.1 Soorten fouten

Vergissen is menselijk, ook bij het schrijven of intikken van een functie. Gelukkig kan de interpreter waarschuwen voor sommige fouten. Als een functiedefinitie niet aan de vormeisen voldoet, krijg je daarvan een melding zodra deze functie geanalyseerd wordt. De volgende definitie bevat een fout:

let isNul x = x = 0

De tweede = had een == moeten zijn (= betekent "is gedefinieerd als", en == betekent "is gelijk aan"). Bij de analyse van deze functie meldt de interpreter:

```
Prelude> let isNul x = x = 0
 <interactive>:2:17: parse error on input ‘=’
```

De vormfouten in een programma (*syntax errors*) worden door de interpreter ontdekt tijdens de eerste fase van de analyse: het ontleden (*to parse*). Andere syntaxfouten zijn bijvoorbeeld openingshaakjes waar geen bijbehorende sluithaakjes bij zijn, of het gebruik van gereserveerde woorden (zoals **where**) op plaatsen waar dat niet mag.

Er zijn behalve syntaxfouten nog andere fouten waar de interpreter voor kan waarschuwen. Een mogelijke fout is het aanroepen van een functie die nergens is gedefinieerd. Vaak zijn dit soort fouten het gevolg van een tikfout. Bij het analyseren van de definitie

```
Prelude> let fac x  =  produkt [1..x]
```

meldt `ghci`:

```
 <interactive>:4:13: Not in scope: ‘produkt’
Perhaps you meant ‘product’ (imported from Prelude)
```

Deze fouten worden opgespoord tijdens de tweede fase: de afhankelijkheidsanalyse (*dependency analysis*).

Het volgende struikelblok voor een programma is de controle van de types (*type checking*). Functies die bedoeld zijn om op getallen te werken mogen bijvoorbeeld niet op boolse waarden toegepast worden, en ook niet op lijsten. Functies op lijsten mogen weer niet op getallen worden gebruikt, enzovoort.

Staat er bijvoorbeeld in een functiedefinitie de expressie $1 +$ *True* dan meldt de Helium interpreter [2].

[2] `http://www.cs.uu.nl/wiki/Helium/WebHome`

```
    (1,8): Type error in infix application
    *** Expression     : 1 + True
    *** Term           : True
    *** Type           : Bool
    *** Does not match : Int
```

De deelexpressie (*term*) *True* heeft het *type Bool* (kort voor boolse waarde). Zo'n boolean kan niet worden opgeteld bij 1, wat van het type *Int* is (een afkorting van *integer, oftewel geheel getal*). Sterker nog, booleans kunnen helemaal niet opgeteld worden.

De `ghci` interpreter geeft een vrij cryptische melding als we vragen om de waarde van $1 + \mathit{True}$:

```
Prelude> 1 + True
 <interactive>:1:0:    No instance for (Num Bool)
      arising from a use of `+' at <interactive>:1:0-7
    Possible fix: add an instance declaration for (Num Bool)
    In the expression: 1 + True
    In the definition of `it': it = 1 + True
```

We zien hier een stukje van de onderliggende structuur van de definitie van de taal Haskell en de achterliggende machinerie. De operator + kan veel verschillende betekenissen hebben, maar kennelijk niet die van een functie die twee booleans bij elkaar kan optellen. We leiden echter ook uit de foutmelding af dat je dat wel zou kunnen definiëren als je dat graag zou willen, en dat is dan ook zo.

Andere typeringsfouten treden bijvoorbeeld op bij het toepassen van de functie length op iets anders dan een lijst, zoals in length 3. Pas als er geen typeringsfouten meer in een programma zitten, kan de vierde analysefase (genereren van code) worden uitgevoerd. Alleen dan kan de functie worden gebruikt.

Alle foutmeldingen worden al gegeven op het moment dat een functie wordt geanalyseerd. De bedoeling hiervan is dat er tijdens het gebruik van een functie geen onaangename verrassingen meer optreden. Een functie die de analyse doorstaat, bevat gegarandeerd geen typeringsfouten meer. Dat wil echter niet zeggen dat je Haskell programma niet kan crashen. Gegeven een definitie als

$$\mathsf{head}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x}$$

zal je programma crashen als je om head [] vraagt, omdat voor dat geval geen definitie is gegeven. Ook zal, bijvoorbeeld, delen door nul leiden tot een run-time exceptie.

Sommige andere talen controleren de typering pas op het moment dat een functie wordt aangeroepen. In dat soort talen weet je nooit zeker of er ergens in een ongebruikte uithoek van het programma nog een typeringsfout verborgen ligt...

Het feit dat een functie de analyse doorstaat wil natuurlijk niet zeggen dat de functie correct is. Als in de functie som een minteken staat in plaats van een plusteken, dan zal de interpreter daar niet over klagen: hij kan immers niet weten dat het de bedoeling is dat som getallen optelt. Dit soort fouten, "logische fouten" genaamd, zijn het moeilijkst te vinden, omdat de interpreter er niet voor waarschuwt.

2.5.2 Typering van expressies

Het type van een expressie kan bepaald worden met de interpreteropdracht `:t` (afkorting van "type"). Achter `:t` staat de expressie die getypeerd moet worden. Bijvoorbeeld:

```
?  :t True && False
True && False :: Bool
```

Het symbool :: kan gelezen worden als "heeft het type". De expressie wordt met de `:type`-opdracht niet uitgerekend; alleen het type wordt bepaald.

We noemen de volgende belangrijke basistypes:

- *Int*: het type van de gehele getallen (*integer numbers* of *integers*), tot een maximum van zo'n $2^{63} - 1$,
- *Integer*: het type van gehele getallen, praktisch zonder begrenzing
- *Float*: het type van de floating-point getallen;
- *Bool*: het type van de boolse waardes *True* en *False*;
- *Char*: het type van letters, cijfers en symbolen op het toetsenbord (*characters*), dat in paragraaf 4.2.2 zal worden besproken. p. 95

Let er op dat deze types met een hoofdletter geschreven worden.

Lijsten kunnen verschillende types hebben. Zo zijn er bijvoorbeeld lijsten van integers, lijsten van booleans, en zelfs lijsten van lijsten van integers. Al deze lijsten hebben een verschillend type:

```
? :t ['a', 'b', 'c']
['a','b','c'] :: [Char]
? :t [True,False]
[True,False] :: [Bool]
? :t [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

Het type van een lijst wordt aangegeven door het type van de elementen van een lijst tussen vierkante haken te zetten: [*Int*] is het type van een lijst gehele getallen. Alle elementen van een lijst moeten van hetzelfde type zijn. Zo niet, dan verschijnt er een melding van een typeringsfout.

Ook functies hebben een type. Het type van een functie wordt bepaald door het type van de parameter en het type van het resultaat. Het type van de functie sum is bijvoorbeeld als volgt:

```
? :t sum
sum :: [Int] -> Int
```

De functie sum werkt op lijsten integers en heeft als resultaat een enkele integer. Het symbool -> in het type van de functie moet een pijltje (→) voorstellen. In handschrift kan dit gewoon als pijltje geschreven worden.

Andere voorbeelden van types van functies zijn:

$$\begin{aligned}
&\mathsf{sqrt} && :: \mathit{Float} \mathrel{->} \mathit{Float} \\
&\mathsf{even} && :: \mathit{Int} \mathrel{->} \mathit{Bool} \\
&\mathsf{reverse} && :: [\mathit{Int}] \mathrel{->} [\mathit{Int}]
\end{aligned}$$

Zo'n regel kun je uitspreken als "even heeft het type int naar bool" of "even is een functie van int naar bool".

Omdat functies (net als getallen, boolse waarden en lijsten) een type hebben, is het mogelijk om functies in een lijst op te nemen. De functies die in één lijst staan moeten dan wel precies hetzelfde type hebben, omdat de elementen van een lijst hetzelfde type moeten hebben. Een voorbeeld van een lijst functies is:

```
? :t [sin,cos,tan]
[sin,cos,tan] :: [Float -> Float]
```

De drie functies sin, cos en tan zijn allemaal functies "van float naar float"; ze kunnen dus in een lijst gezet worden, die dan het type "lijst van functies van float naar float" heeft.

De interpreter kan zelf het type van een expressie of een functie bepalen. Dit gebeurt dan ook bij het controleren van de typering van een programma. Desondanks is het toegestaan om het type van een functie in een programma erbij te schrijven. Een functiedefinitie ziet er dan bijvoorbeeld als volgt uit:

$$\begin{aligned}
&\mathsf{sum} && :: [\mathit{Int}] \mathrel{->} \mathit{Int} \\
&\mathsf{sum}\ [\,] && = 0 \\
&\mathsf{sum}\ (\mathsf{x} : \mathsf{xs}) && = \mathsf{x} + \mathsf{sum}\ \mathsf{xs}
\end{aligned}$$

Hoewel zo'n *typedeclaratie* overbodig is, heeft hij twee voordelen:

- Er wordt gecontroleerd of de functie inderdaad het type heeft dat je ervoor hebt gedeclareerd.
- De declaratie maakt het voor een menselijke lezer eenvoudiger om een functie te begrijpen.

De typedeclaratie hoeft niet direct voor de definitie te staan. Je zou bijvoorbeeld een programma kunnen beginnen met de declaraties van de types van alle functies die erin worden gedefinieerd. De declaraties dienen dan als een soort inhoudsopgave.

2.5.3 Polymorfie

Voor sommige functies op lijsten maakt het niet uit wat het type van de elementen van die lijst is. De standaardfunctie length bijvoorbeeld, kan de lengte bepalen van een lijst integers, maar ook van een lijst boolse waarden, en –waarom niet– van een lijst functies. Het type van de functie length wordt als volgt genoteerd:

length :: [a] -> Int

Dit type geeft aan dat de functie een lijst als parameter heeft, maar het type van de elementen van de lijst doet er niet toe. Het type van deze elementen wordt aangegeven door een *typevariabele*, in het voorbeeld a. Typevariabelen worden, in tegenstelling tot de vaste types als *Int* en *Bool*, met een kleine letter geschreven.

De functie head, die het eerste element van een lijst oplevert, heeft het volgende type:

head :: [a] -> a

Ook deze functie werkt op lijsten waarbij het type van de elementen niet belangrijk is. Het resultaat van de functie head heeft echter hetzelfde type als de elementen van de lijst (het is immers het eerste element van de lijst). Voor het type van het resultaat wordt dan ook dezelfde typevariabele gebruikt als voor het type van de elementen van de lijst.

Een type waar typevariabelen in voorkomen heet een *(parametrisch) polymorf type* (letterlijk: "veelvormig type"). Functies met een polymorf type heten polymorfe functies. Het verschijnsel zelf heet *(parametrische) polymorfie* of *polymorfisme*. Karakteristiek voor functies die parametrisch polymorf zijn, is dat voor alle types waarvoor de functie "werkt" er één enkele definitie is. Zo werkt de definitie van head voor zowel lijsten van integers als voor lijsten van booleans.

Polymorfe functies, zoals length en head, hebben met elkaar gemeen dat ze alleen de *structuur* van de lijst gebruiken. Een niet-polymorfe functie, zoals sum, gebruikt ook eigenschappen van de *elementen* van de lijst, zoals "optelbaarheid".

Polymorfe functies zijn vaak algemeen bruikbaar; in veel programma's moet bijvoorbeeld wel eens de lengte van een lijst bepaald worden. Daarom zijn veel van de standaardfuncties in de prelude polymorfe functies.

Niet alleen functies op lijsten kunnen polymorf zijn. De eenvoudigste polymorfe functie is de identiteitsfunctie (de functie die zijn parameter onveranderd oplevert):

```
id    :: a -> a
id x = x
```

De functie id kan op elementen van willekeurig type werken (en het resultaat is dan van hetzelfde type). Hij kan dus worden toegepast op een integer, bijvoorbeeld id 3, maar ook op een boolse waarde, bijvoorbeeld id *True*. Ook kan de functie werken op lijsten van booleans, bijvoorbeeld id $[True, False]$ of op lijsten van lijsten van integers: id $[[1,2,3],[4,5]]$. De functie kan zelfs worden toegepast op functies van float naar float, bijvoorbeeld id sqrt, of op functies van lijsten van integers naar integers: id sum. Zoals het type al aangeeft kan de functie worden toegepast op parameters van een willekeurig type. De parameter mag dus ook het type $a \rightarrow a$ hebben, zodat de functie id ook op zichzelf kan worden toegepast: id id.

2.5.4 Functies met meer parameters

Ook functies met meer dan één parameter hebben een type. In het type staat tussen de parameters onderling, en tussen de laatste parameter en het resultaat, een pijltje. De functie boven uit paragraaf 2.2.2 heeft twee integer parameters en een integer resultaat. p. 30 Het type is daarom:

boven :: *Int* -> *Int* -> *Int*

De functie abcFormule uit paragraaf 2.4.1 heeft drie floating-point getallen als parameter en een lijst met floating-point getallen als resultaat. De typedeclaratie luidt daarom: p. 38

abcFormule :: *Float* -> *Float* -> *Float* -> [*Float*]

In paragraaf 2.3.6 werd de functie map besproken. Deze functie heeft twee parameters: een functie en een lijst. De functie wordt op alle elementen van de lijst toegepast, zodat het resultaat ook weer een lijst is. Het type van map is als volgt: p. 38

map :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

De eerste parameter van map is een functie tussen willekeurige types (a en b), die niet eens hetzelfde hoeven te zijn. De tweede parameter van map is een lijst, waarvan de elementen hetzelfde type (a) moeten hebben als de parameter van de functieparameter (die functie moet er immers op toegepast kunnen worden). Het resultaat van map is een lijst, waarvan de elementen hetzelfde type (b) hebben als het resultaat van de functieparameter.

In de typedeclaratie van map moeten er haakjes staan om het type van de eerste parameter ($a \rightarrow b$). Anders zou er staan dat map drie parameters heeft: een a, een b, een $[a]$ en een $[b]$ als resultaat. Dat is natuurlijk niet de bedoeling: map heeft twee parameters: een ($a \rightarrow b$) en een $[a]$.

Ook operatoren hebben een type. Operatoren zijn tenslotte gewoon functies met twee parameters die op een afwijkende manier genoteerd worden (tussen de parameters in plaats van ervoor). Voor het type maakt dat niet uit. Er geldt dus bijvoorbeeld:

(&&) :: *Bool* -> *Bool* -> *Bool*

2.5.5 Overloading

De operator + kan gebruikt worden op twee gehele getallen (*Int*) of op twee floating point getallen (*Float*). Het resultaat is weer van datzelfde type. Het type van + kan dus zowel *Int* -> *Int* -> *Int* als *Float* -> *Float* -> *Float* zijn. Toch is + niet echt een (parametrisch) polymorfe operator zoals we die eerder al zagen: als het type a -> a -> a zou zijn, zou de operator namelijk ook op bijvoorbeeld *Bool* parameters moeten werken, en op lijsten van functies. Voor deze vorm van polymorfie wordt de naam ad-hoc polymorfie of overloading gebruikt. Hoewel overloading misschien wel wat lijkt op parametrische polymorfie is er ook een essentieel verschil: omdat de optelling van twee integers anders in zijn werk gaat dan de optelling van twee booleans, moeten we uiteindelijk voor elk type waarvoor we + willen gebruiken een specifieke definitie geven. Dit staat in contrast met parametrische polymorfie waarbij een enkele definitie (uniform) werkt voor een grote hoeveelheid types.

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*). Een klasse is een groep types met een gemeenschappelijk kenmerk. In de prelude worden alvast een paar klassen gedefinieerd:

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- *Ord* is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);
- *Eq* is de klasse van types waarvan de elementen met elkaar vergeleken kunnen worden (equality types).
- Integral is de klasse van types waarvan de elementen gehele getallen zijn, dus *Int* en *Integer*

De operator + heeft nu het volgende type:

$$(+) :: Num\ a \Rightarrow a \to a \to a$$

Dit dient gelezen te worden als: "+ heeft het type a -> a -> a mits a een type is in de klasse *Num*".

Let op het gebruik van het pijltje met dubbele stok. Dit heeft een heel andere betekenis dan een pijltje met enkele stok. Zo'n dubbel pijltje kan maar één keer in een type staan.

Andere voorbeelden van overloaded operatoren zijn:

$$(<) :: Ord\ a \Rightarrow a \to a \to Bool$$
$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$

Zelfgedefinieerde functies kunnen ook overloaded zijn. Bijvoorbeeld de functie

```
kwadraat x = x * x
```

heeft het type

kwadraat :: $Num\ a \Rightarrow a \rightarrow a$

doordat de operator $*$ die erin gebruikt wordt overloaded is.

Ook constanten zijn overloaded. Bijvoorbeeld:

```
? :t 3
3 :: Num a => a
```

Dat betekent dat de constanten `3` overal gebruikt kan worden waar een numeriek type wordt verwacht. Deze overloading strekt zich uit tot lijsten:

```
:t [1,2,3]
[1,2,3] :: Num a => [a]
```

Het gebruik van klassen en de definitie ervan wordt uitgebreid besproken in hoofdstuk 10. p. 189
Klassen werden hier alleen kort genoemd om de overloaded operatoren te kunnen typeren.

Opgaven

2.1 The language Haskell is named after Haskell B. Curry. Who was he? (Look it up on the internet).

2.2 How can occurrences of the tokens below be classified in a program text? Are they
- something with a fixed meaning (reserved word or symbol);
- name of a function or a parameter;
- an operator;
- none of the above?

If it is a function or an operator, is it a datatype constructor then?

```
=>    3a    a3a   ::    :=
:e    X_1   <=>   a'a   _X
***   'a'   A     in    :-<
```

2.3 Calculate:

4.0e3 + 2.0e-2
4.0e3 $*$ 2.0e-2
4.0e3 / 2.0e-2

2.4 How do the expressions `x = 3` and `x == 3` differ in their meaning?

2.5 Write two version of a function noOfSol that, for some a, b, and c, determines the number of solutions of the equation $ax^2 + bx + c = 0$:
- with case distinction
- by combining standard functions

2.6 What is the advantage of using nested comments (see paragraph 2.4.5)? p. 45

2.7 What is the type of the following functions? tail, sqrt, pi, exp, (^), (/=) en noOfSol? (Note that the correct answers (those that you would get if asked the types of these functions from the interpreter) contain type classes, which have not been explained yet. You may give more specific types using Integer and Double.) How can you query the interpreter for the type of an expression and how can you explicitly specify the types of functions in your program?

2.8 Assume that x has the value 5. What is the value of the expression x == 3 and x/= 3? (If you are familiar with the programming lanugage C: what is the value of these expressions in C?)

2.9 What does "*syntax error*" mean? What is the difference between a *syntax error* and a *type error*?

2.10 Determine the types of 3, even, and even 3. How can you figure out the latter? Determine also the type of head, $[1, 2, 3]$, and head $[1, 2, 3]$. What happens when applying a polymorphic function to monomorphic arguments?

2.11 Try to guess the type of the following expressions and verify with `ghci`:
- until even
- until or
- foldr (&&) *True*
- foldr (&&)
- foldr until
- map sqrt
- map filter
- map map

2.12 In paragraph 2.4.4 it says that a recursive function is only sensible if the condition is met that the value of its parameters becomes simpler in each recursive application. Consider the following definition of the faculty function: p. 42

```
fac n | n == 0    = 1
      | otherwise = n * fac (n - 1)
```

a. What happens if you evaluate fac (-3)?

b. How can you formulate the condition more precisely?

2.13 What is the difference between a *list* and a *set* as used in mathematics?

2.14 In paragraaf 2.4.4 a recursive function definition for exponentiation is given. p. 42

a. Give an alternative definition for exponentiation that treats the two cases where n is even and where n is uneven separately. You can exploit the fact that $x^n = (x^{n/2})^2$.

b. Which intermediate results are being computed for the computation of 2^{10} in the old and the new definition?

2.15 Given the following definitions:

$$\begin{aligned}
&\mathsf{thrice\ x} &&= [\mathsf{x}, \mathsf{x}, \mathsf{x}] \\
&\mathsf{sums\ \ (x : y : ys)} &&= \mathsf{x} : \mathsf{sums\ (x + y : ys)} \\
&\mathsf{sums\ \ xs} &&= \mathsf{xs}
\end{aligned}$$

What does the following expression evaluate to?

$$\mathsf{map\ thrice\ (sums\ [0\,..\,4])}$$

3 Getallen en functies

3.1 Operatoren

3.1.1 Operatoren als functies en andersom

Een operator is een functie met twee parameters die tussen de parameters wordt geschreven in plaats van er voor. Namen van functies bestaan uit letters en cijfers, "namen" van operatoren uit symbolen (zie paragraaf 2.3.2 voor de precieze regels voor naamgeving). p. 32

Soms is het gewenst om een operator toch vóór de parameters te schrijven, of een functie er tussen. In Haskell zijn daar twee speciale notaties voor beschikbaar:

- een operator tussen haakjes gedraagt zich als de overeenkomstige functie;
- een functie tussen *back quotes* gedraagt zich als de overeenkomstige operator.

(Een "back quote" is het symbool ‘, vooral niet te verwarren met de ’, de *apostrof*. Op de meeste toetsenborden zit de backquote-toets links van de 1-toets (bij Macs nogal eens links van de Z), en de apostrof links van de entertoets.)

Het is dus toegestaan (+) 1 2 te schrijven in plaats van 1 + 2. Deze notatie is in paragraaf 2.5.5 ook gebruikt om het type van + te kunnen declareren: p. 52

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

Voor de :: moet namelijk een expressie staan; een losse operator is geen expressie, maar een functie wel. Het onderdeel *Num a* geeft aan dat er meerdere functies met de naam (+) kunnen bestaan. De Haskell prelude bevat er dan ook al een flink aantal, zoals voor *Float*s, voor *Int*s (gehele getallen in de machine representatie) en voor *Integer*s, gehele getallen van onbeperkte grootte. [1]

Andersom is het mogelijk om 1 ‘f‘ 2 te schrijven in plaats van f 1 2. Dit wordt vooral gebruikt om een expressie overzichtelijker te maken; de expressie 5 ‘choose‘ 3 leest nu eenmaal makkelijker dan choose 5 3. Dit kan natuurlijk alleen als de functie twee parameters heeft.

[1] Ooit stond in de krant dat de computer van de Amerikaanse belastingdienst problemen had met het correct representeren van het vermogen van Bill Gates (zo rond de $\$38 * 10^9$), met als gevolg een eindeloze rij aanmaningen en excuus brieven. Ga eens na met hoeveel bits vermogens kennelijk gerepresenteerd worden.

3.1.2 Prioriteiten

Iedereen heeft de regel "vermenigvuldigen gaat voor optellen" geleerd, ook wel bekend als "Meneer Van Dalen"[2]. Je kunt dit deftiger uitdrukken als: "de prioriteit van vermenigvuldigen is hoger dan die van optellen". Ook in Haskell zijn deze prioriteiten bekend: de expressie $2 * 3 + 4 * 5$ heeft als waarde 26 en niet 50, 46 of 70.

Er zijn in Haskell nog meer prioriteitsnivo's. De vergelijkingsoperatoren, zoals $<$ en $==$, hebben een lagere prioriteit dan de rekenkundige. Zo heeft $3 + 4 < 8$ de betekenis die je ervan zou verwachten: 3+4 wordt met 8 vergeleken (resultaat *True*), en niet: 3 wordt opgeteld bij het resultaat van $4 < 8$ (dat zou een typeringsfout opleveren).

In totaal zijn er negen nivo's van prioriteiten. De operatoren in de prelude hebben de volgende prioriteit (tussen haakjes staat hoe zin programmatekst staan):

nivo 9	. en !!
nivo 8	^
nivo 7	$*$, /, ‘div‘, ‘rem‘ en ‘mod‘
nivo 6	$+$ en $-$
nivo 5	:, ++ en \\
nivo 4	$==$, $/=$,$<$, $<=$, $>$, $>=$, ‘elem‘ en ‘notElem‘
nivo 3	&&
nivo 2	||
nivo 1	(niet gebruikt in de prelude)

(Nog niet al deze operatoren zijn aan de orde geweest; sommige worden in dit of een volgend hoofdstuk besproken.) Vermenigvuldigen en delen hebben dus dezelfde prioriteit; Nederland schijnt alleen te staan in de voorrang van vermenigvuldigen op delen.

Om af te wijken van de geldende prioriteiten kunnen in een expressie haakjes geplaatst worden rond de deelexpressies die eerst uitgerekend moeten worden: in $2 * (3 + 4) * 5$ wordt wèl eerst $3 + 4$ uitgerekend.

De allerhoogste prioriteit wordt gevormd door het aanroepen van functies (de "onzichtbare" operator tussen f en x in f x). De expressie sqr $3 + 4$ berekent dus het kwadraat van 3, en telt daar 4 bij op. Om het kwadraat van 7 te bepalen zijn haakjes nodig om de hoge prioriteit van functieaanroep te doorbreken: sqr $(3 + 4)$.

Ook bij het definiëren van functies met gebruik van patronen (zie paragraaf 2.4.3) is het van belang te bedenken dat functieaanroep altijd voor gaat. In de definitie p. 41

```
sum []     = 0
sum (x:xs) = x + sum xs
```

[2] Het zinnetje "Meneer Van Dalen Wacht Op Antwoord" is een ezelsbruggetje voor deze regel: de beginletters komen overeen met die van Machtsverheffen, Vermenigvuldigen, Delen, Worteltrekken, Optellen en Aftrekken. In dit ezelsbruggetje zit niet verwerkt dat optellen en aftrekken gelijkwaardig zijn. Sommigen leerden daarom het rijmpje: "vermenigvuldigen gaat altijd voor / daarna komt altijd delen door / daarna komt altijd min of plus / geen van die twee heeft voorrang dus".

zijn de haakjes rond `x : xs` essentieel; zonder haakjes zou dit immers opgevat worden als `(sum x) : xs`, en dat is geen geldig patroon.

3.1.3 Associatie

Met de prioriteitsregels ligt nog steeds niet vast wat er moet gebeuren met operatoren van gelijke prioriteit. Voor optelling maakt dat niet uit, maar voor bijvoorbeeld aftrekken is dat wel belangrijk: is de uitkomst van $8 - 5 - 1$ de waarde 2 (eerst 8 min 5, en dan min 1) of 4 (eerst 5 min 1, en dat aftrekken van 8)?

Voor elke operator wordt in Haskell vastgelegd hoe deze *associeert*. Voor een operator, laten we zeggen $\oplus$, zijn er vier mogelijkheden:

- de operator $\oplus$ *associeert naar links*, dat wil zeggen $a \oplus b \oplus c$ is equivalent aan $(a \oplus b) \oplus c$; [3]
- de operator $\oplus$ *associeert naar rechts*, dat wil zeggen $a \oplus b \oplus c$ is equivalent aan $a \oplus (b \oplus c)$;
- de operator $\oplus$ is *associatief*, dat wil zeggen het maakt niet uit in welke volgorde $a \oplus b \oplus c$ wordt uitgerekend;
- de operator $\oplus$ is *non-associatief*, dat wil zeggen dat het verboden is om $a \oplus b \oplus c$ te schrijven; je moet dus altijd met haakjes aangeven wat de bedoeling is.

Voor de operatoren in de prelude is de keuze overeenkomstig de wiskundige traditie gemaakt. In geval van twijfel zijn de prelude-operatoren non-associatief gemaakt. Voor de associatieve operatoren is toch een keuze gemaakt voor links- of rechtsassociatief. Daarbij is de keuze gevallen op de meest efficiënte volgorde. Je hoeft daar geen rekening mee te houden, want voor het eindresultaat maakt het toch niet uit.

De volgende operatoren associëren naar **links**:

- de "onzichtbare" operator *functieapplicatie*, dus `f x y` moet gelezen worden als `(f x) y` (de reden wordt besproken in sectie 3.2), p. 60
- de operator !! (zie paragraaf 4.1.2), p. 84
- de operator $-$, dus de waarde van $8-5-1$ is 2 (zoals gebruikelijk in de wiskunde) en niet 4,
- de operator / en de verwante operatoren `div`, `rem` en `mod`.

De volgende operatoren associëren naar **rechts**:

- de operator `^` (machtsverheffen), dus de waarde van `2^2^3` is $2^8 = 256$ (zoals gebruikelijk in de wiskunde) en niet $4^3 = 64$,
- de operator : ("zet op kop van"), zodat de waarde van `1 : 2 : 3 : x` een lijst is die begint met de waarden 1, 2 en 3.

De volgende operatoren zijn **non**-associatief:

- de operator `\\` (zie opgave 4.16), p. 114

[3] Ezelsbruggetje: wil je weten hoe een operator, zeg $\oplus$, associeert, schrijf dan eerst $a \oplus b \oplus c$. Als b eerst met de a *links* van hem wil, dan is $\oplus$ *linksassociatief*.

- de vergelijkingsoperatoren ==, < enzovoort: het is niet toegestaan om $a == b == c$ te schrijven. Probeer je dat toch dan krijg je de volgende melding:

```
Precedence parsing error
   cannot mix ‘==’ [infix 4] and ‘==’ [infix 4] in the
      same infix expression
```

  Wil je testen of x tussen 2 en 8 ligt, schrijf dan niet $2 < x < 8$ zoals men soms in de wiskunde doet, maar $2 < x \;\&\&\; x < 8$.

De volgende operatoren zijn associatief:

- de operatoren $*$ en $+$ (deze operatoren worden overeenkomstig wiskundige traditie linksassociërend uitgerekend),
- de operatoren ++, && en || (deze operatoren worden rechtsassociërend uitgerekend omdat dat efficiënter is),
- de operator voor functiecompositie . (zie paragraaf 3.3.3). p. 66

3.1.4 Definitie van operatoren

Wie zelf een operator definieert, kan daarbij aangeven wat de prioriteit is, en op welke manier de associatie plaatsvindt. In de prelude staat gespecificeerd dat ^ prioriteitsnivo 8 heeft en naar rechts associeert:

infixr 8 ^

Voor operatoren die naar links associëren dient het gereserveerde woord **infixl**, en voor non-associatieve operatoren het woord **infix**:

infixl 6 +, −
infix 4 ==, /=, ‘elem‘

Door een slimme keuze voor de prioriteit te maken, kunnen haakjes in expressies zo veel mogelijk worden vermeden. We bekijken nog eens de operator ‘choose‘:

n ‘choose‘ k = fac n / (fac k * fac (n − k))

of met een zelfbedacht symbool:

n !^! k = fac n / (fac k * fac (n − k))

Omdat je misschien wel eens $\binom{a+b}{c}$ wilt berekenen, is het handig om ‘choose‘ een lagere prioriteit te geven dan +; je kunt dan $a + b$ ‘choose‘ c schrijven zonder haakjes. Aan de andere kant zijn expressies als $\binom{a}{b} < \binom{c}{d}$ gewenst. Door ‘choose‘ een hogere prioriteit te geven dan <, zijn ook hierbij geen haakjes nodig en kun je dus direct schrijven a ‘choose‘ $b < c$ ‘choose‘ d.

Voor de prioriteit van ‘choose‘ kan dus het beste 5 gekozen worden (lager dan + (6), maar hoger dan < (4)). Wat betreft de associatie: omdat het weinig gebruikelijk is om

a ‘choose‘ b ‘choose‘ c uit te rekenen, kan de operator het beste non-associatief gemaakt worden. De prioriteitsdefinitie luidt al met al:

infix 5 !^!, ‘choose‘

3.2 Currying

3.2.1 Partieel parametriseren

Stel dat plus een functie is die twee gehele getallen optelt. Je zou dan verwachten dat plus altijd twee argumenten krijgt, zoals bijvoorbeeld in plus 3 5.

In Haskell mag je ook *minder* argumenten aan een functie meegeven. Als plus maar één argument krijgt, bijvoorbeeld plus 1, dan houd je een functie over die nog een argument verwacht. Deze functie kan bijvoorbeeld gebruikt worden om een andere functie te definiëren:

successor :: *Int* -> *Int*
successor = plus 1

Het aanroepen van een functie met minder argumenten dan deze verwacht heet *partieel parametriseren*.

Een tweede toepassing van een partieel geparametriseerde functie is dat deze als argument kan dienen voor een andere functie. De functieparameter van de functie map (die een functie toepast op alle elementen van een lijst) is bijvoorbeeld vaak een partieel geparametriseerde functie:

```
? map (plus 5) [1,2,3]
[6, 7, 8]
```

De expressie plus 5 kun je beschouwen als “de functie die 5 ergens bij optelt”. Deze functie wordt in het voorbeeld door map op alle elementen van de lijst $[1, 2, 3]$ toegepast.

De mogelijkheid van partiële parametrisatie werpt een nieuw licht op het type van plus. Als plus 1, net zoals de functie successor, het type *Int* -> *Int* heeft, dan is plus zelf blijkbaar een functie van *Int* (het type van 1) naar dat type:

plus :: *Int* -> (*Int* -> *Int*)

Door af te spreken dat -> naar rechts associeert, zijn de haakjes hierin overbodig:

plus :: *Int* -> *Int* -> *Int*

Dit is precies de notatie voor het type van een functie met twee parameters, die in paragraaf 2.5.4 werd besproken. p. 51

Een gevolg hiervan is dat er in Haskell eigenlijk helemaal geen "functies met twee parameters" bestaan: alle functies hebben precies één parameter, en eventueel kunnen ze weer een functie als resultaat opleveren. Dit resultaat heeft op zijn beurt weer een parameter, zodat het lijkt alsof de oorspronkelijke functie er één extra heeft.

Deze truc, het simuleren van functies met meer parameters door een functie met één parameter die een functie oplevert, wordt *Currying* genoemd, naar de Engelse wiskundige Haskell Curry. De functie zelf heet een *gecurryde* functie. (Dit eerbetoon is niet helemaal terecht, want de methode werd eerder gebruikt door M. Schönfinkel.)

3.2.2 Haakjes

De "onzichtbare operator" functietoepassing associeert naar links. Dat wil zeggen: de expressie plus 1 2 wordt door de interpreter opgevat als (plus 1) 2. Dat klopt precies met het type van plus: dit is immers een functie die een integer verwacht (1 in het voorbeeld) en dan een functie oplevert, die op zijn beurt een integer kan verwerken (2 in het voorbeeld). Associatie van functietoepassing naar rechts zou onzin zijn: in plus (1 2) zou eerst 1 op 2 worden toegepast en vervolgens plus op het resultaat.

Staan er in een expressie een hele rij letters op een rij, dan moet de eerste daarvan een functie zijn die de andere achtereenvolgens als parameter opneemt:

f a b c d

wordt opgevat als

((((f a) b) c) d)

Als a type A heeft, b type B enzovoort, dan is het type van f:

f :: A -> B -> C -> D -> E

of, als je alle haakjes zou schrijven:

f :: A -> (B -> (C -> (D -> E)))

Zonder haakjes is dit alles natuurlijk veel overzichtelijker. De associatie van -> en functieapplicatie is daarom zó gekozen, dat Currying "geruisloos" verloopt: functieapplicatie associeert naar links, en -> associeert naar rechts.

Haakjes zijn alleen nodig, als je hiervan wilt afwijken. Dat gebeurt bijvoorbeeld in de volgende gevallen:

- In het type als een functie een functie als *parameter* krijgt (bij Currying heeft een functie een functie als *resultaat*). Het type van map is bijvoorbeeld

  map :: (a -> b) -> [a] -> [b]

  De haakjes in (a -> b) zijn essentieel, anders zou map een functie met drie parameters zijn, en dat is niet zo.

- In een expressie als het *resultaat* van een functie aan een andere functie wordt meegegeven, en niet de functie zelf. Bijvoorbeeld, als je het kwadraat van de sinus van een getal wilt uitrekenen:

  sqrt (sin pi)

  Zouden hier de haakjes ontbreken, dan lijkt het alsof sqrt eerst op sin wordt toegepast en het resultaat daarvan vervolgens op pi.

3.2.3 Operatorsecties

Voor het partieel parametriseren van operatoren zijn twee speciale notaties beschikbaar:
- met $(\oplus x)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *rechter* parameter;
- met $(x\oplus)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *linker* parameter.

Deze notaties heten *operatorsecties*.

Met operatorsecties kunnen een aantal functies gedefinieerd worden:

successor	$= (+1)$
verdubbel	$= (2*)$
halveer	$= (/2.0)$
omgekeerde	$= (1.0/)$
kwadraat	$= (\hat{}2)$
tweeTotDe	$= (2\hat{})$
eencijferig	$= (<= 9)$
isNul	$= (== 0)$

De belangrijkste toepassing van operatorsecties is echter het meegeven van zo'n partieel geparametriseerde operator aan een functie:

```
? map (2*) [1,2,3]
[2, 4, 6]
```

3.3 Functies als parameter

3.3.1 Functies op lijsten

In een functionele programmeertaal gedragen functies zich in veel opzichten hetzelfde als andere waarden, zoals getallen en lijsten. Bijvoorbeeld:
- functies hebben een *type*;

- functies kunnen door andere functies worden opgeleverd als *resultaat* (waarvan met Currying veel gebruik wordt gemaakt);
- functies kunnen als *argument* van andere functies worden meegegeven.

Dit laatste speelt een belangrijke rol in het functioneel programmeren. Telkens wanneer we twee functies hebben geschreven waarvan gedeeltes van de definitie gemeenschappelijk is, kunnen we de verschillen uitfactoriseren en tot een parameter maken. Dit proces is heel gebruikelijk bij functioneel programmeuren. Voordelen zijn dat het tot meer hergebruik leidt, en vaak een goed inzicht geeft in de essentie van een algoritme. Door hoofd- en bijzaken goed te scheiden kunnen zeer algemeen toepasbare bibliotheken worden geschreven.

Functies met functieparameters worden *hogere-orde functies* genoemd, om ze te onderscheiden van de overige eerste-orde functies.

De functie map is een voorbeeld van een hogere-orde functie. Deze functie verzorgt het algemene principe "alle elementen van een lijst langsgaan". Wat er met de elementen van de lijst moet gebeuren, wordt aangegeven door een functie die, naast de lijst, als parameter aan map wordt meegegeven.

De functie map is als volgt gedefinieerd:

$$
\begin{array}{ll}
\mathsf{map} & :: (a \mathrel{->} b) \mathrel{->} [a] \mathrel{->} [b] \\
\mathsf{map\ f\ [\,]} & = [\,] \\
\mathsf{map\ f\ (x : xs)} & = \mathsf{f\ x} : \mathsf{map\ f\ xs}
\end{array}
$$

De definitie maakt gebruik van patronen: de functie wordt apart gedefinieerd voor het geval dat het tweede argument de lege lijst is, en voor het geval dat de lijst bestaat uit een eerste element x en een rest xs. De functie is recursief: in het geval van een niet-lege lijst wordt de functie map opnieuw aangeroepen. De parameter is daarbij korter (xs is korter dan x : xs) zodat uiteindelijk het niet-recursieve deel van de functie gebruikt zal kunnen worden.

Een andere veel gebruikte hogere-orde functie op lijsten is filter. Deze functie levert díe elementen uit een lijst, die aan een bepaalde eigenschap voldoen. Welke eigenschap dat is, wordt bepaald door een functie die als parameter aan filter wordt meegeven. Voorbeelden van het gebruik van filter zijn:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

Als de lijstelementen van type a zijn, heeft de functieparameter van filter het type $a \mathrel{->} Bool$. Net als van map is de definitie van filter recursief:

```
filter                        :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x : xs) | p x       = x : filter p xs
                  | otherwise = filter p xs
```

In het geval dat de lijst niet leeg is (dus de vorm x : xs heeft), worden de gevallen onderscheiden dat het eerste element x aan de eigenschap p voldoet, of niet. Zo ja, dan wordt dit element in ieder geval in het resultaat gezet; de andere elementen worden (met een recursieve aanroep) "door het filter gehaald".

Bruikbare hogere-orde functies kun je op het spoor komen door de overeenkomst in functiedefinities op te sporen. Bekijk bijvoorbeeld de definities van de functies sum (die de som van een lijst getallen berekent), product (die het product van een lijst getallen berekent) en and (die kijkt of een lijst Boolse waarden allemaal *True* zijn):

```
sum     []       = 0
sum     (x : xs) = x + sum xs
product []       = 1
product (x : xs) = x * product xs
and     []       = True
and     (x : xs) = x && and xs
```

De structuur van deze drie definities is hetzelfde. Het enige wat verschilt, is de waarde die er bij een lege lijst uitkomt (0, 1 of *True*), en de operator die gebruikt wordt om het eerste element te koppelen aan het resultaat van de recursieve aanroep (+, $*$ of &&).

Door deze twee veranderlijken als parameter mee te geven, ontstaat een algemeen bruikbare hogere-orde functie:

```
foldr op e []       = e
foldr op e (x : xs) = x `op` foldr op e xs
```

Gegeven deze functie, kunnen de andere drie functies gedefinieerd worden door de algemene functie partieel te parametriseren:

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
```

De functie foldr is in veel meer gevallen bruikbaar; daarom is hij als standaardfunctie in de prelude gedefinieerd.

De naam van foldr laat zich als volgt verklaren. De waarde van

```
foldr (+) e (w : (x : (y : (z : []))))
```

is gelijk aan de waarde van de expressie

```
(w + (x + (y + (z + e))))
```

De functie foldr "vouwt" de lijst ineen tot één waarde, door alle "op-kop-van" voorkomens te vervangen door de gegeven operator, en de lege lijst ([]) die helemaal aan de rechter kant staat te vervangen door de startwaarde. (Er bestaat ook een functie foldl die aan de linkerkant begint).

Hogere-orde functies, zoals map en foldr, spelen in functionele talen de rol die controlestructuren (zoals for en while) in imperatieve talen spelen. Die controlestructuren zijn echter "ingebouwd", terwijl de functies zelf gedefinieerd kunnen worden. Dit maakt functionele talen flexibel: er is weinig ingebouwd, maar je kunt alles zelf maken (of uit een bibliotheek halen).

3.3.2 Iteratie

In de wiskunde wordt vaak *iteratie* gebruikt. Dit houdt in: neem een startwaarde, en pas daarop net zolang een functie toe, tot het resultaat aan een bepaalde eigenschap voldoet.

Iteratie is goed te beschrijven met een hogere-orde functie. In de prelude wordt deze functie until genoemd. Het type is:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
```

De functie heeft drie parameters: de eigenschap waar het eindresultaat aan moet voldoen (een functie $a \rightarrow Bool$), de functie die steeds wordt toegepast (een functie $a \rightarrow a$), en de startwaarde (van type a). Het eindresultaat is ook van type a. De aanroep until p f x kan gelezen worden als: "pas net zo lang f toe op x totdat het resultaat voldoet aan p".

De definitie van until is weer recursief. Het recursieve en het niet-recursieve geval worden ditmaal niet onderscheiden door patronen, maar door gevalsonderscheid met guards (zie hoofdstuk 2): p. 21

```
until p f x | p x       = x
            | otherwise = until p f (f x)
```

Als de startwaarde x meteen al aan de eigenschap p voldoet, dan is de startwaarde tevens de eindwaarde. Anders wordt de functie f éénmaal op x toegepast. Het resultaat, (f x), wordt gebruikt als nieuwe startwaarde in de recursieve aanroep van until.

Zoals alle hogere-orde functies kan until goed aangeroepen worden met partieel geparametriseerde functies. Onderstaande expressie berekent bijvoorbeeld de eerste macht van twee die groter of gelijk is aan 1000 (begin met 1 en verdubbel net zo lang, tot 1000 kleiner is dan het resultaat):

```
? until (>1000) (2*) 1
1024
```

Anders dan bij eerder besproken recursieve functies, is de parameter van de recursieve aanroep van until niet "kleiner" dan de formele parameter. Daarom levert until niet altijd

een resultaat op. Bij de aanroep until (== 0) (+1) 1 wordt aan de voorwaarde nooit voldaan; de functie until zal dus tot in de eeuwigheid blijven doortellen, en dus nooit met een resultaat komen.

Als de computer steeds maar geen antwoord geeft omdat hij in zo'n oneindige recursie terecht is gekomen, kan de berekening worden afgebroken door tegelijkertijd de Ctrl-toets en de C-toets in te drukken:

```
until (== 0) (+1) 1
ctrl-C
Interrupted!
?
```

3.3.3 Samenstelling (functiecompositie)

Als f en g functies zijn, dan is $f \circ g$ de wiskundige notatie voor "f na g": de functie die eerst g toepast, en daarna f op het resultaat van deze aanroep. Ook in Haskell komt de operator die twee functies samenstelt goed van pas. Als er zo'n operator ‘after‘ is, dan is het bijvoorbeeld mogelijk om te definiëren:

```
odd      = not `after` even
nearZero = (<10) `after` abs
```

De operator ‘after‘ kan als hogere-orde operator worden gedefinieerd:

```
infixr 8 `after`
f `after` g = h
              where h x = f (g x)
```

Niet alle functies kunnen zomaar worden samengesteld. Het bereik van g moet hetzelfde zijn als het domein van f. Als g dus een functies $a \to b$ is, kan f een functie $b \to c$ zijn. De samenstelling van de twee functies is een functie die van a direct naar c gaat. Dit komt ook tot uiting in het type van after:

```
after :: (b -> c) -> (a -> b) -> (a -> c)
```

Omdat -> naar rechts associeert, is het derde paar haakjes overbodig. Het type van after kan dus ook geschreven worden als

```
after :: (b -> c) -> (a -> b) -> a -> c
```

De functie after kan dus beschouwd worden als functie met drie parameters; door het Currying-mechanisme is dit immers hetzelfde als een functie met twee parameters die een functie oplevert (en hetzelfde als een functie met één parameter die een functie oplevert met één parameter die een functie oplevert). Inderdaad kan after worden gedefinieerd als functie met drie parameters:

```
after f g x = f (g x)
```

Het is dus niet nodig om de functie h apart een naam te geven met een **where**-constructie (al mag dat natuurlijk wel). In de definitie van odd hierboven wordt after dus in feite partieel geparametriseerd met not en even. De derde parameter is nog niet gegeven: deze wordt pas ingevuld als odd wordt aangeroepen.

Het nut van de operator after lijkt misschien beperkt, omdat functies als odd ook gedefinieerd kunnen worden door

```
odd x = not (even x)
```

Een samenstelling van twee functies kan echter als parameter dienen van een andere hogere-orde functie, en dan is het handig dat hij geen naam hoeft te krijgen. De volgende expressie geeft een lijst met de oneven getallen tussen 1 en 100:

```
? filter (not `after` even) [1..100]
```

In de prelude wordt de functiesamenstellingsoperator gedefinieerd. Hij wordt genoteerd als punt (omdat het teken ∘ nu eenmaal niet op het toetsenbord zit). Je kunt dus schrijven:

```
? filter (not.even) [1..100]
```

Deze operator komt vooral goed tot zijn recht als er veel functies samengesteld worden. Het programmeren kan dan geheel op functienivo plaatsvinden (vandaar ook de titel van dit diktaat). Laag-bij-de-grondse dingen als getallen en lijsten zijn uit het gezicht verdwenen. Is het niet veel mooier om f = g . h . i . j . k te kunnen schrijven in plaats van f x = g (h (i (j (k x)))))?

3.3.4 De lambda-notatie (anonieme functies)

In paragraaf 3.2.1 werd opgemerkt dat de functie die je als parameter meegeeft aan een andere functie vaak ontstaat door partiële parametrisatie: p. 60

```
map (plus 5) [1..10]
map (*2) [1..10]
```

In andere gevallen kan de functie die als parameter wordt meegegeven geconstrueerd worden door andere functies samen te stellen:

```
filter (not . even) [1..10]
```

Maar soms is het te ingewikkeld om de functie op die manier te maken, bijvoorbeeld als we $x^2 + 3x + 1$ willen uitrekenen voor alle x'en in een lijst. Het is dan altijd mogelijk om de functie apart te definiëren in een **where**-clausule:

```
ys = map f [1..100]
  where f x = x*x + 3*x + 1
```

Als dit veel voorkomt is het echter een beetje vervelend dat je steeds een naam moet verzinnen voor de functie, en die dan achteraf definiëren.

Voor dit soort situaties is er een speciale notatie beschikbaar, waarmee functies kunnen worden gecreëerd zonder die een naam te geven. Dit is dus vooral van belang als de functie alleen maar nodig is om als parameter meegegeven te worden aan een andere functie. De notatie is als volgt:

`\` *patroon* `->` *expressie*

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool `\` is de beste benadering voor die letter die op het toetsenbord beschikbaar is...)

Een voorbeeld van de lambda-notatie is de functie `\x -> x*x+3*x+1`. Dit kun je lezen als: "de functie die bij parameter x de waarde $x^2 + 3x + 1$ oplevert". De lambda-notatie wordt veel gebruikt bij het meegeven van functies als parameter aan andere functies, bijvoorbeeld:

```
ys = map (\x -> x*x + 3*x + 1) [1..100]
```

3.4 Numerieke functies

3.4.1 Rekenen met gehele getallen

Bij deling van gehele getallen (*Int*) gaat het gedeelte achter de komma verloren: 10 / 3 is 3. Toch is het niet nodig bij delingen dan maar altijd *Float* getallen te gebruiken. Integendeel: vaak is de *rest* van de deling interessanter dan de decimale breuk. De rest van een deling is het getal dat op de laatste regel van een staartdeling staat. Bijvoorbeeld in de deling 345 / 12

```
1 2 / 3 4 5 \ 2 8
      2 4
      -----
      1 0 5
        9 6
      -----
          9
```

is het quotiënt 28 en de rest 9.

De rest van een deling kan bepaald worden met de standaardfunctie `rem` (*remainder*):

```
? 345 `rem` 12
9
```

De rest van een deling is bijvoorbeeld in de volgende gevallen van nut:

- Rekenen met tijden. Als het nu bijvoorbeeld 9 uur is, dan is het 33 uur later $(9 + 33)$ ‘rem‘ $24 = 20$ uur.
- Rekenen met weekdagen. Codeer de dagen als 0=zondag, 1=maandag, ..., 6=zaterdag. Als het nu dag 3 is (woensdag), dan is het over 40 dagen $(3 + 40)$ ‘rem‘ $7 = 1$ (maandag).
- Bepalen van deelbaarheid. Een getal is deelbaar door n als de rest bij deling door n gelijk aan nul is.
- Bepalen van losse cijfers. Het laatste cijfer van een getal x is x ‘rem‘ 10. Het op één na laatste getal is (x / 10) ‘rem‘ 10. Het op twee na laatste (x / 100) ‘rem‘ 10, enzovoort.

Als een wat uitgebreider voorbeeld van het rekenen met gehele getallen volgen hier twee toepassingen: het berekenen van een lijst priemgetallen, en het bepalen van de dag van de week gegeven de datum.

Berekenen van een lijst primes

Een getal is deelbaar door een ander getal als de rest bij deling door dat getal gelijk aan nul is. De functie divisible test twee getallen op deelbaarheid:

```
divisible     :: Int -> Int -> Bool
divisible t n = t `rem` n == 0
```

De delers van een getal zijn de getallen waardoor een getal deelbaar is. De functie divisors bepaalt de lijst delers van een getal:

```
divisors   :: Int -> [Int]
divisors x = filter (divisible x) [1..x]
```

De functie divisible wordt hierin partieel geparametriseerd met x; door de aanroep van filter worden díe elementen uit $[1..x]$ gefilterd, waardoor x deelbaar is.

Een getal is een priemgetal als het precies twee delers heeft: 1 en zichzelf. De functie prime kijkt of de lijst delers inderdaad uit deze twee elementen bestaat:

```
prime   :: Int -> Bool
prime x = (divisors x) == [1,x]
```

De functie primes tenslotte bepaalt alle priemgetallen tot een gegeven bovengrens:

```
primes   :: Int -> [Int]
primes x = filter prime [1..x]
```

Hoewel dit misschien niet de meest efficiënte manier is om priemgetallen te berekenen, is het qua programmeerwerk wel de makkelijkste: de functies zijn een directe vertaling van de wiskundige definities.

Bepalen van de dag van de week

Op welke dag valt het laatste oudjaar van de twintigste eeuw?

```
? dag 31 12 1999
vrijdag
```

Als het nummer van de dag bekend is (volgens de hierboven genoemde codering 0=zondag enz.) dan is de functie dag vrij eenvoudig te schrijven:

dag d m j = weekDay (dayNo d m j)

weekDay 0 = "zondag"
weekDay 1 = "maandag"
weekDay 2 = "dinsdag"
weekDay 3 = "woensdag"
weekDay 4 = "donderdag"
weekDay 5 = "vrijdag"
weekDay 6 = "zaterdag"

De functie weekDay gebruikt zeven patronen om de juiste tekst te selecteren (een woord tussen aanhalingstekens (") is een tekst; voor de details zie paragraaf 4.2.1). p. 94

De functie dayNo kiest een zondag in een ver verleden en telt vervolgens op:

- het aantal sindsdien verstreken jaren maal 365;
- een correctie voor de verstreken schrikkeljaren;
- de lengtes van de dit jaar al verstreken maanden;
- het aantal dagen in de lopende maand.

Van het resulterende (grote) getal wordt de rest bij deling door 7 bepaald: dat is het gevraagde dagnummer.

Sinds de kalenderhervorming van paus Gregorius in 1582 (die in het anti-paapse Nederland en Engeland overigens pas in 1752 werd geaccepteerd) geldt de volgende regel voor schrikkeljaren (jaren met 366 dagen):

- een jaartal deelbaar door 4 is een schrikkeljaar (bijv. 1972);
- uitzondering: als het deelbaar is door 100 is het geen schrikkeljaar (bijv. 1900);
- uitzondering op de uitzondering: als het deelbaar is door 400 is het tòch een schrikkeljaar (bijv. 2000).

Als nulpunt van de dagnummers zouden we de dag van de kalenderhervorming kunnen kiezen, maar het is eenvoudiger om terug te extrapoleren tot het fictieve jaar 0. De functie dayNo is dan namelijk simpeler: de 1e januari van het jaar 0 zou op een zondag zijn gevallen.

```
dayNo d m j = (   (j − 1) ∗ 365
                + (j − 1) / 4
                − (j − 1) / 100
                + (j − 1) / 400
                + sum (take (m − 1) (months j))
                + d
              ) `rem` 7
```

De aanroep take n xs geeft de eerste n elementen van de lijst xs. De functie take kan gedefinieerd worden door:

```
take 0 xs       = []
take n (x : xs) = x : take (n − 1) xs
```

De functie months moet de lengtes van de maanden in een gegeven jaar opleveren:

```
months j = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  where feb | schrikkel j = 29
            | otherwise   = 28
```

De hierin gebruikte functie schrikkel wordt gedefinieerd volgens de eerder genoemde regels:

```
schrikkel j = divisible j 4 && (not (divisible j 100) || divisible j 400)
```

Een andere manier om dit te definiëren is:

```
schrikkel j | divisible j 100 = divisible j 400
            | otherwise       = divisible j 4
```

Hiermee zijn de functie dag en alle benodigde hulpfuncties voltooid. Het is misschien nog verstandig om in de functie dag op te nemen dat hij alleen gebruikt kan worden voor jaartallen na de kalenderhervorming:

```
dag d m j | j > 1752 = weekDay (dayNo d m j)
```

aanroep van dag met een kleiner jaartal geeft dan automatisch een foutmelding.

(Einde voorbeelden).

Bij de opzet van de twee programma's in de voorbeelden is een verschillende strategie gevolgd. In het tweede voorbeeld werd met de gevraagde functie dag begonnen. Daarvoor waren de hulpfuncties weekDay en dayNo nodig. Voor dayNo was een functie months nodig, en voor months een functie schrikkel. Deze benadering heet *top down*: beginnen met het belangrijkste, en dan steeds "lagere" details invullen.

Het eerste voorbeeld gebruikte de *bottom up* benadering: eerst werd een functie divisible geschreven, met behulp daarvan een functie divisors, daarmee een functie prime, en tenslotte de gevraagde primes.

Voor het eindresultaat maakt het niet uit (het maakt voor de interpreter niet uit in welke volgorde de functies staan). Bij het programmeren is het echter handig om te bedenken of je een top down of een bottom up strategie volgt, of dat deze twee strategieën wellicht afwisselend gebruikt kunnen worden (totdat de "top" de "bottom" raakt).

3.4.2 Numeriek differentiëren

Bij het rekenen met *Float* getallen is een exact antwoord meestal niet haalbaar. De uitkomst van een deling wordt bijvoorbeeld afgerond op een bepaald aantal decimalen (afhankelijk van de rekennauwkeurigheid van de computer):

```
? 10.0 / 6.0
1.6666666666666667
```

Voor de berekening van een aantal wiskundige operaties, zoals sqrt, wordt ook een benadering gebruikt. Bij het schrijven van eigen functies die op *Float* getallen werken is het dan ook acceptabel dat het resultaat een benadering is van de "werkelijke" waarde.

Een voorbeeld hiervan is de berekening van de afgeleide functie. De wiskundige definitie van de afgeleide f' van de functie f is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

De precieze waarde van de limiet kan de computer niet berekenen. Een benadering kan echter worden verkregen door voor h een zeer kleine waarde in te vullen (niet té klein, want dan geeft de deling onacceptabele afrondfouten).

De operatie "afgeleide" is een hogere-orde functie: er gaat een functie in en er komt een functie uit. De definitie in Haskell kan luiden:

```
diff :: (Float -> Float) -> (Float -> Float)
diff f = f'
  where f' x = (f (x + h) - f x) / h
        h    = 0.0001
```

Er zijn andere definities van diff mogelijk die een nauwkeurigere benadering geven, maar op deze manier lijkt de definitie van de benaderingsfunctie het meest op de wiskundige definitie.

Door het Currying-mechanisme kan het tweede paar haakjes in het type worden weggelaten, omdat -> naar rechts associeert.

```
diff :: (Float -> Float) -> Float -> Float
```

De functie diff kan dus ook beschouwd worden als functie met twee parameters: de functie waarvan de afgeleide genomen moet worden, en het punt waarin de afgeleide functie berekend moet worden. Vanuit dit gezichtspunt had de definitie kunnen luiden:

```
diff f x = (f (x + h) − f x) / h
  where h = 0.0001
```

De twee definities zijn volkomen equivalent. Voor de duidelijkheid van het programma verdient de tweede versie misschien de voorkeur omdat hij eenvoudiger is (het is niet nodig om de functie f′ een naam te geven en hem vervolgens te definiëren). Aan de andere kant benadrukt de eerste definitie dat diff beschouwd kan worden als een functietransformatie.

De functie diff leent zich goed voor partiële parametrisatie, zoals in de definitie:

```
afgeleide_van_sinus_sqrt = diff (sqrt . sin)
```

De waarde h is in beide definities van diff in een **where** clausule gezet. Daardoor is hij gemakkelijk te wijzigen, als het programma later nog eens veranderd zou moeten worden (dat kan natuurlijk ook in de expressie zelf, maar dan moet het twee keer gebeuren, met het gevaar dat je er een vergeet).

Nog flexibeler is het, om de waarde van h als parameter van diff te gebruiken:

```
flexDiff h f x = (f (x + h) − f x) / h
```

Door h als eerste parameter van flexDiff te definiëren, kan deze functie weer partieel geparametriseerd worden om verschillende versies van diff te maken:

```
grofDiff  =  flexDiff 0.01
fijnDiff  =  flexDiff 0.0001
superDiff = flexDiff 0.000001
```

3.4.3 Zelfgemaakte wortel

In Haskell is de functie sqrt ingebouwd om de vierkantswortel (*square root*) van een getal uit te rekenen. In deze paragraaf wordt een methode besproken hoe je zelf een wortelfunctie kunt maken, als deze niet ingebouwd zou zijn. Het demonstreert een techniek die veel gebruikt wordt bij het rekenen met *Float* getallen. De functie wordt in paragraaf 3.4.5 gegeneraliseerd naar inverses van andere functies dan de sqrt-functie. Daar p. 75 wordt ook verklaard waarom de hier geschetste methode werkt.

Voor de vierkantswortel van een getal x geldt de volgende eigenschap:

als y een goede benadering is voor $\sqrt{x}$
dan is $\frac{1}{2}(y + \frac{x}{y})$ een betere benadering.

Deze eigenschap kan gebruikt worden om de wortel van een getal x uit te rekenen: neem 1 als eerste benadering, en bereken net zolang betere benaderingen, totdat het resultaat goed genoeg is. De waarde y is goed genoeg als benadering voor $\sqrt{x}$ als y^2 niet te veel meer afwijkt van x.

Voor de waarde van $\sqrt{3}$ zijn de benaderingen y_0, y_1, enz. als volgt:

$$
\begin{array}{lcll}
y_0 & = & & = 1 \\
y_1 & = & 0.5 * (y_0 + 3/y_0) & = 2 \\
y_2 & = & 0.5 * (y_1 + 3/y_1) & = 1.75 \\
y_3 & = & 0.5 * (y_2 + 3/y_2) & = 1.732142857 \\
y_4 & = & 0.5 * (y_3 + 3/y_3) & = 1.732050810 \\
y_5 & = & 0.5 * (y_4 + 3/y_4) & = 1.732050807
\end{array}
$$

Het kwadraat van deze laatste benadering wijkt nog maar 10^{-18} af van 3.

Voor het proces "een startwaarde verbeteren totdat het goed genoeg is" kan de functie until uit paragraaf 3.3.2 gebruikt worden: p. 65

```
root x = until goodEnough improve 1.0
  where improve      y = 0.5 * (y + x / y)
        goodEnough y = y * y ~= x
```

De operator ~= is de "ongeveer gelijk aan" operator, die als volgt gedefinieerd kan worden:

```
infix 5 ~=
a ~= b = a - b < h && b - a < h
       where h = 0.000001
```

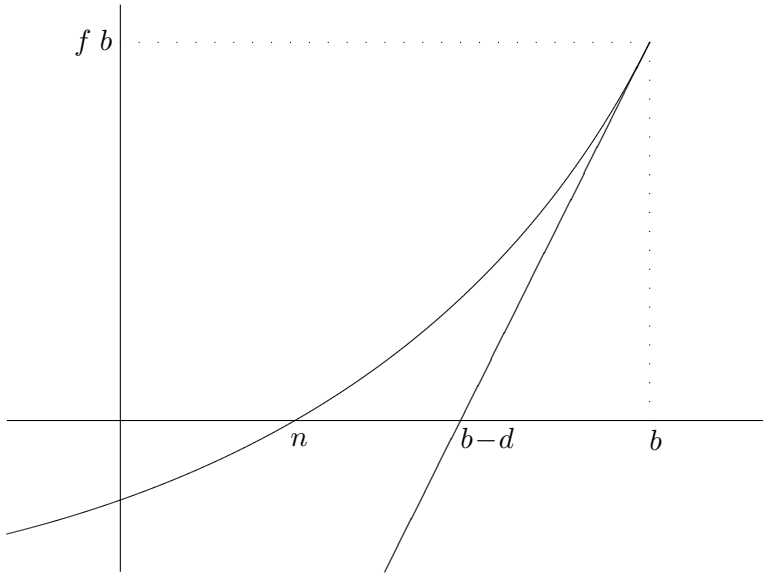
De hogere-orde functie until werkt op de *functies* improve en goodEnough en op de startwaarde 1.0. Hoewel improve naast 1.0 staat, wordt de functie improve dus niet onmiddellijk op 1.0 toegepast; in plaats daarvan worden beiden aan until meegegeven. Door het Currying-mechanisme is het immers alsof de haakjes geplaatst stonden als (((until goodEnough) improve) 1.0). Pas bij de uitwerking van until blijkt dat improve alsnog op 1.0 wordt toegepast.

Iets anders dat opvalt aan de definitie van improve en goodEnough is dat deze functies, behalve van hun parameter y, ook gebruik kunnen maken van x. Voor deze functies is het dus alsof x een constante is. (Vergelijk de definities van de "constanten" d en n in de definitie van abcFormule′ in paragraaf 2.4.1). p. 38

3.4.4 Nulpunt van een functie

Een ander numeriek probleem dat opgelost kan worden met iteratie door middel van until is het bepalen van het nulpunt van een functie.

Beschouw een functie f waarvan het nulpunt n gezocht wordt. Stel dat b een benadering is voor het nulpunt. Dan is het snijpunt van de raaklijn aan f in b met de x-as een betere benadering voor het nulpunt (zie figuur).



Het gezochte snijpunt ligt op afstand d van de eerste benadering b. De waarde van d kan als volgt bepaald worden. De richtingscoëfficiënt van de raaklijn aan f in b is gelijk aan $f'(b)$. Anderzijds is deze richtingscoëfficiënt gelijk aan $f(b)/d$. Dus $d = f(b)/f'(b)$.

Hiermee is een improve-functie gevonden: als b een benadering is voor het nulpunt van f, dan is $b - f(b)/f'(b)$ een betere benadering. Dit staat bekend als de "methode van Newton". (De methode werkt niet altijd; bijvoorbeeld voor functies met lokale extremen: je kunt dan "heen en weer blijven slingeren". Daar gaan we hier niet verder op in.)

Net als bij de functie root kan Newton's improve-functie gebruikt worden als parameter van until. Als "goed genoeg"-functie kan ditmaal gecontroleerd worden of de f-waarde in het benaderde nulpunt al klein genoeg is.

```
zeroPoint f y0 = until goodEnough improve y0
  where improve b      = b − f b / diff f b
        goodEnough b = f b =~ 0.0
```

De eerste benadering die gebruikt kan worden, is als extra parameter aan de functie meegegeven. De differentieerfunctie uit paragraaf 3.4.2 komt ook goed van pas. p. 72

3.4.5 De inverse van een functie

Het nulpunt van de functie f waarbij f x = x ^ 2 − a is $\sqrt{a}$. Dus om de wortel van a te vinden is het voldoende om het nulpunt van f te bepalen. Nu dat we beschikken over de functie zeroPoint kunnen we root ook als volgt opschrijven:

```
root a = zeroPoint f 1.0
  where f x = x ∗ x − a
```

Hetzelfde kunnen we doen voor de derdemachtswortel:

```
cubeRoot a = zeroPoint f 1.0
  where f x = x * x * x - a
```

In feite kunnen we de inverse van elke functie op deze manier bepalen, bijvoorbeeld:

```
arcsin  a = zeroPoint f 0.0
  where f x = sin  x - a
arccos a = zeroPoint f 0.0
  where f x = cos x - a
```

Het bestaan van zo'n manier is vaak een teken dat we onze oplossing kunnen generaliseren door een hogere-orde functie te introduceren (zoals we eerder in paragraph 3.3.1 zagen, alwaar foldr de generalisatie was van sum, product en and). Hier is de gezochte hogere-orde functie de functie inverse, welke een extra parameter g nodig heeft – de functie waarvan we de inverse zoeken. p. 64

```
inverse g a = zeroPoint f 0.0
  where f x = g x - a
```

Als je het algemene patroon hebt gesnapt is het begrijpen van de hogere-orde definitie niet moeilijker dan de meer specifieke instanties. De speciale instanties krijgen we terug door partiële parameterisatie::

```
arcsin  = inverse sin
arccos = inverse cos
ln      = inverse exp
```

Zoals alle functies met verscheidene parameters kan inverse zowel gebruikt worden als een functie met twee parameters en als een functie met één parameter welke een functie oplevert, want het type van inverse is

```
inverse :: (Float -> Float) -> Float -> Float
```

wat hetzelfde is als

```
inverse :: (Float -> Float) -> (Float -> Float)
```

omdat -> rechtsassociërend is.

Merk op dat we wortelfunctie paragraph 3.4.3 ook de methode van Newton gebruikt. Dit wordt duidelijk als we de eerdere definitie herschrijven naar root p. 73

```
root a = zeroPoint f 1.0
  where f x = x * x - a
```

door zeroPoint in de expressie zeroPoint f 1.0 te vervangen door diens definitie. We krijgen

```
root a = until goodEnough improve 1.0
  where improve      b = b − f b / diff f b
        goodEnough   b = f b =~ 0.0
        f            x = x ∗ x − a
```

In dit speciale geval hoeft diff f niet numeriek te worden benadered, omdat de afgeleide van f de functie $(2*)$ is. Daarom kan de formule voor improve b als volgt worden vereenvoudigd:

$$\begin{aligned} & b - \frac{f\ b}{f'\ b} \\ = \ & b - \frac{b^2 - a}{2b} \\ = \ & b - \frac{b^2}{2b} + \frac{a}{2b} \\ = \ & \frac{b}{2} + \frac{a/b}{2} \\ = \ & 0.5 * (b + a/b) \end{aligned}$$

Dit is precies de improve-functie welke in paragraph 3.4.3 werd gebruikt. p. 73

Opgaven

3.1 Explain the placement of the parentheses in the expression (f (x + h) − f x) / h.

3.2 Which parentheses are superfluous in the following expressions?

- (plus 3) (plus 4 5)
- sqrt (3.0) + (sqrt 4.0)
- (+) (3) (4)
- (2 ∗ 3) + (4 ∗ 5)
- (2 + 3) ∗ (4 + 5)
- (*a b*) (*c d*)
- (*a* -> *b*) -> (*c* -> *d*)

3.3 Why is it common in mathematics to associate operators to the right?

3.4 Is the operator after (.) associative?

3.5 In the programming language Pascal, && has the same priority as ∗, and || has the same priority as +. Why is that not clever?

3.6 Give examples for functions with the following types:

- (*Float* -> *Float*) -> *Float*
- *Float* -> (*Float* -> *Float*)
- (*Float* -> *Float*) -> (*Float* -> *Float*)

3.7 In paragraph 3.3.3 it says that after can also be regarded as a function with one parameter. How is that visible from its type? Give a definition of after of the form after y = ... p. 66

3.8 Function composition first applies the latter of the supplied functions to the argument, the former thereafter. Write a function before that can be used to rewrite f . g . h to h `before` g `before` f. What can you say about associativity of . and `before`? Remark: it would be nice to use ; as an operator instead of before. Unfortunately, ; is a reserved character in Haskell.

3.9 Write a recursive function that for a fixed interest determines the number of years it takes until the account balance surpasses a certain limit.

3.10 Give a definition of cubeRoot that does not rely on numeric differentiation (also not indirectly via zeroPoint).

3.11 Define the inverse function from paragraph 3.4.5 using lambda notation. p. 75

3.12 Why can the function root and cubeRoot be written without using the diff function, while it is not possible to write inverse this way in general?

3.13 Write a function integral, that computes the integral of a function f over a given interval. It should divide the interval into a number of small segments and approximate f on each of the segments by a linear function. Choose a parameter order for integral that is most convenient for partial parametrisation.

3.14 We have seen that [...] is a type function that maps types to types. Similarly because -> is a type constructor mapping two types to a type, for some c also c -> is a type function mapping a type a to c -> a. Rewrite the type of map by substituting the type function [...] by c ->. Can you derive an implementation from the resulting type?

4 Lijsten

4.1 Lijsten

4.1.1 Opbouw van een lijst

Lijsten worden gebruikt om een aantal elementen te groeperen. Die elementen moeten van *hetzelfde type* zijn. Voor elk type is er een type "lijst-van-dat-type". Er bestaan dus bijvoorbeeld lijsten-van-integers, lijsten-van-floats, en lijsten-van-functies-van-int-naar-int. Maar ook een aantal lijsten van hetzelfde type kunnen weer in een lijst worden opgenomen; zo ontstaan lijsten-van-lijsten-van-integers, lijsten-van-lijsten-van-lijsten-van-booleans, enzovoort.

Het type van een lijst wordt aangegeven door het type van de elementen tussen vierkante haken. De hierboven genoemde types kunnen dus korter worden aangegeven door $[Int]$, $[Float]$, $[Int \texttt{ -> } Float]$, $[[Int]]$ en $[[[Bool]]]$.

Er zijn verschillende manieren om een lijst te maken: opsomming, opbouw met :, en numerieke intervallen.

Opsomming

Opsomming van de elementen is vaak de eenvoudigste manier om een lijst te maken. De elementen moeten van hetzelfde type zijn. Enkele voorbeelden van lijstopsommingen met hun type zijn:

$$
\begin{array}{ll}
[1,2,3] & :: [Int] \\
[1,3,7,2,8] & :: [Int] \\
[True, False, True] & :: [Bool] \\
[\mathsf{sin}, \mathsf{cos}, \mathsf{tan}] & :: [Float \texttt{ -> } Float] \\
[[1,2,3],[1,2]] & :: [[Int]]
\end{array}
$$

Het maakt voor het type van de lijst niet uit hoeveel elementen er zijn. Een lijst met drie integers en een lijst met twee integers hebben allebei het type $[Int]$. Daarom mogen de lijsten $[1,2,3]$ en $[1,2]$ in het vijfde voorbeeld op hun beurt elementen zijn van één lijst-van-lijsten.

De elementen van de lijst hoeven geen constanten te zijn; ze mogen bepaald worden door een berekening:

$[1 + 2, 3 * \mathsf{x}, \mathsf{length}\ [1, 2]] :: [Int]$
$[3 < 4, a == 5, \mathsf{p}\ \&\&\ \mathsf{q}] :: [Bool]$
$[\mathsf{diff}\ \sin, \mathsf{inverse}\ \cos] :: [Float \text{ -> } Float]$

De gebruikte functies en andere identifiers moeten dan wel een passend type hebben.

Het aantal elementen van een lijst is vrij. Een lijst kan dus ook bestaan uit maar één element:

$[True] :: [Bool]$
$[[1, 2, 3]] :: [[Int]]$

Een lijst met één element wordt ook wel een *singletonlijst* genoemd. De lijst $[[1, 2, 3]]$ is ook een singletonlijst: het is immers een lijst van lijsten, die één element (de lijst $[1, 2, 3]$) heeft.

Let op het verschil tussen een *expressie* en een *type*. Als er tussen de vierkante haken een type staat, is er sprake van een type (bijvoorbeeld $[Bool]$ of $[[Int]]$). Als er tussen de vierkante haken een expressie staat, is het geheel ook een expressie (een singletonlijst, bijvoorbeeld $[True]$ of $[3]$).

Het aantal elementen van een lijst kan ook nul zijn. Een lijst met nul elementen heet de *lege lijst*. De lege lijst heeft een polymorf type: het is een "lijst-van-maakt-niet-uit-wat". Op plaatsen in een polymorf type waar een willekeurig type ingevuld mag worden, staan typevariabelen (zie paragraaf 2.5.3), dus het type van de lege lijst is $[a]$: p. 50

$[] :: [a]$

De lege lijst mag op elke plaats in een expressie gebruikt worden waar een lijst nodig is. Het type wordt daarbij door de context bepaald:

$\mathsf{sum}\ []$	[] is een lege lijst getallen
$\mathsf{and}\ []$	[] is een lege lijst Booleans
$[[], [1, 2], [3]]$	[] is een lege lijst getallen
$[[1 < 2, True], []]$	[] is een lege lijst Booleans
$[[[1]], []]$	[] is een lege lijst lijsten-van-getallen
$\mathsf{length}\ []$	[] is een lege lijst (doet er niet toe waarvan)

Opbouw met :

Een andere manier om een lijst te maken is het gebruik van de :-operator. Deze operator zet een element op kop van een lijst, en maakt zo een langere lijst.

$(:) :: a \text{ -> } [a] \text{ -> } [a]$

Als bijvoorbeeld xs de lijst $[3, 4, 5]$ is, dan is $1 : \mathsf{xs}$ de lijst $[1, 3, 4, 5]$. Gebruik makend van de lege lijst en de :-operator is elke lijst te construeren. Zo is bijvoorbeeld $1 : (2 : (3 : []))$ de lijst $[1, 2, 3]$. De :-operator associeert naar rechts, dus je kunt kortweg $1 : 2 : 3 : []$ schrijven.

In feite is deze manier van opbouw de enige "echte" manier om een lijst te maken. Een opsomming van een lijst is vaak overzichtelijker in programma's, maar heeft precies dezelfde betekenis als de overeenkomstige expressie met :-operatoren. Daarom kost een opsomming ook tijd. In `ghci`:

```
? [1,2,3]
[1, 2, 3]
(7 reductions, 29 cells)
```

Elke aanroep van : (die je niet ziet, maar die er dus wel staat) kost 2 reducties.

Numerieke intervallen

Een derde manier om een lijst te maken is de intervalnotatie: twee numerieke expressies met twee punten ertussen en vierkante haken eromheen:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

(Hoewel de punt gebruikt mag worden als symbool in operatoren, is .. geen operator. Het is namelijk één van symbolencombinaties die in paragraaf 2.3.2 werden gereserveerd p. 32 voor speciaal gebruik.)

De waarde van de expressie [x..y] wordt berekend door enumFromTo x y aan te roepen. De functie enumFromTo is als volgt gedefinieerd:

$$\begin{aligned}\mathsf{enumFromTo\ x\ y} \mid \mathsf{y} < \mathsf{x} &= [\,] \\ \mid \mathsf{otherwise} &= \mathsf{x} : \mathsf{enumFromTo}\ (\mathsf{x}+1)\ \mathsf{y}\end{aligned}$$

Als y kleiner is dan x is de lijst dus leeg; anders is x het eerste element, is het volgende element één groter (tenzij y is gepasseerd), enzovoort.

De notatie voor numerieke intervallen is wat we noemen *syntactische suiker*, die het gebruik van de taal iets makkelijker maakt. Strikt genomen kun je in plaats van de notatie voor numerieke intervallen altijd enumFromTo gebruiken, maar door de notatie worden programma's wel leesbaarder.

4.1.2 Functies op lijsten

Functies op lijsten worden vaak gedefinieerd door gebruik te maken van *patronen*: de functie wordt apart gedefinieerd voor de lege lijst, en voor een lijst die de vorm x : xs heeft. Elke lijst is immers òf leeg, òf heeft een eerste element x, dat op kop staat van een (mogelijk lege) lijst xs.

Een aantal definities van functies op lijsten zijn al ter sprake gekomen: head en tail in paragraaf 2.4.3, som en length in paragraaf 2.4.4, en map, filter en foldr in paragraaf 3.3.1. p. 41 Hoewel dit allemaal standaardfuncties zijn die in de prelude worden gedefinieerd, en ze p. 42 dus niet zelf gedefinieerd hoeven te worden, is het toch belangrijk om hun definitie te p. 64 bekijken. Ten eerste omdat het goede voorbeelden zijn van definities van functies op lijsten, ten tweede omdat de definitie vaak de duidelijkste beschrijving geeft van wat een standaardfunctie doet.

Hieronder volgen nog meer definities van functies op lijsten. Veel van deze functies zijn recursief: ze roepen zichzelf aan op een (kleinere) lijst dan ze als argument meekregen.

Lijsten samenvoegen

Twee lijsten van hetzelfde type kunnen worden samengevoegd tot één lijst met de operator ++. Deze operatie wordt ook wel *concatenatie* ("samenketening") genoemd. Bijvoorbeeld: $[1,2,3]$ ++ $[4,5]$ geeft de lijst $[1,2,3,4,5]$. Concatenatie met de lege lijst (zowel aan de voorkant als aan de achterkant) laat een lijst onveranderd: $[1,2]$ ++ $[\,]$ geeft $[1,2]$.

De operator ++ is een standaardfunctie, maar hij kan gewoon in Haskell gedefinieerd worden (dat gebeurt ook in de prelude). Het is dus geen "ingebouwde" operator zoals de :-operator. De definitie luidt:

```
(++)            :: [a] -> [a] -> [a]
[] ++       ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

In de definitie wordt de linker parameter onderworpen aan patroonanalyse. In het niet-lege geval wordt de operator recursief aangeroepen met de kortere lijst xs als parameter.

Er is nog een functie die lijsten samenvoegt. Deze functie, concat, werkt op een *lijst* van lijsten. Alle lijsten in de lijst van lijsten worden samengevoegd tot één lange lijst. Dus

```
? concat [ [1,2,3], [4,5], [], [6] ]
[1, 2, 3, 4, 5, 6]
```

De definitie van concat is als volgt:

```
concat           :: [[a]] -> [a]
concat []        = []
concat (xs : xss) = xs ++ concat xss
```

Het eerste patroon, [] is de lege lijst; een lege lijst lijsten wel te verstaan. Het resultaat is dan een lege lijst: een lijst zonder elementen. In het tweede geval van de definitie is de lijst lijsten niet leeg. Er staan dus één lijst, xs, op kop, en er is een rest-lijst-van-lijsten

xss. Eerst worden alle lijsten in de rest samengevoegd door de recursieve aanroep van concat; tenslotte wordt de eerste lijst xs daar ook nog voor gezet.

Let op het verschil tussen ++ en concat: de operator ++ werkt op *twee* lijsten, de functie concat werkt op *een lijst* van lijsten. Beide worden "concatenatie" genoemd. (Vergelijk de situatie met de operator &&, die kijkt of twee Booleans *True*, en de functie and die kijkt of een hele lijst van Booleans allemaal *True* zijn).

Delen van een lijst selecteren

In de prelude worden enkele functies gedefinieerd die delen van een lijst selecteren. Bij sommige functies is het resultaat een (kortere) lijst, bij andere is het één element.

Aangezien een lijst wordt opgebouwd uit een kop en een staart, is het eenvoudig om de kop en staart van een lijst weer terug te krijgen:

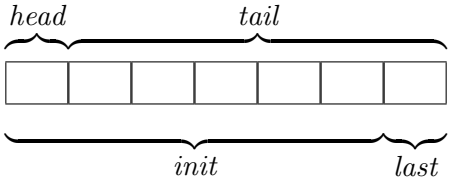
```
head       :: [a] -> a
head (x:xs) = x
tail       :: [a] -> [a]
tail (x:xs) = xs
```

Deze functies doen patroonanalyse op de parameter, maar er is geen aparte definitie voor het patroon []. Als deze functies worden aangeroepen op een lege lijst volgt er dan ook een run-time foutmelding.

Minder eenvoudig is het om een functie te schrijven die het *laatste* element uit een lijst selecteert. Daarvoor is recursie nodig:

```
last       :: [a] -> a
last (x:[]) = x
last (x:xs) = last xs
```

Ook deze functie is niet gedefinieerd voor de lege lijst, omdat die met geen van de twee patronen overeenkomt. Zoals er bij head een functie tail hoort, zo hoort er bij last een functie init. Een schematisch overzicht van deze vier functies:

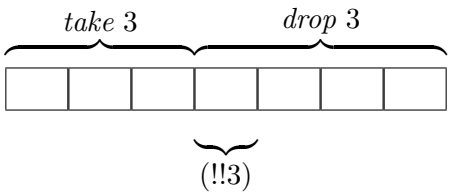


De functie init selecteert alles *behalve* het laatste element. Ook daarvoor is weer recursie nodig:

```
init       :: [a] -> [a]
init (x:[]) = []
init (x:xs) = x : init xs
```

Het patroon x : [ ] kan worden geschreven als [x] (en dat gebeurt dan ook meestal).

In paragraaf 3.4.1 is een functie take ter sprake gekomen. Behalve een lijst heeft take een integer als parameter, die aangeeft hoeveel elementen van de lijst in het resultaat zitten. De tegenhanger van take is drop, die juist een bepaald aantal elementen van het begin van de lijst verwijdert. Tenslotte is er een operator !!, die één gespecificeerd element uit de lijst selecteert. Schematisch: p. 71



Deze functies zijn als volgt gedefinieerd:

```
take, drop      :: Int -> [a] -> [a]
take 0 xs       = []
take n []       = []
take n (x : xs) = x : take (n − 1) xs
drop 0 xs       = xs
drop n []       = []
drop n (x : xs) = drop (n − 1) xs
```

Als de lijst te kort is, dan worden zo veel mogelijk elementen genomen, respectievelijk weggelaten. Dat komt door de tweede regel in de definities: die zegt dat als je een lege lijst in de functies stopt, het resultaat altijd de lege lijst is, wat het gespecificeerde aantal ook is. Als deze regel niet in de definitie had gestaan, dan waren take en drop ongedefinieerd voor te korte lijsten.

De operator !! selecteert één element uit een lijst. De kop van de lijst telt daarbij als "nulde" element, dus xs !! 3 geeft het *vierde* element van de lijst xs. Deze operator mag niet op te korte lijsten worden toegepast; er valt in dat geval immers geen zinvolle waarde op te leveren. De definitie luidt:

```
infixl 9 !!
(!!)          :: [a] -> Int -> a
(x : xs) !! 0 = x
(x : xs) !! n = xs !! (n − 1)
```

Deze operator kost, vooral voor grote getallen, de nodige tijd: de hele lijst wordt er voor vanaf het begin doorlopen. Hij moet dus enigszins spaarzaam worden toegepast. De operator is geschikt om één element uit een lijst te selecteren. De functie weekdag uit paragraaf 3.4.1 had bijvoorbeeld zo gedefinieerd kunnen worden: p. 70

```
weekdag d = ["zondag", "maandag", "dinsdag", "woensdag",
             "donderdag", "vrijdag", "zaterdag"] !! d
```

Moeten echter alle elementen van een lijst achtereenvolgens geselecteerd worden, dan is het beter om map of foldr te gebruiken.

Lijsten omdraaien

De functie reverse in de prelude zet de elementen van een lijst in omgekeerde volgorde. De functie kan eenvoudig recursief worden gedefinieerd. Een omgekeerde lege lijst blijft een lege lijst. Voor een niet-lege lijst moet het staartstuk omgekeerd worden, en het eerste element helemaal aan het eind daarvan geplaatst worden. De definitie kan dus als volgt luiden:

```
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

Alhoewel deze functie correct is, geven we toch de voorkeur aan een iets andere definitie. Wanneer we even nadenken over de efficiëntie van bovenstaande definitie, dan zien we dat de elementen die we van de eerste lijst afhalen aan het eind van een steeds langere lijst gehangen worden. De hoeveelheid werk loopt dus kwadratisch op met de lengte van de om te keren lijst; iets wat we liever niet zien. We kunnen dit, en veel soortgelijke problemen, oplossen door gebruik te maken van een hulpfunctie met een extra parameter (de *accumulator*), die we gebruiken om het uitendelijke resultaat in op te bouwen:

```
reverse x = reverseaccum x []
  where reverseaccum (x : xs) result = reverseaccum xs (x : result)
        reverseaccum []       result = result
```

Eigenschappen van lijsten

Een belangrijke eigenschap van een lijst is zijn lengte. De lengte kan berekend worden met de functie length. Deze functie is in de prelude als volgt gedefinieerd:

```
length ::          [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

In de prelude zit verder een functie elem die test of een bepaald element in een lijst aanwezig is. De functie elem kan als volgt worden gedefinieerd:

```
elem e xs = or (map (== e) xs)
```

De functie vergelijkt alle elementen van xs met e (partiële parametrisering van de operator ==). Dat levert een lijst Booleans op, waarvan or controleert of er minstens één *True* is. De functie kan, met gebruik van de functiecompositieoperator, ook zo geschreven worden:

```
elem e = or . (map (== e))
```

De functie notElem controleert of een element juist níet in een lijst zit:

```
notElem e xs = not (elem e xs)
```

Deze functie kan ook gedefinieerd worden met

```
notElem e = and . (map (/= e))
```

4.1.3 Hogere-orde functies op lijsten

Functies kunnen flexibeler gemaakt worden door ze een functie als parameter mee te geven. Veel standaardfuncties op lijsten hebben een functie als parameter. Het zijn daardoor hogere-orde functies.

map, filter en foldr

Eerder werden al de functies map, filter en foldr besproken. Deze functies doen, afhankelijk van hun functieparameter, iets met alle elementen van een lijst. De functie map past zijn functieparameter toe op alle elementen van de lijst:

```
                xs  =  [ 1 , 2 , 3 , 4 , 5 ]
                       ↓   ↓   ↓   ↓   ↓
map kwadraat xs  =  [ 1 , 4 , 9 , 16 , 25 ]
```

De functie filter gooit de elementen uit een lijst die niet aan een bepaalde Boolean functie voldoen:

```
                xs  =  [ 1 , 2 , 3 , 4 , 5 ]
                       ×   ↓   ×   ↓   ×
filter even xs  =  [       2 ,       4       ]
```

De functie foldr zet een operator tussen alle elementen van een lijst, te beginnen aan de rechterkant met een gespecificeerde waarde:

```
                xs  =  [ 1 , 2 , 3 , 4 , 5 ]
                       ↓   ↓   ↓   ↓   ↓
foldr (+) 0 xs  =    (1 + (2 + (3 + (4 + (5 + 0 )))))
```

Deze drie standaardfuncties worden in de prelude recursief gedefinieerd. Ze werden eerder besproken in paragraaf 3.3.1. p. 64

```
map                 :: (a -> b) -> [a] -> [b]
map f []            = []
map f (x : xs)      = f x : map f xs
filter              :: (a -> Bool) -> [a] -> [a]
filter p []         = []
filter p (x : xs)
   | p x            = x : filter p xs
   | otherwise =      filter p xs
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr op e []       = e
foldr op e (x : xs) = x `op` foldr op e xs
```

Door veel van deze standaardfuncties gebruik te maken, kan de recursie in andere functies verborgen worden. Het "vuile werk" wordt dan door de standaardfuncties opgeknapt, en de andere functies zien er overzichtelijker uit. De functie or, die kijkt of in een lijst Booleans minstens één waarde *True* is, is bijvoorbeeld zo gedefinieerd:

```
or = foldr (||) False
```

Maar het is ook mogelijk om deze functie direct met recursie te definiëren, zonder gebruik te maken van foldr:

```
or []       = False
or (x : xs) = x || or xs
```

Veel functies kunnen geschreven worden als combinatie van een aanroep van foldr en een aanroep van map. De functie elem uit de vorige paragraaf is daar een voorbeeld van:

```
elem e = or . map (== e)
```

Maar ook deze functie kan natuurlijk direct, zonder gebruik te maken van standaardfuncties, gedefinieerd worden. Recursie is dan weer noodzakelijk:

```
elem e []       = False
elem e (x : xs) = x == e || elem e xs
```

takeWhile en dropWhile

Een variant op de functie filter is de functie takeWhile. Deze functie heeft, net als filter, een eigenschap (functie met Boolean resultaat) en een lijst als parameter. Het verschil is dat filter altijd alle elementen van de lijst bekijkt. De functie takeWhile begint aan het begin van de lijst, en stopt met zoeken zodra er één element niet meer aan de eigenschap voldoet. Bijvoorbeeld: takeWhile even $[2, 4, 6, 7, 8, 9]$ geeft $[2, 4, 6]$. Anders dan bij filter komt de 8 niet in het resultaat, want de 7 doet takeWhile stoppen met zoeken. De definitie in de prelude luidt:

```
takeWhile      :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
   | p x       = x : takeWhile p xs
   | otherwise = []
```

Vergelijk deze definitie met die van filter.

Zoals er bij take een functie drop hoort, zo hoort er bij takeWhile een functie dropWhile. Deze laat het beginstuk van een lijst vervallen dat aan een eigenschap voldoet. Bijvoorbeeld: dropWhile even $[2, 4, 6, 7, 8, 9]$ is $[7, 8, 9]$. De definitie luidt:

```
dropWhile      :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x : xs)
   | p x       = dropWhile p xs
   | otherwise = x : xs
```

foldl

De functie foldr zet een operator tussen alle elementen van een lijst, en begint daarbij aan de rechterkant van de lijst. De functie foldl doet hetzelfde, maar begint aan de linkerkant. Net als foldr heeft foldl een extra parameter die aangeeft wat het resultaat is voor de lege lijst.

Een voorbeeld van de werking van foldl op een lijst met vijf elementen is het volgende:

```
          xs  =          [ 1  ,  2  ,  3  ,  4  ,  5  ]
                           ↓     ↓     ↓     ↓     ↓
foldl (+) 0 xs  =  (((((0  +  1) + 2) + 3) + 4) + 5)
```

Om een definitie van deze functie te kunnen geven, is het handig om eerst twee voorbeelden onder elkaar te zetten:

```
foldl (⊕) a [x, y, z] = ((a ⊕ x) ⊕ y) ⊕ z
foldl (⊕) b [   y, z] = (  b     ⊕ y) ⊕ z
```

Hieruit blijkt dat aanroep van foldl op de lijst x:xs (met xs=$[y, z]$ in het voorbeeld) gelijk is aan foldl xs, mits in de recursieve aanroep als startwaarde in plaats van a de waarde $a \oplus$ x genomen wordt. Met deze observatie kan de definitie geschreven worden:

```
foldl op e []       = e
foldl op e (x : xs) = foldl op (e `op` x) xs
```

We maken hier dus weer gebruik van een *accumulerende parameter*.

Voor associatieve operatoren zoals + maakt het niet zo veel uit of je foldr of foldl gebruikt. Echter de functie foldl is wat we noemen *tail-recursief*, d.w.z dat het resultaat gevormd

wordt door een recursieve aanroep van de functie zelf (of in andere woorden: het laatste dat de functie doet is zichzelf aanroepen). Bijvoorbeeld, van de functies take en drop is de eerste niet tail-recursief, en de tweede wel. Een goede compiler herkent deze stituatie, en genereert hier efficiënte code voor. In het algemeen heeft dus het gebruik van foldl de voorkeur boven foldr. Voor niet-associatieve operatoren zoals − is het resultaat van foldl natuurlijk anders dan dat van foldr, en hebben we dus niet de vrije keuze.

4.1.4 Lijsten vergelijken en ordenen

Lijsten van integers vergelijken

Twee lijsten zijn gelijk als ze precies dezelfde elementen hebben, die in dezelfde volgorde staan. Dit is een definitie van de functie eq waarmee de gelijkheid van lijsten van integers getest kan worden:

```
eqListInt ::                 [Int] -> [Int] -> Bool
eqListInt [] []              = True
eqListInt [] (y : ys)        = False
eqListInt (x : xs) []        = False
eqListInt (x : xs) (y : ys) = x == y && eqListInt xs ys
```

In deze definitie kan zowel de eerste als de tweede parameter leeg of niet-leeg zijn; voor alle vier de combinaties is er een definitie. In het vierde geval worden de overeenkomstige elementen met elkaar vergeleken (x == y), en wordt de operator recursief aangeroepen op de restlijsten (eq xs ys).

Lijsten van willekeurige types vergelijken

Behalve lijsten van integers wil je natuurlijk ook wel eens lijsten vergelijken met elementen van een ander type. Je wilt wellicht lijsten vergelijken waarvan de elementen booleans zijn, of strings. In dat geval kun je niet meer de operator == gebruiken om de elementen te vergelijken, want die kan alleen gebruikt worden om integers te vergelijken. Maar hoe moet je de elementen dan vergelijken?

We kunnen de functie waarmee de elementen worden vergeleken *als extra parameter* meegeven aan de functie. De vergelijkingsfunctie is zelf ook een functie maar dat is geen enkel probleem, zoals dat ook bij bijvoorbeeld map het geval was. De definitie wordt dan als volgt:

```
eqList :: (a -> a -> Bool) -> [a] -> [a] -> Bool
eqList test [] []              = True
eqList test [] (y : ys)        = False
eqList test (x : xs) []        = False
eqList test (x : xs) (y : ys) = test x y && eqList test xs ys
```

zodat

```
? eqList (==) [1,2,3] [1,2,3]
True
? eqList eqBool [True,False] [False,True]
False
```

Door eqList partiëel te parametriseren kunnen we een vergelijkingsfunctie voor lijsten maken, die we vervolgens weer kunnen meegeven aan eqList. Op die manier kun je bijvoorbeeld lijsten van lijsten van integers vergelijken:

```
? eqList (eqList (==)) [[1,2],[3,4]] [[1,2],[4,3]]
False
```

In volledig Haskell kunnen we gewoon == gebruiken tussen lijsten. Bij de behandeling van het klassesysteem zullen we zien hoe dit geregeld is.

Lijsten ordenen

Als de elementen van een lijst geordend kunnen worden met $<$, $\leqslant$ enz., dan kunnen ook lijsten geordend worden. Dit gebeurt volgens de *lexicografische ordening* ("woordenboekvolgorde"): het eerste element van de lijsten is bepalend, tenzij het eerste element van beide lijsten gelijk is; in dat geval beslist het tweede element, tenzij dat ook gelijk is, enzovoort. Er geldt dus bijvoorbeeld $[2,3]<[3,1]$ en $[2,1]<[2,2]$. Als een van de twee lijsten een beginstuk is van de ander, dan is de kortste het "kleinste", bijvoorbeeld $[2,3]<[2,3,4]$. Dat in deze beschrijving het woord "enzovoort" nodig is, is een aanwijzing dat er recursie nodig is in de definitie.

Omdat operatoren zoals $<$ al zijn gereserveerd voor het vergelijken van integers, moeten we voor het vergelijken van lijsten een andere naam verzinnen. Of liever gezegd vier namen: voor $<$, $\leq$, $>$ en $\geq$. Net als in het geval van eqList zouden die functies dan een functie als parameter moeten krijgen voor het vergelijken van de elementen én nog een voor het ordenen van de elementen. Dat kan, maar het wordt een beetje ingewikkeld, en daarom is in de prelude voor een andere aanpak gekozen.

Om te beginnen is er een type *Ordering* beschikbaar met drie mogelijke waarden. Zoals een expressie van type *Bool* de waarden *True* en *False* kan hebben, zo kan een expressie van type *Ordering* de waarden *LT* ("less than"), *EQ* ("equal") of *GT* ("greater than"). Voor de elementaire typen zijn er functies die de onderlinge ligging van twee waarden kunnen aangeven, zoals de functie ordInt die als volgt kan worden gedefinieerd:

```
ordInt :: Int -> Int -> Ordering
ordInt x y
   | x < y    = LT
   | x == y = EQ
   | otherwise = GT
```

Die functie zou in een expressie gebruikt kunnen worden:

```
? ordInt 3 4
LT
```

maar hij is vooral bedoeld om als parameter mee te geven aan de functie ordList. Die kan als volgt worden gedefinieerd:

```
ordList                        :: (a -> a -> Ordering) -> [a] -> [a] -> Ordering
ordList test []       (y : ys) = LT
ordList test []       []       = EQ
ordList test (x : xs) []       = GT
ordList test (x : xs) (y : ys) =
  case test x y of
    GT -> GT
    LT -> LT
    EQ -> ordList test xs ys
```

Met behulp van deze functie kun je nu de onderlinge ligging van twee lijsten bepalen, bijvoorbeeld:

```
? ordList ordInt [2,3] [2,3,4]
LT
```

Net als bij eqList kan deze functie partiëel geparametriseerd worden om hem vervolgens aan zichzelf mee te geven. Je kunt op die manier ook lijsten van lijsten van wat je maar wilt ordenen.

Lidmaatschapstest

Als we een functie hebben om elementen te vergelijken, dan kunnen we die gebruiken om een functie maken die bepaalt of een element ergens in een lijst voorkomt. Zo'n functie, genaamd elemBy, krijgt drie parameters: een vergelijkingstestfunctie, een element, en een lijst. De definitie luidt:

```
elemBy :: (a -> a -> Bool) -> a -> [a] -> Bool
elemBy test x []                    = False
elemBy test x (y : ys) | test x y   = True
                       | otherwise = elemBy test x ys
```

In plaats van met expliciete recursie had deze functie ook gedefinieerd kunnen worden door handig gebruik te maken van map en foldr:

```
elemBy test e xs = or (map (test e) xs)
```

Nog compacter wordt het als we deze functie schrijven met gebruikmaking van functiecompositie:

```
elemBy test e = or . map (test e)
```

4.1.5 Lijsten sorteren

Alle tot nu toe genoemde functies op lijsten zijn vrij eenvoudig: door middel van recursie wordt de lijst éénmaal doorlopen om het resultaat te bepalen.

Een functie die niet op deze manier geschreven kan worden, is het sorteren (in opklimmende volgorde zetten van de elementen) van een lijst. Daarvoor moeten de elementen immers helemaal door elkaar gegooid worden.

Toch is het, zeker met hulp van de standaardfuncties, niet moeilijk om een sorteerfunctie te schrijven. Er zijn verschillende mogelijkheden om het sorteerprobleem aan te pakken: er zijn verschillende *algoritmen* mogelijk. Twee algoritmen zullen hier worden besproken. In beide algoritmen is het noodzakelijk dat de elementen van de lijst geordend kunnen worden. Het is dus mogelijk om een lijst integers of een lijst van lijsten van integers te sorteren, maar niet een lijst van functies.

Willen we lijsten sorteren, dan moeten we wel weten hoe de elementen van de lijst onderling worden geordend. Dit komt tot uiting in de eerste parameter van:

```
sorteer :: (a -> a -> Ordering) -> [a] -> [a]
```

In hoofdstuk 10 zullen we zien hoe we met behulp van type classes dit zonder dat eerste argument kunnen doen, want het is natuurlijk niet zo mooi dat we voortdurend dit soort functies moeten meegeven, en niet gewoon (==) of (<) mogen gebruiken. p. 189

Sorteren door invoegen

Stel dat een gesorteerde lijst gegeven is. Dan kan een nieuw element op de juiste plaats in deze lijst worden ingevoegd met de functie insert waar we de ordeningsrelatie weer expliciet meegeven:

```
insert                :: (a -> a -> Ordering) -> a -> [a] -> [a]
insert cmp e []       = [e]
insert cmp e (x : xs) = case e `cmp` x of
                          LT -> e : x : xs
                          _  -> x : insert cmp e xs
```

Als de lijst leeg is, dan wordt het nieuwe element e het enige element. Als de lijst niet leeg is, en element x op kop heeft staan, dan hangt het ervan af of e kleiner is dan x. Zo ja, dan komt e helemaal op kop te staan; zo nee, dan komt x op kop te staan, en moet e elders in de lijst worden ingevoegd. Een voorbeeld van het gebruik van insert:

```
? insert ordInt 5 [2,4,6,8,10]
[2, 4, 5, 6, 8, 10]
```

Voor de werking van insert is het essentieel dat de argumentlijst gesorteerd is; het resultaat is dan ook gesorteerd.

De functie insert kan gebruikt worden om een nog niet gesorteerde lijst te sorteren. Stel dat $[a, b, c, d]$ gesorteerd moet worden. Je kunt dan een lege lijst nemen (die is gesorteerd) en daar het laatste element d in invoegen. Het resultaat is een gesorteerde lijst, waarin c ingevoegd kan worden. Het resultaat blijft gesorteerd, ook nadat b is ingevoegd. Tenslotte kan a op de juiste plaats worden ingevoegd, en het eindresultaat is een gesorteerde versie van $[a, b, c, d]$. De expressie die berekend wordt is:

$$a \text{ `insert` } (b \text{ `insert` } (c \text{ `insert` } (d \text{ `insert` } [])))$$

De structuur van deze berekening is precies die van foldr, met insert als operator en [] als startwaarde. Een mogelijk sorteeralgoritme luidt dus:

```
isort cmp = foldr (insert cmp) []
```

met de functies insert zoals hierboven gedefinieerd. Dit algoritme wordt *insertion sort* genoemd.

Sorteren door samenvoegen

Een ander sorteeralgoritme maakt gebruik van de mogelijkheid om twee gesorteerde lijsten samen te voegen tot één. Daartoe dient de functie merge:

```
merge :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
merge cmp [] ys          = ys
merge cmp xs []          = xs
merge cmp (x : xs) (y : ys) = case x `cmp` y of
                                LT -> x : merge cmp      xs (y : ys)
                                _  ->     y : merge cmp (x : xs)     ys
```

Als één van beide lijsten leeg is, dan is de andere lijst het resultaat. Als beide lijsten niet-leeg zijn, dan komt de kleinste van de twee kopelementen op kop van het resultaat, en worden de overblijvende elementen samengevoegd door een recursieve aanroep van merge.

Net als insert gaat merge ervan uit dat de parameters gesorteerd zijn. In dat geval zorgt hij er voor dat ook het resultaat een gesorteerde lijst is.

Ook op de functie merge kan een sorteeralgoritme (*mergesort*) worden gebaseerd. Dit algoritme maakt er gebruik van dat de lege lijst en lijsten met één element altijd gesorteerd zijn. Langere lijsten kunnen (ongeveer) in tweeën worden gesplitst. De helften kunnen worden gesorteerd door een recursieve aanroep van het sorteeralgoritme. De twee gesorteerde resultaten kunnen tenslotte worden samengevoegd door merge.

```
msort        ::      (a -> a -> Ordering) -> [a] -> [a]
msort cmp xs
   | lengte <= 1 = xs
   | otherwise   = merge cmp (msort cmp ys) (msort cmp zs)
  where ys = take half xs
        zs      = drop half xs
        half    = lengte `div` 2
        lengte =  length xs
```

4.2 Speciale lijsten

4.2.1 Strings

In een voorbeeld in paragraaf 3.4.1 werd gebruik gemaakt van teksten als waarde, bijvoorbeeld `"maandag"`. Een tekst die als waarde in een programma wordt gebruikt heet een *string*. Een string is een lijst, waarvan de elementen lettertekens zijn. p. 70

Alle functies die op lijsten werken, kunnen dus ook op strings gebruikt worden. Bijvoorbeeld de expressie `"zon" ++ "dag"` geeft de string `"zondag"`, en het resultaat van de expressie tail (take 3 `"haskell"`) is de string `"as"`.

Strings worden genoteerd tussen dubbele aanhalingstekens. De dubbele aanhalingstekens geven aan dat een tekst letterlijk genomen moet worden als waarde van een string, en niet als naam van een functie. dus `"until"` is een string die uit vijf tekens bestaat, maar until is de naam van een functie. Om een string moeten daarom altijd dubbele aanhalingstekens geschreven worden.

```
? "zon"++"dag"
"zondag"
```

De elementen van een string zijn van het type *Char*. Dat is een afkorting van het Engelse woord *character*. Mogelijke characters zijn niet alleen de lettertekens, maar ook de cijfersymbolen en de leestekens. Het type *Char* is één van de vier primitieve types van Haskell (de andere drie zijn *Int*, *Float* en *Bool*).

Waarden van het type *Char* kunnen worden aangegeven door een letterteken tussen *enkele aanhalingstekens* oftewel *apostrofs* te zetten, bijvoorbeeld `'B'` of `'*'`. (Let op het verschil met omgekeerde aanhalingstekens (back quotes), die worden gebruikt om van een functie een operator te maken.) Onderstaande drie expressies hebben zeer verschillende betekenissen:

`"f"`	een lijst characters (string) die uit één element bestaat,
`'f'`	een character,
`` `f` ``	de functie `f` als binaire operator beschouwd.

De notatie met dubbele aanhalingstekens voor strings is niets anders dan een afkorting voor een opsomming van een lijst characters. Dus de volgende drie expressies zijn volkomen equivalent:

`"hallo"`	de string notatie
`['h','a','l','l','o']`	een lijst van characters
`'h':'a':'l':'l':'o':[]`	de expliciete constructie van de lijst

Voorbeelden waaruit blijkt dat een string inderdaad een lijst characters is, zijn de expressie `hd "aap"` die het character `'a'` oplevert, en de expressie `takeWhile (== 'e') "eender"` die de string `"ee"` oplevert.

Er is één leesteken dat in een string problemen geeft: het dubbele aanhalingsteken. Bij een dubbel aanhalingsteken in een string zou de string immers afgelopen zijn. Als er toch een dubbel aanhaingsteken in een string nodig is, moet daar het symbool `\` (een omgekeerde deelstreep, of *backslash*) vóór gezet worden. We zeggen dan dat het karakter *geëscapet* is. Bijvoorbeeld:

```
"Hij zei \"hallo\" en liep door"
```

Deze oplossing geeft een nieuw probleem, want nu kan het leesteken `\` zèlf weer niet in een string staan. Als dit teken in een string moet komen, moet het daarom verdubbeld worden:

```
"het teken \\ heet backslash"
```

Zo'n dubbel symbool telt als één character. Dus de lengte van de string `"\"\"\\\""` is vier.

4.2.2 Characters

De waarde van een *Char* kunnen lettertekens, cijfertekens en leestekens zijn. Het is belangrijk om enkele aanhalingstekens rond een character neer te zetten, omdat ze normaal iets anders betekenen:

expressie	type	betekenis
`'x'`	*Char*	het letterteken `'x'`
`x`	...	de naam van bijv. een parameter
`'3'`	*Char*	het cijferteken `'3'`
3	*Int*	het getal 3
`'.'`	*Char*	het leesteken punt
.	$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$	de functie-samenstellings operator

Er zijn 256 mogelijke waardes voor het type *Char*:

- 52 lettertekens
- 10 cijfertekens
- 32 leestekens en de spatie
- 33 speciale tekens
- 128 extra tekens: letters met accenten, meer leestekens enz.

De symbolen die in een string geëscapet moesten worden, worden dat ook binnen enkele aanhalingstekens. En omdat we ook een enkel aanhalingsteken tussen enkele aanhalingstekens willen kunnen zetten, escapen we die ook. Kortom:

```
dubbelepunt     = ':'
aanhalingsteken = '\"'
backslash       = '\\'
apostrof        = '\''
```

De 33 speciale characters worden gebruikt om de lay-out van een tekst te beïnvloeden. De belangrijkste speciale characters zijn de "newline" en de "tab(ulatie)". Ook deze characters kunnen worden weergegeven met behulp van een backslash: `'\n'` is het newline-character, en `'\t'` is het tabulatie-character. Het newline-character kan gebruikt worden om een resultaat van meer dan één regel te maken. Om een string af te drukken waarbij newlines geïnterpreteerd worden passen we de functie putStr er op toe:

```
? putStr "EEN\nTWEE\nDRIE"
EEN
TWEE
DRIE
```

Alle characters zijn genummerd volgens een door de Internationale Standaarden Organisatie (ISO) bepaalde codering[1]. De module *Data.Char* exporteert twee standaardfuncties die de code van een character bepalen, respectievelijk het character met een bepaalde code opleveren:

```
ord :: Char -> Int
chr :: Int -> Char
```

[1] Deze codering wordt meestal de ASCII-codering genoemd (American Standard Code for Information Interchange). Tegenwoordig is de codering internationaal erkend, en moet dus eigenlijk ISO-codering worden genoemd.

Bijvoorbeeld:

```
? ord 'A'
65
? chr 51
'3'
```

De characters zijn geordend volgens deze codering; zoals integers worden geordend door ordInt bestaat er dus voor karakters een functie ordChar. De ordening komt, wat de letters betreft, overeen met de alfabetische ordening, met dien verstande dat alle hoofdletters vóór de kleine letters komen. Deze ordening werkt ook door in strings; strings zijn immers lijsten, en die zijn lexicografisch geordend gebaseerd op de ordening van hun elementen: We kunnen nu schrijven:

```
? isort ordString ["aap", "noot", "Mies", "Wim"]
["Mies", "Wim", "aap", "noot"]
```

4.2.3 Functies op characters en strings

In de module *Data.Char* worden enkele functies gedefinieerd op characters, waarmee bepaald kan worden wat voor soort teken een gegeven character is:

```
isSpace, isUpper, isLower, isAlpha, isDigit, isAlphaNum :: Char -> Bool
isSpace c      = ord c == ord ' '  ||  ord c == ord '\t' || ord c == ord '\n'
isUpper c      = ord c >= ord 'A' && ord c <= ord 'Z'
isLower c      = ord c >= ord 'a' && ord c <= ord 'z'
isAlpha c      = isUpper c || isLower c
isDigit c      = ord c >= ord '0' && ord c <= ord '9'
isAlphaNum c = isAlpha c || isDigit c
```

Deze functies kunnen goed gebruikt worden om in de definitie van een functie op characters de verschillende gevallen te onderscheiden.

In de ISO-codering is de code van het cijferteken `'3'` niet 3, maar 51. De cijfers liggen in de codering gelukkig wel opeenvolgend. Om de numerieke waarde van een cijferteken te bepalen moet dus niet alleen de functie ord worden toegepast, maar ook 48 van het resultaat worden afgetrokken. Dat doet de functie digitValue:

```
digitValue ::    Char -> Int
digitValue c = ord c - ord '0'
```

Deze functie kan eventueel voor "onbevoegd" gebruik worden beveiligd door te eisen dat de parameter inderdaad een digit is:

```
digitValue c | isDigit c = ord c - ord '0'
```

De omgekeerde operatie wordt uitgevoerd door de functie digitChar: deze functie maakt van een integer (die tussen 0 en 9 moet liggen) het bijbehorende cijferteken:

```
digitChar :: Int -> Char
digitChar n = chr (n + ord '0')
```

Deze twee functies worden in de prelude helaas niet gedefinieerd (maar als ze nodig zijn kun je ze natuurlijk altijd zelf even definiëren).

In de *Data.Char* zitten wel twee functies om kleine letters naar hoofdletters om te rekenen en andersom:

```
toUpper, toLower :: Char -> Char
toUpper c | isLower c = chr (ord c - ord 'a' + ord 'A')
          | otherwise = c
toLower c | isUpper c = chr (ord c - ord 'A' + ord 'a')
          | otherwise = c
```

Met behulp van map kunnen deze functies op alle elementen van een string worden toegepast:

```
? map toUpper "Hallo!"
HALLO!
? map toLower "Hallo!"
hallo!
```

Alle polymorfe functies die op lijsten zijn gedefinieerd zijn ook te gebruiken op strings. Daarnaast zijn er in de prelude een paar functies gedefinieerd die specifiek op strings werken:

```
words, lines :: [Char] -> [[Char]]
unwords, unlines :: [[Char]] -> [Char]
```

De functie words splitst een string op in een aantal kleine strings, die ieder één woord van de invoerstring bevatten. De woorden worden gescheiden door spaties. De functie lines doet hetzelfde, maar dan met de afzonderlijke regels, die in de invoerstring gescheiden zijn door newline-characters (`'\n'`). Voorbeelden:

```
? words "dit is een string"
["dit", "is", "een", "string"]
? lines "eerste regel\ntweede regel"
["eerste regel", "tweede regel"]
```

De functies unwords en unlines doen het omgekeerde: ze smeden een lijst woorden, respectievelijk regels, aaneen tot één lange string:

```
? unwords ["dit", "zijn", "de", "woorden"]
"dit zijn de woorden"
? putStr (unlines ["eerste regel", "tweede regel"])
eerste regel
tweede regel
```

In het tweede commando hierboven staan er om het afgedrukte resultaat geen aanhalingstekens omdat het afdrukken het expliciete gevolg is van de (*IO*-actie) putStr.

4.2.4 Oneindige lijsten

Het aantal elementen in een lijst kan (in potentie) oneindig groot zijn. De hier volgende functie vanaf levert een oneindig lange lijst op:

$$\mathsf{vanaf\ n} = \mathsf{n} : \mathsf{vanaf}\ (\mathsf{n} + 1)$$

Natuurlijk kan een computer niet echt een oneindig aantal elementen bevatten. Gelukkig krijg je het beginstuk van de lijst al te zien terwijl de rest van de lijst nog wordt opgebouwd:

```
? vanaf 5
[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,  Ctrl-C
Interrupted!
?
```

Op het moment dat je genoeg elementen hebt gezien, kun je de berekening stoppen door op Ctrl-C te drukken.

Een oneindige lijst kan ook gebruikt worden als tussenresultaat, terwijl het eindresultaat toch eindig is. Dit is bijvoorbeeld het geval bij het probleem: “bepaal alle machten van drie die kleiner zijn dan 1000”. De eerste tien machten van drie zijn te bepalen met de volgende aanroep:

```
? map (3^) [0..9]
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683]
```

De elementen die kleiner zijn dan 1000 kunnen met de functie takeWhile hieruit genomen worden:

```
? takeWhile (<1000) (map (3^) [0..9])
[1, 3, 9, 27, 81, 243, 729]
```

Maar hoe weet je van tevoren dat 10 elementen genoeg is? De oplossing is om in plaats van $[0\,..\,9]$ de oneindige lijst vanaf 0 te gebruiken, om daarmee *alle* machten van drie te berekenen. Dat is zeker genoeg...

```
? takeWhile (<1000) (map (3^) (vanaf 0))
[1, 3, 9, 27, 81, 243, 729]
```

Deze methode kan worden toegepast dankzij het feit dat de Haskell interpreter nogal lui van aard is: werk wordt altijd zo lang mogelijk uitgesteld. Daarom wordt het resultaat van `map (3^) (vanaf 0)` niet in zijn geheel uitgerekend (dat zou oneindig lang duren). In plaats daarvan wordt eerst het eerste element berekend. Dat wordt doorgespeeld aan de buitenwereld, in dit geval de functie takeWhile. Pas als dit element verwerkt is, en takeWhile om een volgend element vraagt, wordt het tweede element uitgerekend. Vroeg of laat zal takeWhile echter niet meer om nieuwe elementen vragen (nadat het eerste getal $\geqslant 1000$ is gepasseerd). Verdere elementen worden door map dan ook niet meer uitgerekend.

4.2.5 Lazy evaluatie

De evaluatiemethode – de manier waarop expressies worden uitgerekend – van Haskell wordt *lazy evaluation* ("luie berekening") genoemd. Bij lazy evaluation wordt een (deel-)expressie pas uitgerekend als zeker is dat de waarde écht nodig is voor het resultaat. Het tegenovergestelde van lazy evaluation is *eager evaluation* ("gretige berekening"). Bij eager evaluation worden bij aanroep van een functie eerst de argumenten helmaal uitgerekend, voordat de functie wordt aangeroepen. We spreken hier ook wel van *stricte* evaluatie.

Het kunnen gebruiken van oneindige lijsten is te danken aan de lazy evaluatie. In talen waarin eager evaluatie gebruikt wordt (zoals alle imperatieve talen, en een aantal oudere functionele talen) zijn oneindige lijsten veel onhandiger te representeren. Sommige talen laten toe dat de programmeur handmatig aangeeft dat een expressie nog niet direct hoeft te worden uitgerekend, en deze moet dan ook handmatig aangeven wanneer dat dan wel moet gebeuren.

Lazy evaluatie heeft nog meer voordelen. Bekijk bijvoorbeeld de functie priem uit paragraaf 3.4.1, die kijkt of een getal een priemgetal is: p. 69

```
priem ::  Int -> Bool
priem x = delers x == [1,x]
```

Zou deze functie àlle delers van x bepalen, en die lijst vervolgens vergelijken met $[1, x]$? Welnee, dat is veel te veel werk! Bij de aanroep van priem 30 gebeurt het volgende. Eerst wordt de eerste deler van 30 bepaald: 1. Deze waarde wordt vergeleken met het eerste element van de lijst $[1, 30]$. Wat het eerste element betreft zijn de lijsten dus gelijk. Dan

wordt de tweede deler van 30 bepaald: 2. Die wordt vergeleken met de tweede waarde van $[1, 30]$: de tweede elementen van de lijsten zijn niet gelijk. De operator == "weet" dat twee lijsten nooit meer gelijk kunnen worden als er een verschillend element in zit. Daarom kan er direct *False* opgeleverd worden. De overige delers van 30 worden dus niet berekend!

Het lazy gedrag van de operator == wordt veroorzaakt door zijn definitie. De definitie van gelijkheid op lijsten ziet er als volgt uit:

```
(x : xs) == (y : ys) = x == y && xs == ys
[]       == []       = True
[]       == _        = False
_        == []       = false
```

Als x == y de waarde *False* oplevert, hoeft xs == ys niet meer uitgerekend te worden: het totale resultaat is toch altijd *False*. Dit lazy gedrag dankt de operator && op zijn beurt aan zijn definitie:

```
False && x = False
True  && x = x
```

Als de linker parameter de waarde *False* heeft, is de waarde van de rechter parameter niet nodig om het resultaat te berekenen. (Dit is de echte definitie van &&. De definitie in paragraaf 2.4.3 is ook goed, maar vertoont niet het gewenste lazy gedrag). We zeggen p. 42 dat && strict is in zijn eerste argument, maar niet in zijn tweede.

Functies die alle elementen van een lijst nodig hebben, mogen niet op oneindige lijsten worden toegepast. Voorbeelden van zulk soort functies zijn sum en length. Bij de aanroep sum (vanaf 1) of length (vanaf 1) helpt zelfs lazy evaluatie niet meer om in eindige tijd het eindresultaat te berekenen. De computer zal in zo'n geval zijn uiterste best gaan doen, maar nooit met een eindantwoord komen (tenzij het resultaat van de berekening nergens gebruikt wordt, want dan wordt de berekening natuurlijk niet uitgevoerd...).

4.2.6 Functies op oneindige lijsten

In de prelude worden een enkele functies gedefinieerd die oneindige lijsten opleveren.

De functie vanaf uit paragraaf 4.2.4 heet in werkelijkheid enumFrom. De functie wordt p. 99 meestal niet als zodanig gebruikt, omdat in plaats van enumFrom n ook [n..] geschreven mag worden. (Vergelijk de notatie [n..m] voor enumFromTo n m, die in paragraaf 4.1.1 p. 81 werd besproken.)

Een oneindige lijst waarin steeds één element herhaald wordt, kan worden gemaakt met de functie repeat:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

De aanroep repeat 't' levert de oneindige lijst "tttttttt.." op.

Een oneindige lijst die door repeat wordt gegenereerd kan weer goed gebruikt worden als tussenresultaat door een functie die wel een eindig resultaat heeft. De functie replicate bijvoorbeeld maakt een eindig aantal kopieën van een element:

```
replicate ::      Int -> a -> [a]
replicate n x = take n (repeat x)
```

Dankzij lazy evaluatie kan replicate gebruik maken van het oneindige resultaat van repeat. De functies repeat en replicate worden in de prelude gedefinieerd.

De meest flexibele functie is ook nu weer een hogere-orde functie, dat wil zeggen een functie met een functie als parameter. De functie iterate krijgt een functie en een startelement als parameter. Het resultaat is een oneindige lijst, waarin elk volgend element verkregen wordt door de functie op het vorige element toe te passen. Bijvoorbeeld:

```
iterate (+1) 3       is  [3, 4, 5, 6, 7, 8,...
iterate (*2) 1       is  [1, 2, 4, 8, 16, 32,...
iterate (/10) 5678   is  [5678, 567, 56, 5, 0, 0,...
```

De definitie van iterate, die in de prelude staat, is als volgt:

```
iterate ::      (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Deze functie lijkt een beetje op de functie until, die in paragraaf 3.3.2 werd gedefinieerd. p. 65
Ook until krijgt immers een functie en een startelement als parameter, en past de functie herhaald toe op het startelement. Het verschil is, dat until stopt als de waarde aan een bepaalde voorwaarde (die ook als parameter wordt meegegeven) voldoet. Bovendien levert until alleen de eindwaarde op (die dus aan het meegegeven stopcriterium voldoet), terwijl iterate alle tussenresultaten in een lijst stopt. Hij moet wel, want bij oneindige lijsten is er geen laatste element...

Hier volgen twee voorbeelden waarin iterate gebruikt wordt om een praktisch probleem op te lossen: de weergave van een getal als string, en het genereren van de lijst van alle priemgetallen.

Weergave van een getal als string

De functie intString maakt van een getal een string waarin de cijfers van dat getal zitten. Bijvoorbeeld: intString 5678 is de string "5678". Dankzij deze functie is het mogelijk om het resultaat van een berekening te combineren met een string, bijvoorbeeld zoals in intString $(3 * 17)$ ++ "\ euro".

De functie intString kan worden samengesteld door na elkaar een aantal functies uit te voeren. Eerst moet het getal met behulp van iterate herhaald door 10 gedeeld worden (zoals in het derde voorbeeld van iterate hierboven). De oneindige staart nullen is oninteressant, en kan worden afgekapt met takeWhile. De gewenste cijfers zijn dan steeds

het laatste cijfer van de getallen in de lijst; het laatste cijfer van een getal is de rest bij deling door 10. De cijfers staan nu nog in de verkeerde volgorde, maar dat kan worden opgelost met de functie reverse. Tenslotte moeten de cijfers (van type *Int*) nog worden omgezet in het overeenkomstige cijferteken (van type *Char*).

Een schema aan de hand van een voorbeeld maakt dit wat duidelijker:

$$
\begin{array}{c}
5678 \\
\downarrow \text{iterate } (/10) \\
[5678, 567, 56, 5, 0, 0, \ldots \\
\downarrow \text{takeWhile } (/= 0) \\
[5678, 567, 56, 5] \\
\downarrow \text{map } (\text{`rem`}10) \\
[8, 7, 6, 5] \\
\downarrow \text{reverse} \\
[5, 6, 7, 8] \\
\downarrow \text{map digitChar} \\
[\texttt{'5'}, \texttt{'6'}, \texttt{'7'}, \texttt{'8'}]
\end{array}
$$

De functie intString kan simpelweg geschreven worden als samenstelling van deze vijf functies. Let er op dat de functies in omgekeerde volgorde opgeschreven moeten worden, omdat de functiesamenstellings operator (.) de betekenis “na” heeft:

```
intString :: Int -> [Char]
intString = map digitChar
          . reverse
          . map (`rem`10)
          . takeWhile (/= 0)
          . iterate (/10)
```

Functioneel programmeren is dus echt programmeren met functies.

De lijst van alle priemgetallen

In paragraaf 3.4.1 werd een functie priem gedefinieerd die bepaalt of een getal een priemgetal is. De (oneindige) lijst van alle priemgetallen kan daarmee worden berekend door p. 69

```
filter priem [2..]
```

De functie priem gaat op zoek naar delers van een getal. Als zo’n deler groot is, duurt het dus vrij lang voordat de functie tot de conclusie komt dat een getal geen priemgetal is.

Door handig gebruik te maken van iterate is echter een veel snellere methode mogelijk. Deze methode begint ook met de oneindige lijst $[2..]$:

$$[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \ldots$$

Het eerste getal, 2, kan in de lijst van priemgetallen worden gestopt. Nu worden 2 en alle veelvouden daarvan uit de lijst weggestreept. Er blijft dan over:

$$[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots$$

Het eerste getal, 3, is een priemgetal. Dit getal en zijn veelvouden worden uit de lijst weggestreept:

$$[5, 7, 11, 13, 17, 19, 23, 25, 29, 31, \ldots$$

Hetzelfde proces wordt weer uitgevoerd, maar nu met 5:

$$[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, \ldots$$

En zo kun je doorgaan. De functie "streep veelvouden van het eerste element weg" wordt steeds uitgevoerd op het vorige resultaat. Dit is dus een toepassing van iterate, met $[2\,..]$ als startwaarde:

```
iterate streepweg [2..]
where streepweg (x:xs) = filter (not . veelvoud x) xs
  veelvoud x y = deelbaar y x
```

(Het getal y is een veelvoud van x als y deelbaar is door x). Doordat de beginwaarde een oneindige lijst is, is het resultaat hiervan een *oneindige lijst van oneindige lijsten*. Die superlijst is als volgt opgebouwd:

$$\begin{aligned}
&[[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, \ldots\\
&, [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, \ldots\\
&, [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, \ldots\\
&, [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, \ldots\\
&, [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 51, 53, \ldots\\
&, \ldots
\end{aligned}$$

Dit ding kun je nooit in zijn geheel te zien krijgen; als je hem probeert te evalueren krijg je alleen het beginstuk van de eerste rij te zien. Maar geen van de rijen hoeft compleet zichtbaar gemaakt te worden: de gewenste priemgetallen zijn de eerste elementen van de rijen. De priemgetallen worden dus bepaald door van elke lijst de head te nemen:

```
priemgetallen :: [Int]
priemgetallen = map head (iterate streepweg [2..])
  where          streepweg (x:xs) = filter (not . veelvoud x) xs
```

Door de lazy evaluatie wordt van elke lijst in de superlijst precies het gedeelte uitgerekend dat nodig is voor het gewenste deel van het antwoord. Wil je het volgende priemgetal weten, dan wordt van elke lijst het noodzakelijke stukje verder uitgerekend.

Het is vaak (zo ook in dit voorbeeld) moeilijk om je precies voor te stellen wat er op welk moment wordt uitgerekend. Maar dat hoeft ook niet: tijdens het programmeren kun je net doen alsof oneindige lijsten echt bestaan; de uitrekenvolgorde wordt door de lazy evaluatie automatisch bepaald.

4.2.7 Lijstcomprehensies

In de verzamelingenleer is een handige notatie in gebruik om verzamelingen te definiëren:

$$V = \{\ x^2\ \mid\ x \in N, x \text{ even}\ \}$$

Naar analogie van deze notatie, de zogenaamde verzamelingcomprehensie (ook wel Zermelo-Fraenkel expressie, of, korter, ZF-expressie genoemd) is in Haskell een vergelijkbare notatie beschikbaar om lijsten te construeren. Deze notatie heet dan ook een *lijstcomprehensie*. Een eenvoudig voorbeeld van deze notatie is de volgende expressie:

$$[\mathsf{x} * \mathsf{x} \mid \mathsf{x} <\!\!- [1\,..\,10]]$$

Deze expressie kan worden uitgesproken als "x kwadraat voor x uit 1 tot 10". In een lijstcomprehensie staat voor de verticale streep een expressie, waarin tot dan toe ongedefinieerde variabelen voor mogen komen. Zo'n variabele (x in het voorbeeld) dient dan wel gedefinieerd te worden in het gedeelte achter de verticale streep. De notatie "x <– xs" heeft de betekenis: "x doorloopt alle waarden van de lijst xs". Voor elk van deze waarden wordt de waarde van de expressie voor de verticale streep uitgerekend.

Bovengenoemd voorbeeld heeft dus dezelfde waarde als de expressie

$$\mathsf{map\ kwadraat}\ [1\,..\,10]$$

waarbij de functie kwadraat is gedefinieerd als

$$\mathsf{kwadraat\ x} = \mathsf{x} * \mathsf{x}$$

Het voordeel van de comprehensienotatie is dat de functie die steeds wordt uitgerekend (kwadraat in het voorbeeld) niet eerst een naam hoeft te krijgen.

De lijstcomprehensienotatie heeft nog meer mogelijkheden. Achter de verticale streep mag namelijk meer dan één lopende variabele worden gebruikt. De expressie voor de verticale streep wordt dan voor alle mogelijke combinaties uitgerekend. Bijvoorbeeld:

```
? [ (x,y) | x<-[1..2], y<-[4..6] ]
[ (1,4), (1,5), (1,6), (2,4), (2,5), (2,6) ]
```

De laatstgenoemde variabele loopt het snelst: voor elke waarde van x doorloopt y de lijst $[4\,..\,6]$.

Behalve definities van lopende variabelen mogen achter de verticale streep uitdrukkingen met de waarde *True* of *False* worden opgenomen. De betekenis daarvan wordt gedemonstreerd door het volgende voorbeeld:

```
? [ (x,y) | x<-[1..5], even x, y<-[1..x] ]
[ (2,1), (2,2), (4,1), (4,2), (4,3), (4,4) ]
```

In de resultaatlijst worden dus alleen díe x verwerkt, waarvoor even x de waarde *True* heeft.

Door elk pijltje (<−) wordt een variabele gedefiniëerd, die in de verdere expressies en in de expressie links van de verticale streep gebruikt mag worden. Zo mag de variabele x behalve in (x, y) gebruikt worden in even x en in [1 . . x]. De variabele y mag echter alleen maar gebruikt worden in (x, y). Het pijltje is een speciaal voor dit doel gereserveerd symbool, en is dus geen operator!

Strikt genomen is de lijstcomprehensie notatie overbodig. Hetzelfde effect kan bereikt worden door combinaties van map, filter en concat. De comprehensienotatie is, zeker in ingewikkelde gevallen, echter veel gemakkelijker te begijpen. Bovenstaand voorbeeld zou anders geschreven moeten worden als

```
concat (map f (filter even [1 . . 5]))
where f x = map g [1 . . x]
            where g y = (x, y)
```

hetgeen veel minder inzichtelijk is.

Een lijstcomprehensie wordt door de interpreter direct vertaald naar een overeenkomstige expressie met map, filter en concat. Net als de notatie voor intervallen is de comprehensienotatie dus puur bedoeld voor het gemak van de programmeur, syntactische suiker dus weer.

4.3 Tupels

4.3.1 Gebruik van tupels

In een lijst moet elk element hetzelfde type hebben. Het is niet mogelijk om in één lijst zowel een integer als een string te stoppen. Toch is het soms nodig om gegevens van verschillende types te groeperen. De gegevens in een bevolkingsregister bestaan bijvoorbeeld uit een string (naam), een boolean (geslacht) en drie integers (geboortedatum). Deze gegevens horen bij elkaar, maar kunnen niet in één lijst gestopt worden.

Voor dit soort gevallen is er, naast lijstvorming, nog een andere manier om samengestelde types te maken: *tupelvorming*. Een *tupel* bestaat uit een vast aantal waarden, die tot één geheel zijn gegroepeerd. De waarden mogen van verschillend type zijn (hoewel dat niet verplicht is).

Tupels worden genoteerd met ronde haakjes rond de elementen (waar bij lijsten vierkante haakjes worden gebruikt). Voorbeelden van tupels zijn:

$(1, \texttt{'a'})$	een tupel met als elementen de integer 1 en het character `'a'`;
$(\texttt{"aap"}, \mathit{True}, 2)$	een tupel met drie elementen: de string `"aap"`, de boolean *True* en het getal 2;
$([1,2], \mathsf{sqrt})$	een tupel met twee elementen: de lijst integers $[1,2]$, en de float-naar-float functie sqrt;
$(1, (2,3))$	een tupel met twee elementen: het getal 1, en het tupel van de getallen 2 en 3.

Voor elke combinatie van types vormt het tupel ervan een apart type. Daarbij is ook de volgorde van belang. Het type van tupels wordt geschreven door de types van de elementen op te sommen tussen ronde haakjes. De vier hierboven genoemde expressies kunnen dus als volgt getypeerd worden:

$$
\begin{array}{ll}
(1, \texttt{'a'}) & :: (\mathit{Int}, \mathit{Char}) \\
(\texttt{"aap"}, \mathit{True}, 2) & :: ([\mathit{Char}], \mathit{Bool}, \mathit{Int}) \\
([1,2], \mathsf{sqrt}) & :: ([\mathit{Int}], \mathit{Float} \texttt{ -> } \mathit{Float}) \\
(1, (2,3)) & :: (\mathit{Int}, (\mathit{Int}, \mathit{Int}))
\end{array}
$$

Een tupel met twee elementen wordt een 2-tupel, of ook wel een *paar* genoemd. Tupels met drie elementen heten 3-tupels, enzovoort. Er bestaan geen 1-tupels: de expressie (7) is gewoon een integer; om elke expressie mogen immers haakjes gezet worden. Wel bestaat er een 0-tupel: de waarde (), die () als type heeft.

In de prelude zijn een paar functies gedefinieerd die op 2-tupels of 3-tupels werken. Deze zijn er meteen een voorbeeld van hoe functies op tupels gedefinieerd kunnen worden: door patroonanalyse.

$$
\begin{array}{lll}
\mathsf{fst} & :: (a,b) \texttt{ -> } a & \\
\mathsf{fst} & (\mathsf{x},\mathsf{y}) & = \mathsf{x} \\
\mathsf{snd} & :: (a,b) \texttt{ -> } b & \\
\mathsf{snd} & (\mathsf{x},\mathsf{y}) & = \mathsf{y} \\
\mathsf{fst3} & :: (a,b,c) \texttt{ -> } a & \\
\mathsf{fst3} & (\mathsf{x},\mathsf{y},\mathsf{z}) & = \mathsf{x} \\
\mathsf{snd3} & :: (a,b,c) \texttt{ -> } b & \\
\mathsf{snd3} & (\mathsf{x},\mathsf{y},\mathsf{z}) & = \mathsf{y} \\
\mathsf{thd3} & :: (a,b,c) \texttt{ -> } c & \\
\mathsf{thd3} & (\mathsf{x},\mathsf{y},\mathsf{z}) & = \mathsf{z}
\end{array}
$$

Deze functies zijn polymorf, maar het is natuurlijk ook mogelijk om functies te schrijven die maar op één specifiek tupeltype werken:

$$
\begin{array}{lll}
\mathsf{f} & :: (\mathit{Int}, \mathit{Char}) \texttt{ -> } [\mathit{Char}] & \\
\mathsf{f} & (\mathsf{n}, \quad c) & = \mathsf{intString}\ \mathsf{n} \texttt{ ++ } [c]
\end{array}
$$

Als twee waarden van hetzelfde type gegroepeerd moeten worden kan daarvoor een lijst gebruikt worden. In sommige gevallen is een tupel geschikter. Een punt in het platte

vlak wordt bijvoorbeeld beschreven door twee *Float* getallen. Zo'n punt kan worden gerepresenteerd door een lijst, of door een 2-tupel. In beide gevallen is het mogelijk om functies te definiëren die op punten werken, bijvoorbeeld "afstand tot de oorsprong". De functie afstandL is de lijstversie, afstandT de tupelversie hiervan:

```
afstandL :: [Float]          -> Float
afstandL  [x, y]             =  sqrt (x * x + y * y)
afstandT :: (Float, Float)   -> Float
afstandT  (x,       y      ) =  sqrt (x * x + y * y)
```

Zolang de functie correct wordt aangeroepen is er geen verschil. Maar het zou kunnen gebeuren dat de functie elders in het programma, door een tikfout of een denkfout, met drie coördinaten wordt aangeroepen. Bij gebruik van afstandT wordt daarvoor tijdens de analyse van het programma voor gewaarschuwd: een tupel met drie getallen is een ander type dan een tupel met twee getallen. In het geval van afstandL is het programma echter goed getypeerd. Pas als de functie inderdaad gebruikt wordt blijkt dat afstandL voor lijsten met drie elementen ongedefinieerd is. Het gebruik van tupels in plaats van lijsten helpt dus in dit geval om fouten zo vroeg mogelijk op te sporen.

Nog een plaats waar tupels van pas komen zijn functies die meer dan één resultaat hebben. Functies met meerdere parameters zijn mogelijk door het Currying-mechanisme; functies die meerdere resultaten hebben, zijn het eenvoudigst te verkrijgen door die resultaten te "verpakken" in een tupel. Het tupel in z'n geheel is dan immers één resultaat.

Een voorbeeld van een functie die eigenlijk twee resultaten heeft, is de functie splitAt die in de prelude wordt gedefinieerd. Deze functie levert de resultaten van take en drop in één keer op. De functie zou dus zo gedefinieerd kunnen worden:

```
splitAt ::       Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

Het werk van beide functies kan echter in één keer worden gedaan, vandaar dat splitAt uit efficiëntieoverwegingen als volgt is gedefinieerd:

```
splitAt  :: Int -> [a] -> ([a], [a])
splitAt 0 xs =       ([], xs)
splitAt n [] =       ([], [])
splitAt n (x : xs) = (x : ys, zs)
          where (ys, zs) = splitAt (n - 1) xs
```

De aanroep splitAt 3 `"haskell"` geeft bijvoorbeeld het 2-tupel (`"has"`, `"kell"`) als resultaat. In de definitie is (bij de recursieve aanroep) te zien hoe zo'n resultaattupel gebruikt kan worden: door het te onderwerpen aan een patroonanalyse ((ys, zs) in het voorbeeld).

4.3.2 Typedefinities

Bij veelvuldig gebruik van lijsten en tupels worden typedeclaraties vaak nogal ingewikkeld. Bijvoorbeeld bij het schrijven van functies op punten, zoals de functie afstand hierboven.

De eenvoudigste functies zijn nog wel te overzien:

```
afstand :: (Float, Float) -> Float
verschil :: (Float, Float) -> (Float, Float) -> Float
```

Maar lastiger wordt het bij lijsten van punten, en vooral bij hogere-orde functies:

```
opp_veelhoek    ::      [(Float, Float)] -> Float
transf_veelhoek ::      ((Float, Float) -> (Float, Float))
                     -> [(Float, Float)] -> [(Float, Float)]
```

In zo'n geval komt een *typedefinitie* van pas. Met een typedefinitie is het mogelijk om een (duidelijkere) naam te geven aan een type, bijvoorbeeld:

```
type Punt = (Float, Float)
```

Na deze typedefinitie zijn de typedeclaraties eenvoudiger te schrijven:

```
afstand         :: Punt   -> Float
verschil        :: Punt   -> Punt            -> Float
opp_veelhoek    :: [Punt] -> Float
transf_veelhoek :: (Punt -> Punt) -> [Punt] -> [Punt]
```

Nog beter is het om ook voor "veelhoek" een typedefinitie te maken:

```
type Veelhoek  = [Punt]
opp_veelhoek    :: Veelhoek          -> Float
transf_veelhoek :: (Punt -> Punt) -> Veelhoek -> Veelhoek
```

Een paar dingen om in de gaten te houden bij typedefinities:

- het woord **type** is een, speciaal voor dit doel, gereserveerd woord;
- de naam van het nieuw gedefinieerde type moet met een hoofdletter beginnen (het is een constante, niet een variabele);
- een *typedeclaratie* specificeert het type van een functie; een *typedefinitie* definieert een nieuwe naam voor een type.

De nieuw gedefinieerde naam wordt door de interpreter puur beschouwd als afkorting. Bij het typeren van een expressie krijg je gewoon weer $(Float, Float)$ te zien in plaats van *Punt*. Als er twee verschillende namen aan één type gegeven worden, bijvoorbeeld:

```
type Punt    = (Float, Float)
type Complex = (Float, Float)
```

dan mogen die namen door elkaar gebruikt worden. Een *Punt* is hetzelfde als een *Complex* is hetzelfde als een $(Float, Float)$. (In paragraaf 6.3 wordt een methode beschreven p. 130 hoe *Punt* als een echt *nieuw* type gedefinieerd kan worden.) Een type is dus hetzelfde als een ander type als het op dezelfde manier is opgebouwd, zelfs al heeft het een andere naam; we zeggen dan dat Haskell's typesysteem gebaseerd is op *structurele equivalentie*. In veel andere programmeertalen is het typesysteem nominaal: er wordt naar de namen van types gekeken, en niet naar hoe ze opgebouwd zijn.

4.3.3 Rationale getallen

Een toepassing waarbij tupels goed gebruikt kunnen worden is een implementatie van de *rationale getallen*. De rationale getallen vormen de wiskundige verzameling $\mathbf{Q}$, getallen die als *breuk* te schrijven zijn. Voor het rekenen met rationale getallen kunnen geen *Float* getallen gebruikt worden: het is de bedoeling dat er *exact* gerekend wordt, en dat de uitkomst van $\frac{1}{2} + \frac{1}{3}$ de breuk $\frac{5}{6}$ oplevert, en niet de *Float* 0.833333.

Rationale getallen, oftewel breuken, kunnen worden gerepresenteerd door een teller en een noemer, die allebei gehele getallen zijn. De volgende typedefinitie ligt daarom voor de hand:

```
type Ratio = (Int, Int)
```

Een aantal veelgebruikte breuken kunnen een aparte naam krijgen:

```
qNul  = (0, 1)
qEen  = (1, 1)
qTwee = (2, 1)
qHalf = (1, 2)
qDerde = (1, 3)
qKwart = (1, 4)
```

Het is de bedoeling om functies te schrijven die de belangrijkste rekenkundige operaties op rationale getallen uitvoeren:

```
qMaal :: Ratio -> Ratio -> Ratio
qDeel :: Ratio -> Ratio -> Ratio
qPlus :: Ratio -> Ratio -> Ratio
qMin  :: Ratio -> Ratio -> Ratio
```

Een probleem is, dat één waarde door verschillende breuken weergegeven kan worden. Een "half" bijvoorbeeld, wordt gerepresenteerd door het tupel $(1, 2)$, maar ook door $(2, 4)$ en $(17, 34)$. Het resultaat van twee maal een kwart (twee-vierde) zou daardoor wel eens kunnen "verschillen" van een half (een-tweede). Om dit probleem op te lossen, is er een functie eenvoud nodig, die een breuk kan vereenvoudigen. Door na elke operatie op breuken deze functie toe te passen, wordt een breuk altijd op dezelfde manier gerepresenteerd. Het resultaat van twee maal een kwart kan dan veilig vergeleken worden met een half: het resultaat is *True*.

De functie eenvoud deelt de teller en de noemer van een breuk door hun *grootste gemene deler*. De grootste gemene deler (ggd) van twee getallen is het grootste getal waardoor beide deelbaar zijn. Daarnaast zorgt eenvoud ervoor, dat een eventueel minteken altijd in de teller van de breuk staat. De definitie is als volgt:

```
eenvoud (t, n) = ((signum n * t) / d, abs n / d)
  where d = ggd t n
```

Een eenvoudige definitie van ggd x y (die alleen werkt als x en y positief zijn) bepaalt de grootste deler van x waardoor y deelbaar is, gebruik makend van de functies delers en deelbaar uit paragraaf 3.4.1: p. 69

```
ggd x y = last (filter (deelbaar y') (delers x'))
  where x' = abs x
        y' = abs y
```

In de prelude wordt een functie gcd (*greatest common divisor*) gedefinieerd, die sneller werkt:

```
gcd x y = gcd' (abs x) (abs y)
  where gcd' x 0 = x
        gcd' x y = gcd' y (x `rem` y)
```

Deze methode is erop gebaseerd dat als x en y deelbaar zijn door d, dat dan ook x `rem` y (=x − (x / y) ∗ y) deelbaar is door d.

Met behulp van de functie eenvoud kunnen nu de rekenkundige functies gedefinieerd worden. Om twee breuken te vermenigvuldigen, moeten de teller en de noemer vermenigvuldigd worden ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Daarna kan het resultaat vereenvoudigd worden (tot $\frac{5}{6}$):

```
qMaal (x, y) (p, q) = eenvoud (x * p, y * q)
```

Delen door een getal is vermenigvuldigen met het omgekeerde, dus:

```
qDeel (x, y) (p, q) = eenvoud (x * q, y * p)
```

Voor het optellen van twee breuken moeten ze eerst gelijknamig worden gemaakt ($\frac{1}{4} + \frac{3}{10} = \frac{10}{40} + \frac{12}{40} = \frac{22}{40}$). Als gelijke noemer kan het product van de noemers dienen. De tellers moeten dan met de noemer van de andere breuk worden vermenigvuldigd, waarna ze kunnen worden opgeteld. Het resultaat moet tenslotte vereenvoudigd worden (tot $\frac{11}{20}$).

```
qPlus (x, y) (p, q) = eenvoud (x * q + y * p, y * q)
qMin (x, y) (p, q)  = eenvoud (x * q − y * p, y * q)
```

Het resultaat van berekeningen met rationale getallen wordt als tupel op het scherm gezet. Als dat niet mooi genoeg is, kan er eventueel een functie ratioString worden gedefinieerd:

```
ratioString :: Ratio -> String
ratioString (x, y)
   | y' == 1   = intString x'
   | otherwise = intString x' ++ "/" ++ intString y'
       where (x', y') = eenvoud (x, y)
```

4.3.4 Tupels en lijsten

Tupels komen vaak voor als elementen van een lijst. Veel gebruikt wordt bijvoorbeeld een lijst van 2-tupels, die als opzoeklijst (woordenboek, telefoonboek enz.) kan dienen. De opzoekfunctie is heel eenvoudig te definiëren met behulp van patronen; voor de lijst wordt een patroon gebruikt voor "niet-lege lijst waarvan het eerste element een 2-tupel is (en de andere elementen dus ook)".

```
zoekOp :: (a -> a -> Bool) -> [(a, b)] -> a -> b
zoekOp eq ((x, y) : ts) z
  | eq x z     = y
  | otherwise = zoekOp eq ts z
```

De functie is polymorf, dus werkt op lijsten 2-tupels van willekeurig type. Elementen van type a, de eerste component van de paren kunnen worden vergeleken met de functie eq welke als argument moet worden meegegeven.

Het op te zoeken element (van type a) is opzettelijk als tweede parameter gedefinieerd, zodat de functie zoekOp eenvoudig partieel geparametriseerd kan worden met een specifieke opzoeklijst, bijvoorbeeld:

```
telefoonNr = zoekOp eqString telefoonboek
vertaling  = zoekOp eqString woordenboek
```

waarbij telefoonboek en woordenboek apart als constante gedefinieerd kunnen worden. In beide gevallen heeft de zoeksleutel type *String*.

Een andere functie waarin lijsten van 2-tupels een rol spelen is de functie zip. Deze functie wordt in de prelude gedefinieerd. De functie zip heeft twee lijsten als parameter, die in het resultaat per element aan elkaar gekoppeld worden. Bijvoorbeeld: zip $[1,2,3]$ `"abc"` geeft de lijst $[(1,$ `'a'`$),(2,$ `'b'`$),(3,$ `'c'`$)]$. Als de parameterlijsten niet even lang zijn, is de lengte van de kortste van de twee bepalend. De definitie is zeer rechtstreeks:

```
zip :: [a] -> [b]     -> [(a, b)]
zip  []     ys        = []
zip  xs     []        = []
zip  (x : xs) (y : ys) = (x, y) : zip xs ys
```

De functie is polymorf, en kan dus op lijsten met elementen van willekeurige types worden toegepast. De naam *zip* betekent letterlijk "rits": de twee lijsten worden als het ware aan elkaar geritst.

Een hogere-orde variant van zip is de functie zipWith. Deze functie krijgt behalve twee lijsten ook een functie als parameter, die aangeeft hoe de overeenkomstige elementen aan elkaar gekoppeld moeten worden:

```
zipWith :: (a -> b -> c) -> [a] -> [b]     -> [c]
zipWith  f                  []     ys       = []
zipWith  f                  xs     []       = []
zipWith  f                  (x : xs) (y : ys) = f x y : zipWith f xs ys
```

Deze functie past een functie (met twee parameters) toe op alle elementen van twee lijsten. Behalve op zip lijkt zipWith ook sterk op map, die immers een functie (met één parameter) toepast op alle elementen van één lijst.

Gegeven de functie zipWith kan zip gedefinieerd worden als partiële parametrisatie daarvan:

```
zip =     zipWith maak2tupel
  where maak2tupel x y = (x, y)
```

4.3.5 Tupels en Currying

Met behulp van tupels is het mogelijk om functies met meer dan één parameter te schrijven, zonder het Curry-mechanisme te gebruiken. Een functie kan namelijk een tupel als (enige) parameter krijgen, waarmee toch twee waarden naar binnen gesmokkeld worden:

```
plus (x, y) = x + y
```

Deze functiedefinitie ziet er heel klassiek uit. De meeste mensen zouden zeggen dat plus een functie is met twee parameters, en dat parameters "natuurlijk" tussen haakjes staan. Maar wij zeggen dat deze functie één parameter heeft, en wel een tupel; de definitie vindt plaats met behulp van een patroon voor een tupel.

De Curry-methode is overigens vaak te prefereren boven de tupelmethode. Gecurryde functies zijn immers partieel te parametriseren, en functies met een tupelparameter niet. Alle standaardfuncties met meer dan één parameter werken dan ook volgens de Curry-methode.

In de prelude wordt een functietransformatie (functie met functie als parameter en andere functie als resultaat) gedefinieerd, die van een gecurryde functie een functie met tupelparameter maakt. Deze functie heet uncurry:

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

Andersom is er een functie curry die van een functie met tupelparameter een gecurryde functie maakt. Dus curry plus, met plus zoals hierboven, kan wèl partieel geparametriseerd worden.

Opgaven

4.1 Define a function which returns the last element of a list.

4.2 Define a function that returns the one but last element of a list.

4.3 Define an operator (!!) which returns the i^{th} element of a list

4.4 Define the function that returns the length of a list.

4.5 Define a function that reverses the order of the elements in a list.

4.6 Define a function that determines whether a list is a palindrome.

4.7 Define the function concat :: $[[a]]$ -> $[a]$ which flattens a list of lists: concat $[[1,2],[3],[],[4,5]]$ evaluates to $[1,2,3,4,5]$.

4.8 Define a function remSuccessiveDuplicates which removes succesive repeated elements from a list: $[1,2,2,3,2,4]$ is mapped to $[1,2,3,2,4]$.

4.9 Define a function that groups successive duplicate elements in a list into sublists: $[1,2,2,3,2,4]$ is mapped to $[[1],[2,2],[3],[2],[4]]$.

4.10 Define a function that determines the run-length encoding' of a list: $[1,2,2,3,2,4]$is mapped to $[(1,1),(2,2),(1,3),(1,2),(1,4)]$

4.11 Verify that the definition of ++ indeed maps $[1,2]$ ++ $[]$ to $[1,2]$. Hint: write $[1,2]$ as $1:(2:[])$

4.12 Define the function concat using foldr

4.13 Which of the following expressions returns *True* for all lists xs, and which *False*

```
[[]] ++ xs   == xs
[[]] ++ xs   == [xs]
[[]] ++ xs   == [[], xs]
[[]] ++ [xs] == [[], xs]
[xs] ++ []   == [xs]
[xs] ++ [xs] == [xs, xs]
```

4.14 The function filter can be defined in terms of concat and map:

```
filter p = concat . map box
  where box x =
```

Complete the definition of box.

4.15 Use the function iterate to give a non-recursive definition of repeat.

4.16 Write a function which takes two lists and removes all the elements from the second list from the first list. (This function is defined in Data.List as (\\).)

4.17 Use the functions map and concat to define the following value without list-comprehensions:

$$[(\mathsf{x}, \mathsf{y} + \mathsf{z}) \mid \mathsf{x} <\!\!- [1\,..\,10], \mathsf{y} <\!\!- [1\,..\,\mathsf{x}], \mathsf{z} <\!\!- [1\,..\,\mathsf{y}]]$$

4.18 Het vereenvoudigen van breuken is niet nodig als breuken nooit direct met elkaar vergeleken worden. Schrijf een functie qEq die gebruikt kan worden in plaats van ==. Deze functie levert *True* op als twee breuken dezelfde waarde hebben, ook als de breuken niet vereenvoudigd zijn.

4.19 Schrijf de vier rekenkundige functies voor het rekenen met *complexe getallen*. Complexe getallen zijn getallen van de vorm $a + bi$, waarbij a en b reële getallen zijn, en i een "getal" is met de eigenschap $i^2 = -1$. Hint: leid voor de deelfunctie eerst een formule af voor $\frac{1}{a+bi}$ door x en y op te lossen uit $(a+bi) * (x+yi) = (1+0i)$.

4.20 Schrijf een functie stringInt, die van een string de overeenkomstige integer maakt. Bijvoorbeeld: stringInt `"123"` levert de waarde 123. Beschouw daarvoor de string als lijst characters, en bepaal welke operator tussen de characters moet staan. Moet je daarbij van rechts of van links beginnen?

4.21 We kunnen een matrix representeren als een lijst van even lange lijsten. Schrijf een functie transpose :: $[[a]]$ -> $[[a]]$, die het i^e element van de j^e lijst op het j^e element van de i^e lijst afbeeldt. Hint: je kunt gebruik maken van de functie:

```
zipWith op (x : xs) (y : ys) = (x `op` y) : zipWith xs ys
zipWith op _        _        = []
```

5 Type inference

5.1 Introduction

When designing a new programming language, a major choice to make is how extensive the type system of the language should be, and whether it should be *strongly* or *weakly typed*: do you want to have precise guarantees when you write programs, or not. One may even want to do without any types at all. But values like strings and boolean and numbers do form conceptually different classes of values to most programmers, and hence it does make sense to use *types* to classify them. Such a choice then makes it an error to, say, concatenate two numbers (like 10 ++ 20 in Haskell), or compare a boolean to a string (*True* <= `"False"`).

When the type correctness of a program is enforced at *compile time*, then we call a language *statically typed*, otherwise we say it is *dynamically typed*. Static typing guarantees that no type errors occur at run-time. The often-coined phrase (originally by Robin Milner) here is "well-typed programs do not go wrong". Moreover, in a dynamically typed language types must be checked at run-time which imposes a run-time overhead. Many languages that are dynamic and interpreted are dynamically typed. Compiled languages are generally statically typed.

But what exactly is "do not go wrong"? Well, that depends on the language. For example, PHP was designed to not perform any compile-time type checking, but the language itself was defined in a way that values of (almost) any kind could be silently converted (coerced) into values of (almost) any other kind. In other words, designers of PHP might hold that PHP programs could not really "go wrong", i.e., they would typically not crash due to a problem arising from type incompatibility (resources form an exception here, but let's ignore that for the moment). So, technically, PHP was close to being statically typed. But some of the coercions were bound to lose the program so much information, that one might also say that having them in any program was very likely to be a bug, at the very least suspicious. For example, using an array in a context in which a string was expected, makes the PHP interpreter convert the array to the string "Array", essentially throwing away all the content of the array in the process. So the statement

```
print $a
```

does not print the content of the array on screen as one might at first believe, but simply displays `Array` instead. Something similar happens when you "print" objects.

On the other extreme, some languages have such a powerful type system that it can be checked at compile-time that an index will fall within the range of allowed indices for a list or array. However, these languages tend to be on the cutting-edge of language research and are not yet suitable for major uptake in industry.

Haskell lies somewhere in between. It has a powerful type system that we tend to regard as stronger than that of object-oriented languages such as Java and C#. However, since the languages are so different, an objective comparison is hard to make.

One remarkable property of the Haskell type system is that for most functions we do not have to specify a type. Instead, the types are *inferred* by the compiler. This makes that Haskell programs can be very short, and also that we may easily change programs, since in general there is no need to update type specifications of many functions. It is however considered good programming practice to include the types in the code once development has been completed: they both serve as a guard against further unintentional changes and as cheap, machine-checked documentation.

Not everybody agrees that strong typing is a good thing. As Martin Fowler writes in his book on domain-specific languages (`http://martinfowler.com/books/dsl.html`, he prefers to catch type errors by writing a good suite of unit tests that he needs to write anyway to also catch other kinds of errors. Indeed, programming in a strongly type language should not make you over-confident: there are kinds of mistakes that the Haskell type system does not help you to avoid. For example, Haskell will allow you to run `head []` and get a run-time error. Division by zero is another kind of error that is typically only caught at run-time. And Haskell certainly does not help you when you accidentally write $<$ instead of $>$ in your program. This kind of *logical* error cannot be caught by the type inferencer, and like Fowler you will typically resort to unit-testing of some kind. For Haskell, we typically use `QuickCheck` (see later in this course). Coming back to Fowler, we should mention that tests can only show the presence of bugs not their absence: a program is only as correct as its set of tests imply. A type system can prove the absence of type inconsistencies which is independent of how good your tests are (if we assume the correctness of the compiler as a given). Personally, we prefer to see as much as possible handled by the compiler, or, if necessary, the run-time, and this includes the testing of the application.

Before we shall look at the core of Haskell's type system, we should mention a disadvantage of the static typing discipline: theoretically, no type system can be statically enforced that exactly rejects all programs that may go wrong, and accepts all programs that will always go right. This means that any type system is an approximation. Since we want guarantees, a statically typed language takes for granted that some programs that will always go right will be rejected by the compiler. Language designers can always make the type system a bit more precise, and allow more programs that do not go wrong, but like we said: we can never include them all. Practically, this means that sometimes you write a program that you know will not go wrong, but that the compiler refuses to accept. Typically, you will then need to program around the limitation.

5.2 The Haskell type system

Haskell is a large language and many of its extensions have made the complete type system quite intricate. At the core however we find a rather simple type system which is known as the *Damas-Hindley-Milner* or sometime just *Hindley-Milner* type system. This system was for the first time employed in the design of the strict functional programming language ML.

Since it is useful when programming in Haskell to have an idea of how the inference algorithm works we give a couple of examples. A full treatment of the system would go too far at this point.

There are a couple of things to keep in mind when trying to find out what the type is of some expression:

(1) The type of a parameter is the same as the type of the occurrences of the corresponding identifier in an expression.

(2) Every application in an expression generates an equation: the type of the argument should be equal to the type that the function expects.

(3) If we call a polymorphic function, we actually call a monomorphic instance of that function, i.e., we are free to choose what types to take for the polymorphic types occurring in the type of the function, but we have to choose.

(4) If our systems of equations does not fix a specific type variables in an type, we may generalise that type to a polymorphic type. Note that this only makes sense if we bind the value to an identifier, since otherwise the expression is part of another expression where we need a monomorphic instance.

These rules may look a bit cryptic at this point, but we will see how they work out through a list of examples.

5.3 `\x -> x`

We start out by using the rules above to find a type for the expression `\x -> x`. Using rule (1) we conclude that if the type of the parameter is some type `t1` then also the result is of the body is of type `t1`. From this we conclude that the type of the whole expression matches `t1 -> t1`, for any `t1` we want to choose; we do not have further equations available to find out something about `t1`. Hence we apply rule (4) and conclude `\x -> x` :: $\forall a$. a `->` a. This type (scheme) reads as: the expression `\x -> x` has as type a `->` a whatever type we choose for the type variable a. In effect, the type $\forall a$. a `->` a describes an infinite set of types.

There is one thing you need to be aware of: consider what happens when you go into `ghci` and write

```
Prelude> let id =  x -> x
Prelude> :t id
id :: t -> t
Prelude> id 2
2
Prelude> id "2"
"2"
Prelude> id (id  2)
2
```

The ∀ is not mentioned in the type of id! But as you can see from the fact that it is okay to apply id to 2, "2", and even to 2 and id 2 in one expression, it is the case that id may be applied to values of any type. So it is there alright, but the interpreter leaves it implicit. Maybe superfluously we note that the polymorphic type $\forall a\ .\ a \rightarrow a$ and $\forall \mathsf{t}\ .\ \mathsf{t} \rightarrow \mathsf{t}$ describe the exact same set of types, so we do not distinguish between them. Technically, we call these polymorphic types *equivalent up to alpha-renaming*.

5.4 map

Consider the definition of map:

$$\mathsf{map\ f\ [\,]} = [\,]$$
$$\mathsf{map\ f\ (x : xs)} = (\mathsf{f\ x} : \mathsf{map\ f\ xs})$$

From the first equation we conclude that the type of the second parameter is a list, but we do not learn anything about the elements of that list, so let us assume it is [t1] for some type t1. From the [] at the right hand side of this equation we only learn that there must be some type t2 such that the result is of type [t2].

Using this information we may conclude from the pattern (x : xs) that x must also be of type t1. From the expression f x at the right hand side we conclude that f apparently is a function, which takes t1 as a parameter, and its result becomes an element of the result list, and thus must be of type t2.

So we conclude that the type of map is a function which takes two parameters: a function from t1 to t2 and a list of type [t1], and that it returns a list of type [t2], hence the type for map we find here is: (t1 -> t2) -> [t1] -> [t2].

If we now check what the type is of the expression map f xs on the right hand side we see that it is again [t2] for the type of map we just deduced. Here we do not learn anything new, but just check the definition for consistency.

By applying rule (4) and making the polymorphic positions explicit in our notation we finally deduce that $\mathsf{map} :: \forall a\ b\ .\ (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ for the definition above.

5.5 until even

In order to compute the most general type of an expression like until even we first repeat the definitions of until and even:

$$\mathsf{until} :: (a \to Bool) \to (a \to a) \to a \to a$$
$$\mathsf{even} :: Int \to Bool.$$

Despite the fact that the function until expects three arguments it is only applied to one here; so the result will be a function which still expects (at least) two more arguments:

$$\mathsf{until} :: \underbrace{(a \to Bool)}_{\text{argument type}} \to \underbrace{(a \to a) \to a \to a}_{\text{result type}}.$$

The function is polymorphic in its first argument (we have a type variable a at that position) and it is applied to an arguments of type $Int \to Bool$:

$$\underbrace{a \to Bool}_{\text{argument type of until}} \equiv \underbrace{Int \to Bool}_{\text{type of even}} \quad \Rightarrow \quad \begin{cases} a & \equiv & Int \\ Bool & \equiv & Bool. \end{cases}$$

Here, the notation t1\equiv t2 (for types t1 and t2) expresses that, for the program from which it arose to be type correct, the two types should be (made) equal. In some cases, to make them equal we have to instantiate a type variable with a more specific type.

In the example, the second equivalence ($Bool \equiv Bool$) is trivially true; the first equivalence ($a \equiv Int$) makes that we have to replace all occurrences of type variable a in the type of until by Int:

$$\mathsf{until} :: \underbrace{(\cancel{a}^{Int} \to Bool)}_{\text{argument type}} \to \underbrace{(\cancel{a}^{Int} \to \cancel{a}^{Int}) \to \cancel{a}^{Int} \to \cancel{a}^{Int}}_{\text{result type}}.^{Int}$$

This leads us to:

$$\mathsf{until} :: \underbrace{(Int \to Bool)}_{\text{type of even}} \to \underbrace{(Int \to Int) \to Int \to Int}_{\text{type of until even}}.$$

For until even we thus find:

$$\mathsf{until\ even} :: (Int \to Int) \to Int \to Int.$$

5.6 until or

A similar deduction we can make for the expression until or:

$$\begin{aligned}&\mathsf{until} :: (a \to Bool) \to (a \to a) \to a \to a\\ &\mathsf{or} \quad :: [Bool] \to Bool.\end{aligned}$$

Again until is being applied to a single argument:

$$\mathsf{until} :: \underbrace{(a \to Bool)}_{\text{argument type}} \to \underbrace{(a \to a) \to a \to a}_{\text{result type}}.$$

Making the argument type agree with the type of or gives:

$$\underbrace{a \to Bool}_{\text{argument type of until}} \equiv \underbrace{[Bool] \to Bool}_{\text{type of or}}$$

$$\Rightarrow \begin{cases} a \equiv [Bool] \\ Bool \equiv Bool. \end{cases}$$

The first equivalence leads to a substitution:

$$\mathsf{until} :: \underbrace{(\cancel{a}^{[Bool]} \to Bool)}_{\text{argument type}} \to \underbrace{(\cancel{a}^{[Bool]} \to \cancel{a}^{[Bool]}) \to \cancel{a}^{[Bool]} \to \cancel{a}^{[Bool]}}_{\text{result type}}$$

which can be written as:

$$\mathsf{until} :: \underbrace{([Bool] \to Bool)}_{\text{type of or}} \to \underbrace{([Bool] \to [Bool]) \to [Bool] \to [Bool]}_{\text{type of until or}}.$$

Finally we get:

$$\mathsf{until\ or} :: ([Bool] \to [Bool]) \to [Bool] \to [Bool].$$

5.7 foldr (&&) *True*

For foldr (&&) *True* we start with:

$$\begin{aligned}&\mathsf{foldr} \ :: (a \to b \to b) \to b \to [a] \to b\\ &(\&\&) :: Bool \to Bool \to Bool\\ &True \ :: Bool.\end{aligned}$$

The function foldr expects three arguments but gets passed two:

$$\mathsf{foldr} :: \underbrace{(a \to b \to b)}_{1^{e}\text{ argument type}} \to \underbrace{b}_{2^{e}\text{ argument type}} \to \underbrace{[a] \to b}_{\text{result type}}.$$

For the first argument, (&&), we find:

$$\underbrace{a \to b \to b}_{1^{e}\text{ argument type of foldr}} \equiv \underbrace{Bool \to Bool \to Bool}_{\text{type of (\&\&)}}$$

$$\Rightarrow \begin{cases} a \equiv Bool \\ b \equiv Bool. \end{cases}$$

Substituting leads to:

$$\mathsf{foldr} :: \underbrace{(\overset{Bool}{a} \to \overset{Bool}{b} \to \overset{Bool}{b})}_{1^{e}\text{ argument type}} \to \underbrace{\overset{Bool}{b}}_{2^{e}\text{ argument type}} \to \underbrace{[\overset{Bool}{a}] \to \overset{Bool}{b}}_{\text{result type}},$$

or

$$\mathsf{foldr} :: \underbrace{(Bool \to Bool \to Bool)}_{\text{type of (\&\&)}} \to \underbrace{Bool}_{\text{argument type of foldr (\&\&)}} \to \underbrace{[Bool] \to Bool}_{\text{result type of foldr (\&\&)}}.$$

For the seconde argument, *True*,this leads to:

$$\underbrace{Bool}_{\text{argument type of foldr (\&\&)}} \equiv \underbrace{Bool}_{\text{type of } True}.$$

This equivalence does not provide us with extra information so we conclude:

$$\mathsf{foldr} :: \underbrace{(Bool \to Bool \to Bool)}_{\text{type of (\&\&)}} \to \underbrace{Bool}_{\text{type of } True} \to \underbrace{[Bool] \to Bool}_{\text{type of foldr (\&\&) } True}$$

and thus

$$\mathsf{foldr}\ (\&\&)\ True :: [Bool] \to Bool.$$

5.8 foldr (&&)

The deduction of the type of foldr (&&) proceeds along the same lines as that of foldr (&&) *True*, so with 1

$$\begin{array}{l}\mathsf{foldr} \;::(a \mathrel{->} b \mathrel{->} b) \mathrel{->} b \mathrel{->} [a] \mathrel{->} b \\ (\&\&) :: \mathit{Bool} \mathrel{->} \mathit{Bool} \mathrel{->} \mathit{Bool},\end{array}$$

but this time we apply foldr to a single argument only:

$$\mathsf{foldr} :: \underbrace{(a \mathrel{->} b \mathrel{->} b)}_{\text{argument type}} \mathrel{->} \underbrace{b \mathrel{->} [a] \mathrel{->} b}_{\text{result type}}.$$

For the argument we find:

$$\underbrace{a \mathrel{->} b \mathrel{->} b}_{\text{argument type of foldr}} \quad \equiv \quad \underbrace{\mathit{Bool} \mathrel{->} \mathit{Bool} \mathrel{->} \mathit{Bool}}_{\text{type of (\&\&)}}$$

$$\Rightarrow \left\{ \begin{array}{lcl} a & \equiv & \mathit{Bool} \\ b & \equiv & \mathit{Bool}, \end{array} \right.$$

from which the substitution

$$\mathsf{foldr} :: \underbrace{(\cancel{a}^{\mathit{Bool}} \mathrel{->} \cancel{b}^{\mathit{Bool}} \mathrel{->} \cancel{b}^{\mathit{Bool}})}_{1^{\text{e}}\text{ argument type}} \mathrel{->} \underbrace{\cancel{b}^{\mathit{Bool}}}_{2^{\text{e}}\text{ argument type}} \mathrel{->} \underbrace{[\cancel{a}^{\mathit{Bool}}] \mathrel{->} \cancel{b}^{\mathit{Bool}}}_{\text{result type}},$$

follows:

$$\mathsf{foldr} :: \underbrace{(\mathit{Bool} \mathrel{->} \mathit{Bool} \mathrel{->} \mathit{Bool})}_{\text{type of (\&\&)}} \mathrel{->} \underbrace{\mathit{Bool} \mathrel{->} [\mathit{Bool}] \mathrel{->} \mathit{Bool}}_{\text{type of foldr (\&\&)}}.$$

Finally we thus find:

$$\mathsf{foldr}\ (\&\&) :: \mathit{Bool} \mathrel{->} [\mathit{Bool}] \mathrel{->} \mathit{Bool}.$$

5.9 foldr until

The situation becomes more interesting once we have more then a single polymorphic function in an expression, as in e.g. in foldr until:

$$\mathsf{foldr} :: (a \to b \to b) \to b \to [a] \to b$$
$$\mathsf{until} :: (a \to \mathit{Bool}) \to (a \to a) \to a \to a.$$

To avoid confusion we start out by rewriting the type of until using type variables which *do not* occur in the type of foldr[1]:

$$\mathsf{foldr} :: (a \to b \to b) \to b \to [a] \to b$$
$$\mathsf{until} :: (c \to \mathit{Bool}) \to (c \to c) \to c \to c.$$

Here too foldr is applied to a single argument:

$$\mathsf{foldr} :: \underbrace{(a \to b \to b)}_{\text{argument type}} \to \underbrace{b \to [a] \to b}_{\text{result type}}.$$

matching the type leads us to:

$$\underbrace{a \to b \to b}_{\text{argument type of foldr}} \quad \equiv \quad \underbrace{(c \to \mathit{Bool}) \to (c \to c) \to c \to c}_{\text{type of until}}$$

$$\Rightarrow \begin{cases} a & \equiv & c \to \mathit{Bool} \\ b & \equiv & c \to c, \end{cases}$$

which results in the substitution:

$$\mathsf{foldr} :: \underbrace{(\cancel{a}^{(c \to \mathit{Bool})} \to \cancel{b}^{(c \to c)} \to \cancel{b}^{(c \to c)})}_{\text{argument type}} \to \underbrace{\cancel{b}^{(c \to c)} \to [\cancel{a}^{(c \to \mathit{Bool})}] \to \cancel{b}^{(c \to c)}}_{\text{result type}}.$$

So:

$$\begin{aligned} \mathsf{foldr} \quad :: \quad & \underbrace{((c \to \mathit{Bool}) \to (c \to c) \to c \to c)}_{\text{type of until}} \to \\ & \underbrace{(c \to c) \to [c \to \mathit{Bool}] \to c \to c}_{\text{type of foldr until}}. \end{aligned}$$

For foldr until we have got now:

$$\mathsf{foldr} :: (c \to c) \to [c \to \mathit{Bool}] \to c \to c.$$

[1] We may do so because in until all type variables are polymorphic. This means that we get a valid type for until whatever we substitute for a. In particular when we replace a everywhere in until with another type variable, like c in this example, we obtain such a valid type. And since we replace a type variable with another (fresh) type variable, we do not even restrict the type of until by doing so.

Following conventions we now finally replace the type variable c by a:

$$\mathsf{foldr} :: (a \text{ -> } a) \text{ -> } [a \text{ -> } \mathit{Bool}] \text{ -> } a \text{ -> } a.$$

5.10 map filter

In case of map filter we have

$$\mathsf{map} :: (a \text{ -> } b) \text{ -> } [a] \text{ -> } [b]$$
$$\mathsf{filter} :: (c \text{ -> } \mathit{Bool}) \text{ -> } [c] \text{ -> } [c],$$

where we have chosen fresh type variables in the type of filter. The function map gets a single argument:

$$\mathsf{map} :: \underbrace{(a \text{ -> } b)}_{\text{argument type}} \text{ -> } \underbrace{[a] \text{ -> } [b]}_{\text{result type}}.$$

matching the types of the arguments,

$$\underbrace{a \text{ -> } b}_{\text{argument type of map}} \quad \equiv \quad \underbrace{(c \text{ -> } \mathit{Bool}) \text{ -> } [c] \text{ -> } [c]}_{\text{type of filter}}$$

$$\Rightarrow \begin{cases} a & \equiv & c \text{ -> } \mathit{Bool} \\ b & \equiv & [c] \text{ -> } [c], \end{cases}$$

gives

$$\mathsf{map} :: \underbrace{(\cancel{a} \overset{(c \to \mathit{Bool})}{\text{->}} \cancel{b})}_{\text{argument type}} \overset{[c] \to [c]}{\text{->}} \underbrace{[\cancel{a}] \overset{(c \to \mathit{Bool})}{\text{->}} [\cancel{b}]}_{\text{result type}} \overset{[c] \to [c]}{}.$$

and thus

$$\mathsf{map} :: \underbrace{((c \text{ -> } \mathit{Bool}) \text{ -> } [c] \text{ -> } [c])}_{\text{type of filter}} \text{ -> } \underbrace{[c \text{ -> } \mathit{Bool}] \text{ -> } [[c] \text{ -> } [c]]}_{\text{type of map filter}}$$

Finally we get for map filter

$$\mathsf{map\ filter} :: [c \text{ -> } \mathit{Bool}] \text{ -> } [[c] \text{ -> } [c]],$$

or:

$$\mathsf{map\ filter} :: [a \text{ -> } \mathit{Bool}] \text{ -> } [[a] \text{ -> } [a]].$$

5.11 map map

Finally we take a look at map map, which is special since it contains two occurrences of the same polymorphic function. We create a special version for each of these occurrences, carefully choosing distinct type variables:

$$\mathsf{map} :: (a \to b) \to [a] \to [b]$$
$$\mathsf{map} :: (c \to d) \to [c] \to [d]$$

with map applied to a single argument:

$$\mathsf{map} :: \underbrace{(a \to b)}_{\text{argument type}} \to \underbrace{[a] \to [b]}_{\text{result type}}.$$

Matching

$$\underbrace{a \to b}_{\text{argument type of map}} \equiv \underbrace{(c \to d) \to [c] \to [d]}_{\text{type of map}}$$

$$\Rightarrow \begin{cases} a \equiv c \to d \\ b \equiv [c] \to [d], \end{cases}$$

gives

$$\mathsf{map} :: \underbrace{(\overset{(c \to d)}{a} \to \overset{[c] \to [d]}{b})}_{\text{argument type}} \to \underbrace{[\overset{(c \to d)}{a}] \to [\overset{[c] \to [d]}{b}]}_{\text{result type}}$$

and thus

$$\mathsf{map} :: \underbrace{((c \to d) \to [c] \to [d])}_{\text{type of map}} \to \underbrace{[c \to d] \to [[c] \to [d]]}_{\text{type of map map}}.$$

It is important to realize that matching does *not* lead to the equations: $a \equiv (c \to d) \to [c]$ and $b \equiv [d]$. Why is that? If we include in the type of map the parentheses that we have omitted due to the right associativity of $\to$ we obtain $((c \to d) \to ([c] \to [d]))$. If we then match the type with $a \to b$, then the top-level $\to$s indeed match, and we still have to match a with $c \to d$ and b with $[c] \to [d]$.

Finally we get:

$$\mathsf{map\ map} :: [c \to d] \to [[c] \to [d]],$$

or

$$\mathsf{map\ map} :: [a \to b] \to [[a] \to [b]].$$

Exercises

5.1 What is the type of foldr map?

(1) $[a] \to [a \to a] \to [a]$

(2) $[a] \to [[a \to a]] \to [a]$

(3) $[a] \to [[a \to a] \to [a]]$

(4) $[[a]] \to [a \to a] \to [a]$

5.2 What is the type of map . foldr?

(1) $(a \to a \to a) \to [a] \to [[a] \to a]$

(2) $(a \to a \to a) \to [b] \to [b \to a]$

(3) $(b \to a \to a) \to [a] \to [[b] \to a]$

(4) $(b \to a \to a) \to [b] \to [[a] \to a]$

5.3 Which of the following is the type of concat . concat

(1) $[[a]] \to [[a]] \to [[a]]$

(2) $[[a]] \to [[a]] \to [a]$

(3) $[[[a]]] \to [a]$

(4) $[a] \to [[a]] \to [a]$

5.4 What is the type of map (map map)?

(1) $[[a \to b]] \to [[[a] \to [b]]]$

(2) $[a \to b] \to [[[a] \to [b]]]$

(3) $[[a \to b]] \to [[[a \to b]]]$

(4) $[[a \to b] \to [[a] \to [b]]]$

5.5 Which observation is correct when comparing the types of (map map) map and map (map map)?

(1) The type of the first is less polymorphic than the type of the second.

(2) The type of the first is more polymorphic than the type of the second.

(3) The types are the same, since function composition is associative.

(4) One of the expressions does not have any type at all.

6 Datastructuren

We hebben nu gezien dat Haskell een aantal ingebouwde types kent: getallen (*Int*, *Float*), booleans (*Bool*), letters (*Char*), lijsten en tupels. In principe kan je met die types alle soorten informatie modelleren. Soms is het echter duidelijker en is er minder kans op fouten als je een eigen type gebruikt. In Haskell is het daarom ook mogelijk om zelf nieuwe types te definiëren, zogenaamde *datatypes*.

6.1 Enumeratietypes

De meest eenvoudige datatypes zijn enumeratietypes. De definitie van een enumeratietype bestaat uit een opsomming van de mogelijke waarden van dat type. Het kan gebruikt worden als je een eindig aantal mogelijkheden wilt representeren, bijvoorbeeld windrichtingen of een beperkt aantal kleuren. Windrichtingen zou je ook kunnen coderen als getallen en dan zou je kunnen afspreken dat 0 noord is, 1 oost enzovoort. Maar wie weerhoudt je er dan van om een negatief getal op te schrijven of een getal groter dan 3? Juist, niemand. Met een eigen type kun je dit soort problemen wel voorkomen. Voor windrichtingen zou de definitie er zo uit zien:

```
data Richting = Noord | Oost | Zuid | West
```

Een datatype definitie begint met het gereserveerde woord **data**. Dan komt de naam van het nieuwe type en een =-teken. Daarachter staan de namen van de mogelijke waarden gescheiden door verticale strepen. Deze namen moeten uit dezelfde letters bestaan als functienamen en als extra eis moet de eerste letter een hoofdletter zijn. We noemen deze namen *constructoren* omdat je er waarden van het nieuwe type (hier Richting) mee kunt construeren. De constructoren hebben als type Richting: Noord is van het type Richting en dat geldt ook voor de andere richtingen. Dat maakt het mogelijk om een lijst te maken met verschillende richtingen er in; ze hebben immers hetzelfde type:

```
verticaleRichtingen :: [Richting]
verticaleRichtingen = [Noord, Zuid]
```

Functies op dit soort types kunnen gewoon met behulp van patronen worden geschreven, bijvoorbeeld:

```
move                :: Richting -> (Int, Int) -> (Int, Int)
move Noord (x, y) = (x    , y + 1)
move Oost  (x, y) = (x + 1, y    )
move Zuid  (x, y) = (x    , y - 1)
move West  (x, y) = (x - 1, y    )
```

De voordelen van zo'n eindig type boven een codering met integers of characters zijn:

- functiedefinities zijn duidelijker doordat de namen van de elementen gebruikt kunnen worden, in plaats van obscure coderingen;
- het typesysteem klaagt als je richtingen per ongeluk zou optellen (als de richtingen door integers gecodeerd werden, dan zou dit geen foutmelding geven, met alle vervelende gevolgen van dien).

Eindige types zijn niets nieuws: in feite kan het type *Bool* op deze manier gedefinieerd worden:

```
data Bool = False | True
```

Dit is ook de reden dat *False* en *True* met een hoofdletter geschreven moeten worden: het zijn de constructoren van *Bool*. (Deze definitie staat overigens niet echt in de prelude. Booleans zijn niet "voorgedefinieerd" maar "ingebouwd". De reden daarvoor is, dat andere ingebouwde taalconstructies de Booleans al moeten "kennen", zoals gevalsonderscheid met | in een functiedefinitie.)

Ook het type *Ordering* is natuurlijk op deze manier gedefinieerd in de prelude:

```
data Ordering = LT | EQ | GT
```

6.2 Constructoren met parameters

Alle elementen van een lijst moeten hetzelfde type hebben. In een tupel mogen waarden van verschillend type worden opgeslagen, maar bij tupels is het aantal elementen weer niet variabel. Soms wil je echter een lijst maken, waarvan bijvoorbeeld sommige elementen integers zijn, en andere elementen characters.

Met een datadefinitie is het mogelijk een type IntOfChar te maken, die als elementen zowel de integers als de characters heeft:

```
data IntOfChar = EenInt Int
               | EenChar Char
```

Dit is niet alleen maar een opsomming van constructoren; je ziet hier types staan achter EenInt en EenChar. Constructoren kunnen parameters hebben, zo blijkt uit dit voorbeeld, en dan noemen we ze ook wel *constructorfuncties*. Om een waarde te construeren van het type IntOfChar dienen we een van de constructorfuncties toe te passen op een parameter van het juiste type:

```
getal :: IntOfChar
getal = EenInt 4

letter :: IntOfChar
letter = EenChar 'a'
```

We kunnen hiermee ook een "gemengde" lijst maken:

```
xs :: [IntOfChar]
xs = [EenInt 1, EenChar 'a', EenInt 2, EenInt 3]
```

De enige prijs die je moet betalen, is dat elk element gemarkeerd moet worden met de constructorfunctie EenInt of EenChar. Deze functies zijn te beschouwen als conversiefuncties:

```
EenInt  :: Int -> IntOfChar
EenChar :: Char -> IntOfChar
```

waarvan het gebruik vergelijkbaar is met dat van ingebouwde conversiefuncties zoals

```
truncate :: Float -> Int
chr      :: Int -> Char
```

Met patroonherkenning kunnen we functies schrijven die iets met waarden van het nieuwe type doen:

```
toonIntOfChar               :: IntOfChar -> String
toonIntOfChar (EenInt i)    = showInt i
toonIntOfChar (EenChar c) = [c]
```

Merk op dat de ronde haakjes nodig zijn omdat het er anders uitziet alsof toonIntOfChar twee parameters krijgt!

6.3 Beschermde types

In paragraaf 4.3.2 werd een nadeel genoemd van typedefinities: als twee types op dezelfde manier worden gedefinieerd, bijvoorbeeld p. 108

```
type Datum = (Int, Int)
type Ratio = (Int, Int)
```

dan kunnen ze door elkaar worden gebruikt. "Datums" kunnen daardoor ineens worden verwerkt alsof het "rationale getallen" zijn, zonder dat dat foutmeldingen van de typechecker oplevert.

Met datadefinities is het mogelijk om echte nieuwe types te maken, zodat bijvoorbeeld een *Ratio* niet meer zonder meer uitwisselbaar is met elke andere (Int, Int). In plaats van de typedefinitie wordt daartoe de volgende datadefinitie gegeven:

data *Ratio* = *Rat* (*Int*, *Int*)

Er is dus slechts één constructorfunctie. Om een breuk te maken met een teller 3 en een noemer 5, is het nu niet meer voldoende om $(3, 5)$ te schrijven, maar moet je schrijven *Rat* $(3, 5)$. Net als bij verenigingstypes kan *Rat* worden beschouwd als conversiefunctie van (*Int*, *Int*) naar *Ratio*. Het is eigenlijk wel zo handig om ook constructorfuncties te curryen. In dat geval krijgen ze niet een tupel als parameter, maar twee losse waarden. De bijbehorende datatypedefinitie is:

data *Ratio* = *Rat Int Int*

Deze methode wordt veel gebruikt om *beschermde types* te maken. Een beschermd type bestaat uit een datadefinitie en een aantal functies die op het gedefinieerde type werken (in het geval van *Ratio* bijvoorbeeld qPlus, qMin, qMaal en qDeel).

De rest van het programma (dat mogelijkerwijs door een andere programmeur geschreven kan zijn) mag van het type gebruik maken via de daarvoor bedoelde functies. Het mag echter geen gebruik maken van de manier waarop het type is opgebouwd. Dat is te bereiken door de naam van de constructorfunctie "geheim te houden". Als later de representatie van rationale getallen om een of andere reden gewijzigd zou moeten worden, hoeven alleen de vier basisfuncties opnieuw geschreven te worden; de rest van het programma blijft gegarandeerd werken.

Als naam voor de constructorfunctie wordt vaak dezelfde naam als de naam van het type gekozen, dus bijvoorbeeld

data *Ratio* = *Ratio Int Int*

Daar is niets op tegen; voor de interpreter is er geen verwarring mogelijk (het woord *Ratio* in een type, bijvoorbeeld achter ::, stelt het type voor; in een expressie is het de constructorfunctie).

6.4 Polymorfe datatypes

Je kunt functies bedenken die niet altijd een antwoord kunnen geven: het opzoeken van een waarde in een tabel (misschien komt de waarde niet voor) of worteltrekken uit een getal (misschien is het getal negatief). In Haskell moet er echter wel altijd iets opgeleverd worden. Het is soms mogelijk om de waarde te coderen in het resultaattype, bijvoorbeeld een wortel uit een negatief getal levert -1 op. Dat heeft echter dezelfde nadelen als eerder het coderen van windrichtingen als getallen. Ook hier kunnen datatypes uitkomst brengen:

data MisschienFloat = Ja *Float*
| Nee

Een waarde van het type MisschienFloat is ofwel Ja toegepast op een *Float* ofwel Nee. Een veilige worteltrekfunctie ziet er nu zo uit:

```
veiligeWortel                :: Float -> MisschienFloat
veiligeWortel x | x >= . 0.0 = Ja (sqrt x)
                | otherwise  = Nee
```

Als we nu een functie willen maken die "misschien een Int" oplevert, bijvoorbeeld een veilige deling van twee gehele getallen, dan hebben we het type MisschienInt nodig. En als we in een tabel iemands adres willen opzoeken dan hebben we MisschienString nodig of zelfs MisschienTupelVanStringEnInt als we straatnaam en huisnummer apart opslaan. Je kunt je voorstellen dat er willekeurig veel Misschien-types zijn. Gelukkig kunnen we het type dat we misschien opleveren abstraheren uit de datatype definitie; we kunnen een datatype maken dat *polymorf* is in het type dat achter Ja staat! Dat doen we door een typevariable te gebruiken in plaats van het concrete type *Float*. Alle typevariabelen die we gebruiken in de definitie van de constructoren verschijnen als parameter aan het type dat we definiëren:

```
data Misschien a = Ja a
                 | Nee
```

De constructorfunctie Ja kunnen we nu toepassen op een waarde van een willekeurig type (zeg T) en daarmee krijgen we een waarde van het type Misschien T. Concreter gezegd, Ja 3.0 is van het type Misschien *Float*, Ja `'a'` is van het type Misschien *Char* en Ja (`"Dorpsstraat"`, 5) is van het type Misschien (*String*, *Int*). Kortom, we hebben één datatype met oneindig veel concrete invullingen. Hier zijn voor de duidelijkheid de types van de twee constructorfuncties:

```
Ja  :: a -> Misschien a
Nee :: Misschien a
```

Misschien-types komen zoveel voor dat in de prelude is een vergelijkbaar type is gedefinieerd:

```
data Maybe a = Nothing
             | Just a
```

Gebruikmakend hiervan is er in de prelude een functie lookupBy gemaakt, die een waarde in een lijst van tupels zoekt, als die gevonden wordt *Just* de andere helft van het tupel oplevert, en als de waarde niet in de lijst zit *Nothing* oplevert:

```
lookupBy          :: (a -> a -> Bool) -> a -> [(a, b)] -> Maybe b
lookupBy _ _ []    = Nothing
lookupBy eq k ((x, y) : xys)
  | k `eq` x       = Just y
  | otherwise      = lookupBy eq k xys
```

6.5 Recursieve datatypes

Met datatypes, zoals we tot nu toe gezien hebben, kunnen we alleen maar eindige waarden maken. Van tevoren staat vast welke constructoren er zijn en hoeveel parameters ze hebben en daarmee ook hoe groot een waarde van dat type is. Voor lijsten in Haskell geldt dat niet; een lijst kan willekeurig lang zijn en dan nog kunnen we er een element voor plakken. Dat komt door de manier waarop lijsten zijn opgebouwd: een lijst is een element op kop van een andere lijst. Je ziet dat deze definitie recursief is; om een lijst te bouwen hebben we een andere lijst nodig. En gelukkig is er de lege lijst [] zodat we er ook nog een keer mee op kunnen houden.

Het zou mooi zijn als we zelf ook datastructuren kunnen maken die willekeurig groot kunnen worden. En dat kan: constructoren kunnen als parameter een waarde krijgen van het type dat we aan het definiëren zijn. Ofwel, een datatype kan recursief zijn. Hier is een voorbeeld:

```
data IntBoom = Tak Int IntBoom IntBoom
             | Blad
```

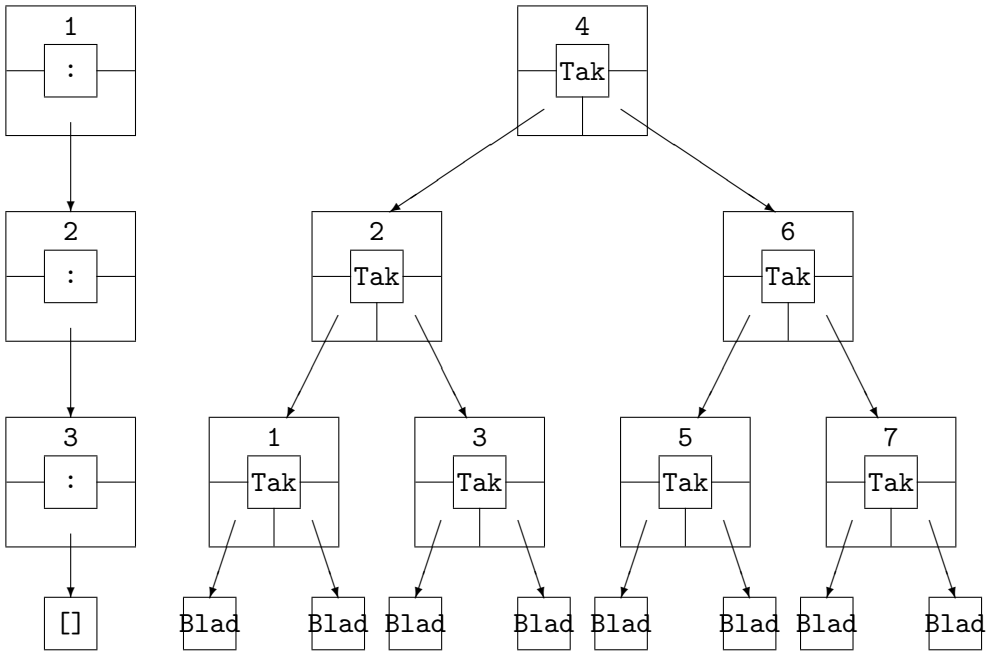
We zien dat het nieuwe type IntBoom twee constructoren heeft. Een Blad is een boom en de constructorfunctie Tak toegepast op drie parameters is ook een boom. De eerste parameter is simpelweg een *Int* en geeft de mogelijkheid om bij een Tak een getal op te slaan. De twee andere parameters zijn van type IntBoom, het type dat we aan het definiëren zijn. Hierdoor kunnen we willekeurig grote bomen bouwen want een Tak bevat een boom die een Tak kan zijn die een boom bevat... En Blad is er zodat we met dit proces kunnen stoppen.

Het IntBoom-type legt het type vast van de waarden die in iedere Tak worden opgeslagen. Nu hebben we net gezien dat datatypes algemener gemaakt kunnen worden door ze polymorf te maken. Laten we daarom in plaats van IntBoom het meer algemene type *Boom* bekijken:

```
data Boom a = Tak a (Boom a) (Boom a)
            | Blad
```

Je kunt deze definitie als volgt uitspreken. "Een boom met elementen van type a (kortweg boom-over-a) kan op twee manieren worden opgebouwd: (1) door de functie Tak toe te passen op drie parameters (één van type a en twee van type boom-over-a), of (2) door de constante Blad te gebruiken." Hier is een voorbeeld van een waarde van dit type:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
             (Tak 3 Blad Blad)
      )
      (Tak 6 (Tak 5 Blad Blad)
             (Tak 7 Blad Blad)
      )
```



lijststructuur *boomstructuur*

Het hoeft niet zo mooi over de regels gespreid te worden; ook toegestaan is:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad) (Tak 3 Blad Blad))
  (Tak 6 (Tak 5 Blad Blad) (Tak 7 Blad Blad))
```

De eerstgenoemde constructie is natuurlijk wel duidelijker. En een figuur (zie blz. 134) maakt de boomstructuur nog veel duidelijker.

Functies op een boom kunnen gedefinieerd worden door voor elke constructorfunctie een patroon te maken. De volgende functie bepaalt bijvoorbeeld het aantal Tak-constructies in een boom:

```
omvang              :: Boom a -> Int
omvang Blad         = 0
omvang (Tak x p q) = 1 + omvang p + omvang q
```

Vergelijk deze functie met de functie length op lijsten.

6.6 Lijsten zijn ook bomen

De ingebouwde lijsten van Haskell zijn ook te zien als "bomen". Bij iedere tak is er dan sprake van één deelboom in plaats van twee. We zouden zelf de lijsten na kunnen maken met het volgende datatype:

```
data Lijst a = OpKop a (Lijst a)
             | Leeg
```

In plaats van $[1, 2, 3]$ wat ook te zien is als $1 : 2 : 3 : [\,]$, schrijf je dan
OpKop 1 (OpKop 2 (OpKop 3 Leeg))). Het symbool $[\,]$ is eigenlijk een heel speciale notatie voor een constructor zonder parameters. En de operator : is een constructor met twee parameters: een waarde en een lijst. We kunnen zelf ook constructorfuncties maken die je als een operator tussen de twee parameters schrijft. De naam van die operator moet beginnen met een dubbele punt en de lijstconstructor : is daar dus een voorbeeld van. Voor de rest mag de naam bestaan uit tekens waar gewone operatoren ook uit bestaan.

Er zijn er nog veel meer variaties van bomen te bedenken:

- Bomen waarbij de informatie alleen in de eindpunten wordt opgeslagen (in plaats van op de splitspunten zoals bij *Boom*):

  ```
  data Boom2 a = Tak2 (Boom2 a) (Boom2 a)
               | Blad2 a
  ```

- Bomen waarbij de informatie van type a in de splitspunten is opgeslagen, en informatie van type b in de eindpunten:

  ```
  data Boom3 a b = Tak3 a (Boom3 a b) (Boom3 a b)
                 | Blad3 b
  ```

- Bomen die zich op elk splitspunt in drieën splitsen in plaats van in tweeën:

  ```
  data Boom4 a = Tak4 a (Boom4 a) (Boom4 a) (Boom4 a)
               | Blad4
  ```

- Bomen waarin het aantal uitgaande takken in een splitspunt variabel is:

  ```
  data Boom5 a = Tak5 a [Boom5 a]
  ```

  In deze boom is geen aparte constructor voor "eindpunt" nodig, omdat daarvoor een splitspunt met nul uitgaande takken gebruikt kan worden.

- Bomen waarin elk splitspunt slechts één uitgaande tak heeft:

  ```
  data Boom6 a = Tak6 a (Boom6 a)
               | Blad6
  ```

  Een "boom" volgens dit type is in feite een lijst: hij heeft een lineaire structuur.

- Bomen met verschillende soorten splitsingen:

  ```
  data Boom7 a b = Tak7a Int a (Boom7 a b) (Boom7 a b)
                 | Tak7b Char (Boom7 a b)
                 | Blad7a b
                 | Blad7b Int
  ```

6.7 Zoekbomen

Een goed voorbeeld van een situatie waarin beter bomen gebruikt kunnen worden dan lijsten, is het zoeken naar (de aanwezigheid van) een waarde in een grote collectie. Daarvoor kunnen *zoekbomen* gebruikt worden.

In paragraaf 4.1.2 werd de functie elem gedefinieerd, die *True* oplevert als een element in een lijst aanwezig is. Of deze functie nu met behulp van de standaardfuncties map en or wordt gedefinieerd, of direct met recursies maakt voor de efficiëntie ervan niet zo veel uit. In beide gevallen worden de elementen van de lijst één voor één geïnspecteerd. Op het moment dat het element gevonden is, geeft de functie direct een resultaat (dankzij lazy evaluatie), maar als het element niet aanwezig is moet de functie alle elementen van de lijst bekijken om tot die conclusie te komen. p. 85

Iets handiger werkt het als de functie mag aannemen dat de te doorzoeken lijst gesorteerd is, dat wil zeggen dat de elementen op stijgende volgorde staan. Het zoekproces kan dan namelijk ook gestopt worden als het gevorderd is tot "voorbij" de gezochte waarde. De prijs is wel dat we in staat moeten zijn om elementen te kunnen ordenen, op gelijkheid testen is niet meer voldoende[1]:

```
elem'                    :: Ord a => a -> [a] -> Bool
elem' e []               = False
elem' e (x : xs) | e < x = False
                 | e == x = True
                 | e > x  = elem' e xs
```

Een veel grotere verbetering is het echter als de elementen niet in een lijst zijn opgeslagen, maar in een *zoekboom*. Een zoekboom is een soort "gesorteerde boom". Het is een boom die is opgebouwd volgens de definitie van *Boom* uit de vorige paragraaf:

```
data Boom a = Tak a (Boom a) (Boom a)
            | Blad
```

Op elk splitspunt is een element opgeslagen, en twee (kleinere) bomen: een linkerdeelboom en een rechterdeelboom (zie de figuur op blz. 134). In een zoekboom wordt nu bovendien geëist dat alle waarden in de linkerdeelboom *kleiner* zijn dan de waarde in het splitspunt, en alle waarden in de rechterdeelboom *groter*. De waarden in de voorbeeldboom in de genoemde figuur zijn zo gekozen, dat de afgebeelde boom inderdaad een zoekboom is.

[1] Alvast vooruitkijkend naar het hoofdstuk over type classes, introduceren we hier in de typesignatuur van de functie elem' het predikaat *Ord a*. Zonder nu in te willen gaan op de details garandeert deze dat bij de instantiatie van het type *a* er alleen een instantie gekozen mag worden voor welke definities van de operatoren <, ==, > (en nog een paar andere) zijn gegeven. Een ander dergelijk predikaat is *Eq a* welke garandeert dat voor *a* de operatoren == and /= zijn gedefineerd. Voor nu is het genoeg om te onthouden dat door het gebruik van de predikaten de gegeven operatoren beschikbaar zijn.

In een zoekboom is het zoeken naar een waarde heel eenvoudig. Als de gezochte waarde gelijk is aan de opgeslagen waarde in een splitspunt: mooi zo. Als de gezochte waarde kleiner is dan de opgeslagen waarde, dan moet doorgezocht worden in de linkerdeelboom (in de rechterdeelboom zitten immers grotere waarden). Andersom, als de gezochte waarde groter is dan de opgeslagen waarde, moet juist in de rechterdeelboom worden doorgezocht. De functie elemBoom is dus als volgt:

```
elemBoom                          :: Ord a => a -> Boom a -> Bool
elemBoom e Blad                   = False
elemBoom e (Tak x li re) | e == x = True
                         | e < x  = elemBoom e li
                         | e > x  = elemBoom e re
```

Als de boom evenwichtig is opgebouwd, zal het te doorzoeken aantal elementen bij elke stap ongeveer halveren. Het gezochte element of een Blad-eindpunt is dan snel gevonden: een verzameling van duizend elementen hoeft maar 10 keer gehalveerd te worden, en een verzameling van een miljoen elementen 20 keer. Vergelijk dat met de gemiddeld half miljoen stappen die de functie elem kost op een verzameling met een miljoen elementen.

In het algemeen kun je zeggen dat het geheel doorzoeken van een verzameling met n elementen met elem n stappen kost, maar met elemBoom slechts ${}^2\log n$ stappen.

Zoekbomen zijn goed te gebruiken als een grote hoeveelheid gegevens vaak moet worden doorzocht. Ook in bijvoorbeeld de functie zoekOp uit paragraaf 4.3.4 is met behulp van zoekbomen een dramatische snelheidswinst te boeken. p. 112

Opbouw van een zoekboom

De vorm van een zoekboom voor een bepaalde collectie gegevens kan "met de hand" bepaald worden. De zoekboom kan vervolgens worden ingetikt als grote expressie met veel constructorfuncties. Dat is echter een vervelend werk, dat eenvoudig kan worden geautomatiseerd.

Zoals de functie insert een element op de juiste plaats toevoegt aan een gesorteerde lijst (zie paragraaf 4.1.5), voegt de functie insertBoom een element toe aan een zoekboom. Het resultaat blijft een zoekboom, dat wil zeggen het element wordt op de juiste plaats ingevoegd: p. 92

```
insertBoom                          :: Ord a => a -> Boom a -> Boom a
insertBoom e Blad                   = Tak e Blad Blad
insertBoom e (Tak x li re) | e <= x = Tak x (insertBoom e li) re
                           | e > x  = Tak x li (insertBoom e re)
```

In het geval dat het element wordt toegevoegd aan Blad (een "lege" boom), wordt een klein boompje gebouwd uit e en twee lege boompjes. Anders is de boom niet leeg, en

bevat dus een opgeslagen waarde x. Deze waarde wordt gebruikt om te beslissen of e in de linker- of rechterdeelboom ingevoegd moet worden.

Door de functie insertBoom herhaald te gebruiken, kunnen alle elementen van een lijst in een zoekboom worden gezet:

```
lijstNaarBoom :: Ord a => [a] -> Boom a
lijstNaarBoom = foldr insertBoom Blad
```

Vergelijk deze functie met de functie isort in paragraaf 4.1.5. p. 92

Het gebruik van lijstNaarBoom heeft het nadeel dat de zoekboom die het resultaat is niet altijd evenwichtig is. Bij gegevens die in een willekeurige volgorde worden ingevoegd valt dat meestal wel mee. Als de lijst die tot boom wordt gemaakt echter al gesorteerd is, is het resultaat een "scheefgegroeide" boom:

```
? lijstNaarBoom [1..7]
Tak 7 (Tak 6 (Tak 5 (Tak 4 (Tak 3 (Tak 2 (Tak 1 Blad Blad)
Blad)Blad)Blad) Blad) Blad) Blad
```

Dit is weliswaar een zoekboom (elke waarde ligt tussen de waardes in de linker- en de rechter zoekboom), maar is helemaal scheefgetrokken zodat een bijna lineaire structuur is ontstaan. De gewenste logaritmische zoektijden zijn in deze boom dan ook niet mogelijk. Een betere (niet-scheve) boom met dezelfde waarden zou zijn:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
             (Tak 3 Blad Blad))
      (Tak 6 (Tak 5 Blad Blad)
             (Tak 7 Blad Blad))
```

Sorteren met zoekbomen

De hierboven ontwikkelde functies kunnen worden gebruikt in een nieuw sorteeralgoritme. Daarbij is nog één extra functie nodig: een functie die de elementen van een zoekboom op volgorde in een lijst zet. Deze functie is als volgt:

```
labels                :: Boom a -> [a]
labels Blad           = []
labels (Tak x li re) = labels li ++ [x] ++ labels re
```

In tegenstelling tot insertBoom doet deze functie een recursieve aanroep op de linkerdeelboom èn de rechterdeelboom. Op deze manier wordt elk element in de complete boom bekeken. Doordat de waarde x er op de juiste plaats tussen wordt geplakt, is het resultaat een gesorteerde lijst (mits de parameter een zoekboom is).

Een willekeurige lijst kan nu gesorteerd worden door er een zoekboom van te maken met lijstNaarBoom, en de elementen vervolgens op volgorde op te sommen met labels:

```
sorteer :: Ord a => [a] -> [a]
sorteer = labels . lijstNaarBoom
```

Weglaten uit zoekbomen

Een zoekboom kan als database gebruikt worden. Naast de operaties opsommen, invoegen en opbouwen, waarvoor al functies geschreven zijn, zou daarbij een functie voor het weglaten van een te specificeren element goed van pas komen[2]. Deze functie lijkt een beetje op de functie insertBoom; de functie wordt al naar gelang de aangetroffen waarde recursief aangeroepen op de linker- of de rechterdeelboom.

```
deleteBoom          :: Ord a => a -> Boom a -> Boom a
deleteBoom e Blad = Blad
deleteBoom e (Tak x li re)
   | e < x          = Tak x (deleteBoom e li) re
   | e == x         = samenvoegen li re
   | e > x          = Tak x li (deleteBoom e re)
```

Als de waarde echter in de boom aangetroffen wordt (het geval e == x), kan hij niet zomaar worden weggelaten zonder een "gat" achter te laten. Daarom is er een functie samenvoegen nodig, die twee zoekbomen samenvoegt. Deze functie werkt door het grootste element uit de linker deelboom te gebruiken als nieuw splitspunt. Als de linkerdeelboom leeg is, is samenvoegen natuurlijk ook geen probleem:

```
samenvoegen          :: Boom a -> Boom a -> Boom a
samenvoegen Blad b2 = b2
samenvoegen b1 b2   = Tak x b1' b2
  where
    (x, b1') = grootsteUit b1
```

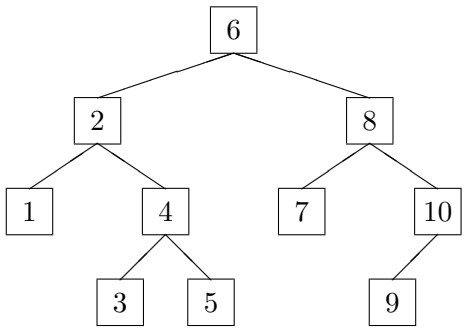
De functie grootsteUit levert behalve het grootste element van een boom ook de boom op die ontstaat door dit grootste element te verwijderen. Deze twee resultaten worden in een tupel samengevoegd. Het grootste element kun je vinden door steeds in de rechterdeelboom af te dalen:

```
grootsteUit                  :: Boom a -> (a, Boom a)
grootsteUit (Tak x b1 Blad) = (x, b1)
grootsteUit (Tak x b1 b2)   = (y, Tak x b1 b2')
  where
    (y, b2') = grootsteUit b2
```

[2]We geven de voorkeur aan het woord "weglaten" boven "verwijderen", omdat deleteBoom de argumentboom niet verandert door iets te verwijderen, maar een nieuwe boom opbouwt waaruit het gekozen element is weggelaten.

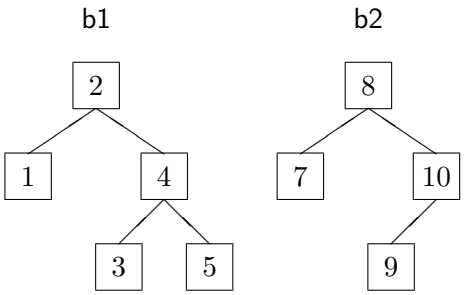
Om de werking van deleteBoom te demonstreren bekijken we een voorbeeld, waarbij we voor de duidelijkheid de bomen grafisch voorstellen. Bij de aanroep van

deleteBoom 6



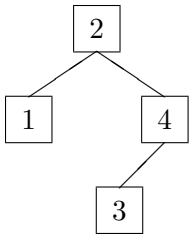
wordt de functie samenvoegen aangeroepen met de linker- en de rechterdeelboom als parameter:

samenvoegen



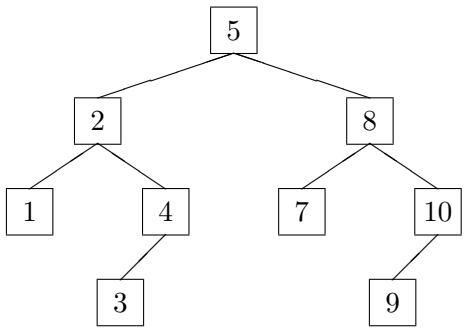
Door samenvoegen wordt de functie grootsteUit aangeroepen met b1 als parameter. Dat levert een tweetupel (x, b1′) op:

(5 ,



)

De bomen b1′ en b2 worden als linker- en rechter deelboom gebruikt in een nieuwe zoekboom:



Omdat de functie grootsteUit alleen maar wordt aangeroepen vanuit samenvoegen, hoeft hij niet gedefinieerd te worden op een Blad-boom. Hij wordt immers alleen maar met niet-lege bomen aangeroepen, omdat de lege boom in de functie samenvoegen al apart wordt afgehandeld.

Opgaven

6.1 Schrijf een zoekboomversie van de functie zoekOp, zoals elemBoom een zoekboomversie is van elem. Geef ook het type van de functie.

6.2 De functie map kan op functies worden toegepast. Het resultaat is ook weer een functie (met een ander type). Er is geen enkele voorwaarde verbonden aan het soort functies waarop `map` toegepast kan worden. Je kunt hem dus ook op de functie `map` zelf toepassen! Wat is het type van de expressie map map ?

6.3 Geef een definitie van until die gebruik maakt van iterate en dropWhile.

6.4 Geef een directe definitie van de operator $<$ op lijsten. Deze definitie mag dus geen gebruik maken van operatoren zoals $<=$ op lijsten. (Als je deze definitie daadwerkelijk met `ghci` wilt uitproberen, gebruik dan een andere naam dan $<$, omdat de operator $<$ al in de prelude wordt gedefinieerd.)

6.5 Schrijf de functie length als aanroep van foldr. (Hint: begin aan de rechterkant met het getal 0, en zorg ervoor dat de operator die achtereenvolgens op alle elementen van de lijst wordt toegepast steeds 1 bij het tussenresultaat optelt, ongeacht de waarde van het lijstelement.) Wat is het type van de functie die daarbij aan foldr wordt meegegeven?

6.6 In paragraaf 4.1.5 werden twee sorteermethodes genoemd: de op insert gebaseerde functie isort, en de op merge gebaseerde functie msort. Een andere sorteermethode werkt volgens het volgende principe. Bekijk het eerste element van de te sorteren lijst. Neem nu alle elementen van de lijst die kleiner zijn dan deze waarde. In het eindresultaat moeten al deze waarden vóór het eerste element komen. Ze moeten wel eerst (met een recursieve aanroep) gesorteerd worden. De waarden uit de lijst p. 92

die juist groter zijn dan het eerste element moeten (gesorteerd) erachter komen. (Dit algoritme staat bekend onder de naam *quicksort*.)
Schrijf een functie die volgens dit principe werkt. Bedenk zelf wat het basisgeval is. Wat is het essentiële verschil tussen deze functie en msort?

6.7 Beschouw de functie groepeer met het volgende type:

```
groepeer :: Int -> [a] -> [[a]]
```

Deze functie deelt een gegeven lijst in deellijsten (die in een lijst van lijsten worden opgeleverd), waarbij de deellijsten een gegeven lengte hebben. Alleen de laatste deellijst mag zonodig wat korter zijn. De functie kan als bijvoorbeeld als volgt gebruikt worden:

```
? groepeer 3 [1 .. 11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```

Schrijf deze functie volgens hetzelfde principe als de functie intString in paragraaf 4.2.6, dat wil zeggen door samenstelling van een aantal partiële parametrisaties van iterate, takeWhile en map. p. 101

6.8 Bekijk bomen van het volgende type:

```
data Boom2 a = Blad2 a
             | Tak2 (Boom2 a) (Boom2 a)
```

Schrijf een functie mapBoom en een functie foldBoom die op een Boom2 werken, naar analogie van de functies map en foldr op lijsten. Geef ook het type van deze functies.

6.9 Schrijf een functie diepte, die oplevert uit hoeveel nivo's een Boom2 bestaat. Geef een definitie met inductie, en een alternatieve definitie waarin je mapBoom en foldBoom gebruikt.

6.10 Schrijf een functie toonBoom, die een aantrekkelijk representatie als string van een boom zoals gedefinieerd op blz. 133 geeft. In de string moet elk blad op een aparte regel komen te staan (gescheiden door `"\n"`); bladeren op een dieper nivo moeten meer zijn ingesprongen dan bladeren op een minder diep nivo.

6.11 Stel dat een boom b diepte n heeft. Wat is het minimale en het maximale aantal bladeren dat b kan bevatten?

6.12 Schrijf een functie die gegeven een gesorteerde lijst een zoekboom oplevert (dus een boom die als je er labels op toepast een gesorteerde lijst oplevert). Zorg ervoor dat de boom niet "scheefgroeit", zoals het geval is als je de functie lijstNaarBoom zou gebruiken.

6.13 Schrijf functies die de waarden van type a in de knopen van een boom van type:

$$\begin{aligned}&\textbf{data}\ \mathit{Boom}\ a = \mathsf{Knoop}\ (\mathit{Boom}\ a)\ a\ (\mathit{Boom}\ a)\\ &\qquad\qquad\quad \mid \mathsf{Blad}\end{aligned}$$

in depth-first (pre-orde), infix (in-orde) en breadth-first order oplevert.

6.14 Schrijf een functie die alle paden van type $[a]$ oplevert van de wortel tot een blad in een boom van type $\mathit{Boom}\ a$.

6.15 Schrijf een functie die van een boom zoals in de vorige opgave een lijst oplevert van knopen die zich op een van de langste paden van de wortel tot een blad bevinden. Probeer de oplossing lineair in de omvang van de boom te houden.

7 Case study: lijstalgoritmen

Verschillende hoofdstukken in dit dictaat kennen een titel met daarin de tekst "case study". Deze hoofdstukken zijn bedoeld om naast de concepten die in de andere hoofdstukken worden geintroduceerd ook een idee te geven hoe je bepaalde veelvoorkomende zaken in Haskell implementeert. In dit hoofdstuk staan lijsten centraal: in sectie 7.1 gaat p. 144 het om combinatorische functies die uit een gegeven lijst een aantal lijsten construeert met een specifieke eigenschap. Vervolgens behandelen we in dit hoofdstuk ook operaties op matrices (lijsten van lijsten), en rekenen met polynomen. In hoofdstuk 8 laten we p. 172 zien hoe je verscheidene soorten expressies door middel van Haskell datatypes representeert, en hoe je daar dan operaties over kunt definiëren. In hoofdstuk 9 behandelen we p. 179 het efficiënter coderen van informatie met behulp van de zogenoemde Huffman-codering. Hoewel de case study hoofdstukken weinig nieuws introduceren als het om Haskell gaat, vormen ze wel een goede bron van tentamenopgaven. En tijdens de ontwikkeling van de code in deze hoofdstukken proberen we ook zoveel mogelijk aan te geven, hoe (in algemenere zin) de code vorm wordt gegeven.

7.1 Combinatorische functies

7.1.1 Segmenten en deelrijen

Combinatorische functies werken op een lijst. Ze leveren een lijst van lijsten op, waarbij geen gebruik gemaakt wordt van specifieke eigenschappen van de elementen van de lijst. Het enige wat combinatorische functies kunnen doen, is elementen weglaten, elementen verwisselen, of elementen tellen.

In deze en de volgende paragraaf worden een aantal combinatorische functies gedefinieerd. Omdat ze geen gebruik maken van eigenschappen van de elementen van hun parameterlijst, zijn het polymorfe functies:

```
inits, tails, segs :: [a] -> [[a]]
subs, perms    ::     [a] -> [[a]]
combs          :: Int -> [a] -> [[a]]
```

Om de werking van deze functies te illustreren volgen hieronder de uitkomsten van deze functies toegepast op de lijst $[1, 2, 3, 4]$

inits	tails	segs	subs	perms	combs 2	combs 3
[]	[1,2,3,4]	[]	[]	[1,2,3,4]	[1,2]	[1,2,3]
[1]	[2,3,4]	[4]	[4]	[2,1,3,4]	[1,3]	[1,2,4]
[1,2]	[3,4]	[3]	[3]	[2,3,1,4]	[1,4]	[1,3,4]
[1,2,3]	[4]	[3,4]	[3,4]	[2,3,4,1]	[2,3]	[2,3,4]
[1,2,3,4]	[]	[2]	[2]	[1,3,2,4]	[2,4]	
		[2,3]	[2,4]	[3,1,2,4]	[3,4]	
		[2,3,4]	[2,3]	[3,2,1,4]		
		[1]	[2,3,4]	[3,2,4,1]		
		[1,2]	[1]	[1,3,4,2]		
		[1,2,3]	[1,4]	[3,1,4,2]		
		[1,2,3,4]	[1,3]	[3,4,1,2]		
			[1,3,4]	[3,4,2,1]		
			[1,2]	[1,2,4,3]		
			[1,2,4]	[2,1,4,3]		
			[1,2,3]	[2,4,1,3]		
			[1,2,3,4]	[2,4,3,1]		
				[1,4,2,3]		
				(7 andere)		

Zoals uit de voorbeelden waarschijnlijk al duidelijk is, is de betekenis van deze zes functies als volgt:

- inits levert alle *beginsegmenten* van een lijst, dat wil zeggen aaneengesloten stukken van de lijst die aan het begin beginnen. De lege lijst telt ook als beginsegment.
- tails levert alle *eindsegmenten* van een lijst: aaneengesloten stukken die tot het eind doorlopen. Ook de lege lijst is een eindsegment.
- segs levert *alle segmenten* van een lijst: beginsegmenten en eindsegmenten, maar ook aaneengesloten stukken uit het midden.
- subs levert alle *subsequences* (deelrijen) van een lijst. In tegenstelling tot segmenten hoeven de elementen van een deelrij in de origninele lijst niet aaneengesloten te zijn. Er zijn dus meer deelrijen dan segmenten.
- perms levert alle *permutaties* van een lijst. Een permutatie van een lijst bevat dezelfde elementen, maar mogelijk in een andere volgorde.
- combs n levert alle *combinaties van* n *elementen*, dus alle manieren om n elementen te kiezen uit een lijst. De volgorde is daarbij hetzelfde als in de originele lijst.

Deze combinatorische functies kunnen recursief worden gedefinieerd. In de definitie worden dus steeds de gevallen [] en (x : xs) apart behandeld. In het geval (x : xs) wordt de functie recursief aangeroepen op de lijst xs.

Er is een handige manier om op een idee te komen van de definitie van een functie f. Kijk bij een voorbeeldlijst (x : xs) wat het resultaat is van de recursieve aanroep f xs, en probeer het resultaat aan te vullen tot de uitkomst van f (x : xs).

inits

Bij de beschrijving van beginsegmenten hierboven is ervoor gekozen om de lege lijst ook als beginsegment te laten tellen. De lege lijst heeft dus één beginsegment: de lege lijst zelf. De definitie van inits voor het geval "lege lijst" is daarom als volgt:

$$\mathsf{inits}\ [\,] = [[\,]]$$

Voor het geval (x : xs) kijken we naar de gewenste uitkomsten voor de lijst $[1, 2, 3, 4]$.

```
inits [1,2,3,4]  =  [ [] , [1] , [1,2] , [1,2,3] , [1,2,3,4] ]
inits [2,3,4]    =  [ [] , [2] , [2,3] , [2,3,4] ]
```

Hieruit blijkt dat de tweede t/m vijfde elementen van inits $[1, 2, 3, 4]$ overeenkomen met de elementen van inits $[2, 3, 4]$, alleen steeds met een extra 1 op kop. Deze vier lijsten moeten dan nog worden aangevuld met een lege lijst.

Dit mechanisme wordt algemeen beschreven in de tweede regel van de definitie van inits:

```
inits []       = [[]]
inits (x : xs) = [] : map (x:) (inits xs)
```

tails

Net als bij inits heeft de lege lijst één eindsegment: de lege lijst. Het resultaat van tails [] is dus een lijst met als enige element de lege lijst.

Om op een idee te komen voor de definitie van tails (x : xs) kijken we eerst weer naar het voorbeeld $[1, 2, 3, 4]$:

```
tails [1,2,3,4]  =  [ [1,2,3,4] , [2,3,4] , [3,4] , [4] , [] ]
tails [2,3,4]    =  [ [2,3,4] , [3,4] , [4] , [] ]
```

Bij deze functie zijn het tweede t/m vijfde element dus precies gelijk aan de elementen van de recursieve aanroep. Het enige wat moet gebeuren is uitbreiding met een eerste element ([1,2,3,4] in het voorbeeld).

Gebruikmakend van dit idee kan de complete definitie geschreven worden:

```
tails []       = [[]]
tails (x : xs) = (x : xs) : tails xs
```

Het tweede paar haakjes op de tweede regel is essentieel: zonder die haakjes zou de typering incorrect zijn, omdat de operator : dan naar rechts associeert.

segs

Het enige segment van de lege lijst is weer de lege lijst. De uitkomst van segs [] is dus, net als bij inits en tails, een lijst met daarin alleen een lege lijst, [[]].

Om op het spoor van de definitie van segs (x : xs) te komen, passen we de beproefde methode weer toe:

```
segs [1,2,3,4]  =  [ [] , [4] , [3] , [3,4] , [2] , [2,3] , [2,3,4] , [1] , [1,2] , [1,2,3] , [1,2,3,4] ]
segs [2,3,4]    =  [ [] , [4] , [3] , [3,4] , [2] , [2,3] , [2,3,4] ]
```

Als je ze maar op de goede volgorde zet, blijkt dat de eerste zeven elementen van het gewenste resultaat precies overeenkomen met de recursieve aanroep. In het tweede deel

van het resultaat (de lijsten die met een 1 beginnen) zijn de beginsegmenten van [1,2,3,4] te herkennen (alleen de lege lijst is daarbij weggelaten, want die zit al in het resultaat).

Als definitie van segs kan dus genomen worden:

```
segs []       = [[]]
segs (x : xs) = segs xs ++ tail (inits (x : xs))
```

Een andere manier om de lege lijst uit de inits te verwijderen is:

```
segs []       = [[]]
segs (x : xs) = segs xs ++ map (x:) (inits xs)
```

subs

De lege lijst is de enige deelrij van de lege lijst. Voor de definitie van subs (x : xs) kijken we weer naar het voorbeeld:

```
subs [1,2,3,4]  =  [ [1,2,3,4] , [1,2,3] , [1,2,4] , [1,2] , [1,3,4] , [1,3] , [1,4] , [1]
                   , [2,3,4] , [2,3] , [2,4] , [2] , [3,4] , [3] , [4] , [] ]
subs [2,3,4]    =  [ [2,3,4] , [2,3] , [2,4] , [2] , [3,4] , [3] , [4] , [] ]
```

Het aantal elementen van subs (x : xs) (16 in het voorbeeld) is precies twee keer zo groot als het aantal elementen van de recursieve aanroep subs xs. De tweede helft van het totaal resultaat is precies gelijk aan het resultaat van de recursieve aanroep. Ook in de eerste helft zijn deze 8 lijsten weer te herkennen, alleen staat er daar steeds een 1 op kop.

De definitie kan dus luiden:

```
subs []       = [[]]
subs (x : xs) = map (x:) (subs xs) ++ subs xs
```

De functie wordt tweemaal recursief aangeroepen met dezelfde parameter. Dat is zonde van het werk: beter kan de aanroep maar éénmaal gedaan worden, waarna het resultaat tweemaal gebruikt wordt. Dit levert veel tijdwinst op, want ook voor het bepalen van subs xs wordt de functie weer tweemaal recursief aangeroepen, en in die recursieve aanroepen weer... Een veel efficiëntere definitie is dus:

```
subs []       = [[]]
subs (x : xs) = map (x:) subsxs ++ subsxs
  where
    subsxs = subs xs
```

7.1.2 Permutaties en combinaties

perms

Een *permutatie* van een lijst is een lijst met dezelfde elementen, maar mogelijk in een

andere volgorde. De lijst van alle permutaties van een lijst kan goed met een recursieve functie worden gedefinieerd.

De lege lijst heeft één permutatie: de lege lijst. Alle 0 elementen zitten daar namelijk in, en in dezelfde volgorde...

Het interessante geval is natuurlijk de niet-lege lijst (x : xs). We kijken eerst weer naar een voorbeeld:

```
perms [1,2,3,4] = [ [1,2,3,4] , [2,1,3,4] , [2,3,1,4] , [2,3,4,1]
                  , [1,3,2,4] , [3,1,2,4] , [3,2,1,4] , [3,2,4,1]
                  , [1,3,4,2] , [3,1,4,2] , [3,4,1,2] , [3,4,2,1]
                  , [1,2,4,3] , [2,1,4,3] , [2,4,1,3] , [2,4,3,1]
                  , [1,4,2,3] , [4,1,2,3] , [4,2,1,3] , [4,2,3,1]
                  , [1,4,3,2] , [4,1,3,2] , [4,3,1,2] , [4,3,2,1] ]
perms [2,3,4]   = [ [2,3,4] , [3,2,4] , [3,4,2] , [2,4,3] , [4,2,3] , [4,3,2] ]
```

Het aantal permutaties loopt flink op: van een lijst met vier elementen zijn er viermaal zoveel permutaties als van een lijst met drie elementen. In dit voorbeeld is het wat moeilijker om het resultaat van de recursieve aanroep te herkennen. Dit lukt pas door de 24 elementen in 6 groepjes van 4 te verdelen. In elke groepje zitten lijsten met dezelfde waarden als die van één lijst van de recursieve aanroep. Het nieuwe element wordt daar op alle mogelijke manieren tussen gezet.

Bijvoorbeeld, het derde groepje [[1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1]] bevat steeds de elementen [3,4,2], waarbij het element 1 is toegevoegd respectievelijk aan het begin, op de tweede plaats, op de derde plaats, en aan het eind.

Voor het op alle manieren tussenvoegen van één element in een lijst kan een hulpfunctie worden geschreven, die ook weer recursief gedefinieerd is:

```
tussen            :: a -> [a] -> [[a]]
tussen e []       = [[e]]
tussen e (y : ys) = (e : y : ys) : map (y:) (tussen e ys)
```

In de definitie van perms (x : xs) wordt deze functie, partiëel geparametriseerd met x, toegepast op alle elementen van het resultaat van de recursieve aanroep. In het voorbeeld levert dat een lijst met zes lijsten van vier lijstjes. De bedoeling is echter dat er één lange lijst van 24 lijstjes uitkomt. Op de lijst van lijsten van lijstjes moet dus nog de functie concat worden toegepast, die immers dat effect heeft.

Al met al wordt de functie perms als volgt gedefinieerd:

```
perms []       = [[]]
perms (x : xs) = concat (map (tussen x) (perms xs))
  where
    tussen e []       = [[e]]
    tussen e (y : ys) = (e : y : ys) : map (y:) (tussen e ys)
```

combs

Een laatste voorbeeld van een combinatorische functie is de functie combs. Deze functie heeft, behalve de lijst, ook een getal als parameter:

```
combs :: Int -> [a] -> [[a]]
```

De bedoeling is dat in het resultaat van combs n xs alle deelrijen van xs met lengte n zitten. De functie kan dus eenvoudigweg gedefinieerd worden door

```
combs n xs = filter goed (subs xs)
  where
    goed xs = length xs == n
```

Deze definitie is echter niet zo erg efficiënt. Het aantal deelrijen is namelijk meestal erg groot, dus subs kost veel tijd, terwijl de meeste deelrijen door filter weer worden weggegooid. Een betere definitie is te verkrijgen door combs direct te definiëren, zonder subs te gebruiken.

In de definitie van combs worden voor de integerparameter de gevallen 0 en n onderscheiden. In het geval n worden ook voor de lijstparameter twee gevallen onderscheiden. De definitie krijgt dus de volgende vorm:

```
combs 0 xs =  ...
combs n []      = ...
combs n (x : xs) = ...
```

Deze drie gevallen worden hieronder apart bekeken.

- Voor het kiezen van nul elementen uit een lijst is er één mogelijkheid: de lege lijst. Het resultaat van combs 0 xs is daarom weer [[]]. Het maakt daarbij niet uit of xs leeg is of niet.
- Het patroon n is hier te lezen als "1 of meer". Het kiezen van minstens één element uit de lege lijst is onmogelijk. Het resultaat van combs n [] is dan ook de lege lijst: er is geen oplossing mogelijk. Let wel: hier is dus sprake van een *lege lijst oplossingen* en niet van *de lege lijst als enige oplossing* zoals in het vorige geval. Een belangrijk verschil!
- Het kiezen van n elementen uit de lijst x : xs is wel mogelijk, mits het kiezen van n − 1 elementen uit xs mogelijk is. De oplossingen zijn te verdelen in twee groepen: lijstjes waar x in zit, en lijstjes waar x niet in zit.
  - Voor de lijstjes waar x wel in zit, moeten uit de overige elementen xs nog n elementen gekozen worden. Daarin moet dan steeds x op kop gezet worden.
  - Voor de lijstjes waar x niet in zit, moeten alle n elementen uit xs gekozen worden.

  Voor beide gevallen kan combs recursief aangeroepen worden. De resultaten kunnen gecombineerd worden met ++.

De definitie van combs komt er dus als volgt uit te zien:

```
combs 0 xs       = [[]]
combs n []       = []
combs n (x : xs) = map (x:) (combs (n − 1) xs) ++ combs n xs
```

Bij deze functie kunnen de twee recursieve aanroepen niet gecombineerd worden, zoals bij subs. De twee aanroepen hebben hier namelijk verschillende parameters.

7.1.3 De @-notatie

De definitie van tails heeft iets omslachtigs:

```
tails []       = [[]]
tails (x : xs) = (x : xs) : tails xs
```

In de tweede regel wordt de parameterlijst door het patroon gesplitst in een kop x en een staart xs. De staart wordt gebruikt bij de recursieve aanroep, maar de kop en de staart worden ook weer samengevoegd tot de lijst (x : xs). Dat is zonde van het werk, want deze lijst is in feite ook al beschikbaar als parameter.

Een andere definitie van tails zou kunnen luiden:

```
tails [] = [[]]
tails xs = xs : tails (tail xs)
```

Nu is het opnieuw opbouwen van de parameterlijst niet nodig, omdat hij helemaal niet gesplitst wordt. Maar nu moet, om de staart bij de recursieve aanroep te kunnen meegeven, expliciet de functie tail worden gebruikt. Het leuke van patronen was nu juist, dat dat niet nodig is.

Ideaal zou het zijn om het goede van deze twee definities te combineren. De parameter moet dus zowel als geheel beschikbaar zijn, als gesplitst in een kop en een staart. Voor deze situatie is een speciale notatie beschikbaar. Vóór een patroon mag een naam worden geschreven die het geheel aanduidt. De naam wordt van het patroon gescheiden door het symbool @.

Met gebruik van deze constructie wordt de definitie van tails als volgt:

```
tails []             = [[]]
tails lyst@(x : xs) = lyst : tails xs
```

Hoewel het symbool @ in operatorsymbolen gebruikt mag worden, is een losse @ speciaal gereserveerd voor deze constructie.

Bij functies die al eerder in dit dictaat gedefinieerd werden, komt een @-patroon ook goed van pas. Bijvoorbeeld in de functie dropWhile:

```
dropWhile p []       = []
dropWhile p ys@(x : xs)
  | p x              = dropWhile p xs
  | otherwise        = ys
```

7.2 Matrixrekening

7.2.1 Vectoren en matrices

Matrixrekening is een tak van wiskunde die zich bezighoudt met lineaire afbeeldingen in meerdimensionale ruimtes. In deze paragraaf worden de belangrijkste begrippen uit de matrixrekening ingevoerd. Verder wordt aangegeven hoe deze begrippen in Haskell als type of functie gemodelleerd kunnen worden. De Haskell-definitie van de functies volgt in de volgende twee paragrafen.

De generalisatie van de eendimensionale lijn, het tweedimensionale platte vlak en de driedimensionale ruimte is de n-dimensionale ruimte. In een n-dimensionale ruimte kan elk "punt" aangeduid worden door n getallen. Zo'n aanduiding wordt ook wel een *vector* genoemd. In Haskell zou een vector gerepresenteerd kunnen worden als element van het volgende type:

```
type Vector = [Float]
```

Het aantal elementen in de lijst die een Vector voorstelt bepaalt de dimensie van de ruimte. Om geen verwarring te krijgen met andere lijsten-van-floats, die geen vectoren voorstellen, is het beter om een "beschermd type" te definiëren zoals beschreven in paragraaf 6.3: p. 130

```
data Vector = Vec [Float]
```

Op een lijst getallen moet dus de constructorfunctie Vec worden toegepast om er een Vector van te maken.

In plaats van als *punt* in de (n-dimensionale) ruimte kan een vector ook worden beschouwd als *(gericht) lijnstuk* van de oorsprong naar dat punt. Een nuttige functie bij het werken met vectoren is het bepalen van de afstand van een punt tot de oorsprong, of, in de lijnstukinterpretatie, de *lengte* van het lijnstuk:

```
vecLengte :: Vector -> Float
```

Bij twee vectoren (in dezelfde ruimte) kan bepaald worden of ze *loodrecht* op elkaar staan, en meer algemeen welke *hoek* ze met elkaar maken:

```
vecLoodrecht :: Vector -> Vector -> Bool
vecHoek      :: Vector -> Vector -> Float
```

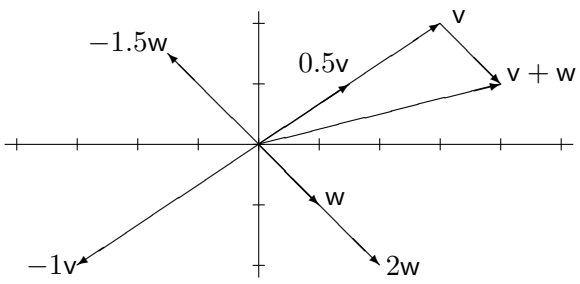
Verder kan een vector met een getal worden "vermenigvuldigd" door alle getallen in de lijst (alle coördinaten) met dat getal te vermenigvuldigen. Twee vectoren worden "opgeteld" door alle coördinaten op te tellen:

```
vecScale :: Float -> Vector -> Vector
vecPlus  :: Vector -> Vector -> Vector
```

In de volgende paragraaf zullen deze functies geschreven worden. In de gericht-lijnstuk-interpretatie van vectoren hebben deze functies de volgende meetkundige interpretatie:

- vecScale: de vector blijft in dezelfde richting wijzen, maar wordt "verlengd" met een factor volgens het aangegeven getal. Als de absolute waarde van dit getal < 1 is wordt de vector verkort; als het getal < 0 is gaat de vector de andere kant op wijzen.
- vecPlus: de twee vectoren worden "kop aan staart" gelegd, en wijzen zo een nieuw punt aan (de volgorde maakt daarbij niet uit).

Als voorbeeld bekijken we een paar vectoren in de 2-dimensionale ruimte:



v	=	$[3, 2]$
w	=	$[1, -1]$
vecPlus v w	=	$[4, 1]$
vecScale 0 v	=	$[0, 0]$
vecScale 0.5 v	=	$[1.5, 1]$
vecScale $-$ 1 v	=	$[-3, -2]$
vecScale 2 w	=	$[2, -2]$
vecScale $-$ 1.5 w	=	$[-1.5, 1.5]$

Functies van vectoren naar vectoren heten *afbeeldingen*. Van bijzonder belang zijn *lineaire* afbeeldingen. Dat zijn afbeeldingen waarbij elke coördinaat van de beeldvector een lineaire combinatie is van coördinaten van het origineel.

In de 2-dimensionale ruimte kan elke lineaire afbeelding geschreven worden als

$$f\ (x, y)\ =\ (\ a * x + b * y\ ,\ c * x + d * y\ )$$

waarbij a, b, c en d vrij gekozen kunnen worden. De n^2 getallen die een lineaire afbeelding in de n-dimensionale ruimte beschrijven, worden in de wiskunde als rechthoekig blok getallen met haken eromheen geschreven. Bijvoorbeeld:

$$\begin{pmatrix} \frac{1}{2}\sqrt{3} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2}\sqrt{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

beschrijft een lineaire afbeelding in de 3-dimensionale ruimte (rotatie over 30° om de z-as). Zo'n blok getallen heet een *matrix* (meervoud: *matrices*).

In Haskell kan een matrix gerepresenteerd worden door een lijst van lijsten. We maken er maar meteen een beschermd (data)type van:

```
data Matrix = Mat [[Float]]
```

Daarbij moet gekozen worden of de lijsten de rijen of de kolommen van de matrix voorstellen. In dit diktaat is voor de rijen gekozen (dat is nu de meest logische keuze, omdat elke rij met één vergelijking in de lineaire afbeelding overeenkomt).

Mocht de kolomrepresentatie nodig zijn, dan is er een functie die de rijen van een matrix tot kolommen maakt en andersom. Dit heet *transponeren* van een matrix. De functie die dat doet heeft als type:

```
matTransp :: Matrix -> Matrix
```

Er geldt dus bijvoorbeeld

$$\mathsf{matTransp}\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

De belangrijkste functie die op matrices werkt, is de functie die de lineaire afbeelding uitvoert. Dit wordt wel het *toepassen* van een matrix op een vector genoemd. Het type van deze functie is:

```
matApply :: Matrix -> Vector -> Vector
```

De samenstelling van twee lineaire afbeeldingen is weer een lineaire afbeelding. Bij twee matrices horen twee lineaire afbeeldingen; de samenstelling van deze afbeeldingen wordt weer beschreven door een matrix. Het bepalen van deze matrix wordt het *vermenigvuldigen* van matrices genoemd. Er is dus een functie:

```
matProd :: Matrix -> Matrix -> Matrix
```

Net als functiesamenstelling (de operator (.)) is de functie matProd associatief (dus $A \times (B \times C) = (A \times B) \times C$). Matrixvermenigvuldiging is echter niet commutatief ($A \times B$ is niet altijd gelijk aan $B \times A$). Het matrixproduct $A \times B$ is de afbeelding die eerst matrix B toepast, en dan matrix A.

De identiteitsafbeelding is ook een lineaire afbeelding. Hij wordt beschreven door de *identiteitsmatrix*. Dit is een matrix waarin het getal 1 staat op de diagonaal, en het getal 0 op de andere plaatsen. Voor elke dimensie is er zo'n identiteitsmatrix, die wordt bepaald door de volgende functie:

```
matId :: Int -> Matrix
```

Sommige lineaire afbeeldingen zijn *inverteerbaar*. De inverse afbeelding (als die bestaat) is ook lineair, en wordt dus beschreven door een matrix:

```
matInv :: Matrix -> Matrix
```

De dimensie van het beeld van een lineaire afbeelding hoeft niet hetzelfde te zijn als die van het origineel. Een afbeelding van de 3-dimensionale ruimte naar het 2-dimensionale vlak wordt bijvoorbeeld beschreven door een matrix met de vorm

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

Een afbeelding van een p-dimensionale naar een q-dimensionale ruimte heeft dus p kolommen en q rijen. Bij het samenstellen van afbeeldingen (vermenigvuldigen van matrices) moeten deze afmetingen kloppen. In het matrixproduct $A \times B$ moet het aantal kolommen van A gelijk zijn aan het aantal rijen van B. Het aantal kolommen van A is immers de dimensie van het origineel van de afbeelding A; deze moet gelijk zijn aan de dimensie van het beeld van de afbeelding B. De samenstelling $A \times B$ heeft hetzelfde origineel als B, en dus evenveel kolommen als B; het heeft hetzelfde beeld als A, en dus evenveel rijen als A. Bijvoorbeeld:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}$$

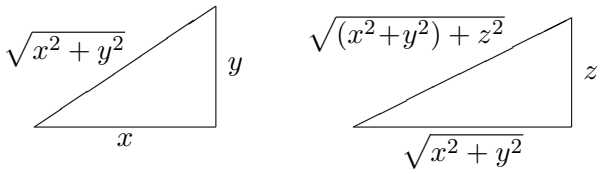
Het is duidelijk dat de begrippen "identiteitsmatrix" en "inverse" alleen zin hebben voor vierkante matrices, dus matrices met dezelfde origineel- en beeldruimte. En zelfs voor vierkante matrices is de inverse niet altijd gedefinieerd. De matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ bijvoorbeeld heeft geen inverse.

7.2.2 Elementaire operaties

In deze en de volgende paragraaf worden de definities gegeven van een aantal functies op vectoren en matrices.

Lengte van een vector

De lengte van een vector wordt bepaald volgens de stelling van Pythagoras. De volgende figuren illustreren de situatie in 2 en 3 dimensies:

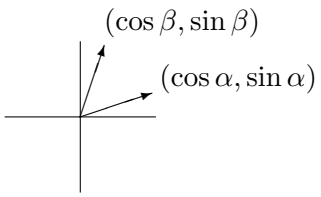


In het algemeen (willekeurige dimensie) kan de lengte van een vector uitgerekend worden door de wortel van de som van de kwadraten van de coördinaten te berekenen. De functie luidt dus:

```
vecLengte (Vec xs) = sqrt (sum (map kwadraat xs))
```

Hoek van twee vectoren

Bekijk twee vectoren met lengte 1. De eindpunten van deze vectoren liggen dus op de eenheidscirkel. Als de hoek die deze vectoren met de x-as maken respectievelijk α en β is, dan zijn de coördinaten van het eindpunt respectievelijk $(\cos\alpha, \sin\alpha)$ en $(\cos\beta, \sin\beta)$.



De hoek die de twee vectoren met elkaar maken is $\beta - \alpha$. Voor het verschil van twee hoeken geldt de volgende rekenregel:

$$\cos(\beta - \alpha) \;=\; \cos\alpha\ \cos\beta \;+\; \sin\alpha\ \sin\beta$$

In het geval van de twee vectoren is de cosinus van de ingesloten hoek dus gelijk aan de som van de producten van overeenkomstige coördinaten (dit is ook zo in hogere dimensies dan 2). Deze formule is zo belangrijk, dat hij een aparte naam heeft gekregen: het *inwendig product* van twee vectoren (of kortweg *inproduct*). De waarde wordt berekend door de volgende functie:

```
vecInprod (Vec xs) (Vec ys) = sum (zipWith (*) xs ys)
```

Voor vectoren met een andere lengte dan 1 moet het inproduct door de lengte gedeeld worden om de cosinus van de hoek te bepalen. De hoek kan dus als volgt berekend worden:

```
vecHoek v w = acos (vecInprod v w / (vecLengte v * vecLengte w))
```

De functie acos is de inverse van de cosinus. Als hij niet ingebouwd zou zijn, kon hij berekend worden met de functie inverse uit paragraaf 3.4.5. p. 75

De cosinus van zowel 90° als −90° is 0. Om te bepalen of twee vectoren loodrecht op elkaar staan, hoeft de arccos helemaal niet berekend te worden: het inproduct volstaat. Dit hoeft zelfs niet door de lengtes van de vectoren gedeeld te worden, omdat alleen het nul-zijn van belang is:

```
vecLoodrecht v w = vecInprod v w == 0.0
```

Vectoren optellen en verlengen

De functies vecScale en vecPlus zijn eenvoudige toepassingen van de standaardfuncties map en zipWith:

```
vecScale              :: Float -> Vector -> Vector
vecScale k (Vec xs)   = Vec (map (k*) xs)

vecPlus               :: Vector -> Vector -> Vector
vecPlus (Vec xs) (Vec ys) = Vec (zipWith (+) xs ys)
```

Soortgelijke functies zijn ook op matrices van nut. Het is handig om eerst twee functies te maken die werken zoals map en zipWith, maar dan op de elementen van *lijsten van* lijsten. Deze functies zullen we mapp en zippWith noemen (dit zijn geen standaardfuncties).

Om een functie toe te passen op alle elementen van een lijst van lijstjes, moet "het toepassen op alle elementen van een lijstje" worden toegepast op alle elementen van de grote lijst. Dus map f moet worden toegepast op alle elementen van de grote lijst:

```
mapp   :: (a -> b) -> [[a]] -> [[b]]
mapp f = map (map f)
```

Anders gezegd:

```
mapp = map . map
```

Voor zippWith geldt iets dergelijks:

```
zippWith :: (a -> b -> c) -> [[a]] -> [[b]] -> [[c]]
zippWith = zipWith . zipWith
```

Deze functies kunnen gebruikt worden in matScale en matPlus:

```
matScale                 : Float -> Matrix -> Matrix
matScale k (Mat xss)     = Mat (mapp (k*) xss)

matPlus                  :: Matrix -> Matrix -> Matrix
matPlus (Mat xss) (Mat yss) = Mat (zippWith (+) xss yss)
```

Matrices transponeren

Een getransponeerde matrix is een matrix waarvan de rijen en de kolommen zijn omgewisseld. Deze operatie is ook op gewone lijsten van lijsten (zonder de constructor Mat) van belang. Daarom schrijven we eerst een functie

```
transpose :: [[a]] -> [[a]]
```

Daarna is matTransp eenvoudig:

```
matTransp (Mat xss) = Mat (transpose xss)
```

De functie transpose is een generalisatie van zip. Waar zip *twee* lijsten aan elkaar ritst tot lijst van *tweetallen*, ritst transpose een *lijst van* lijsten aan elkaar tot een lijst van *lijsten*.

De functie kan recursief worden gedefinieerd, maar eerst bekijken we een voorbeeld. Er moet gelden:

```
transpose [ [1,2,3] , [4,5,6] , [7,8,9] , [10,11,12] ] = [ [1,4,7,10] , [2,5,8,11] , [3,6,9,12] ]
```

Als de lijst van lijsten maar uit één rij bestaat, is de functie eenvoudig: de rij van n elementen worden n kolommetjes van ieder één element. Dus:

```
transpose [rij] = map singleton rij
  where
    singleton x = [x]
```

Voor het recursieve geval gaan we ervan uit dat de getransponeerde van alles behalve de eerste rij al bepaald is. Dus in het voorbeeld van zoëven:

```
transpose [ [4,5,6] , [7,8,9] , [10,11,12] ] = [ [4,7,10] , [5,8,11] , [6,9,12] ]
```

Hoe moet de eerste rij [1,2,3] nu gecombineerd worden met deze deeloplossing tot een totale oplossing? De elementen ervan moeten steeds op kop gezet worden van de recursieve oplossing. Dus 1 komt op kop van [4,7,10], 2 komt op kop van [5,8,11], en 3 komt op kop van [6,9,12]. Dit kan eenvoudig met zipWith, als volgt:

```
transpose (xs : xss) = zipWith (:) xs (transpose xss)
```

Hiermee is de functie transpose gedefinieerd. Hij kan alleen niet toegepast worden op een matrix met nul rijen, maar dat is ook een beetje onzinnig geval.

Niet-recursieve matrix transponering

Er is ook een niet-recursieve definitie van transpose mogelijk, die het "vuile werk" laat opknappen door de standaardfunctie foldr. De definitie heeft namelijk de vorm

```
transpose (y : ys) = f y (transpose ys)
```

(met voor f de partiëel geparametriseerde functie zipWith (:)). Functies die deze vorm hebben zijn een speciaal geval van foldr (zie paragraaf 3.3.1). Als functieparameter van foldr kan f, dat wil zeggen zipWith (:), genomen worden. Blijft de vraag wat het "neutrale element" is, dat wil zeggen het resultaat van transpose [ ]. p. 64

Een "lege matrix" kan beschouwd worden als een matrix met 0 rijen van ieder n elementen. De getransponeerde daarvan is een matrix met n rijen van ieder 0 elementen, dus een lijst met daarin n lege lijstjes. Maar hoe groot is n? Er zijn slechts nul rijen beschikbaar, dus we kunnen niet even kijken hoe lang de eerste rij is. Om geen risico te lopen dat we n te klein kiezen, nemen we n oneindig groot: de getransponeerde van een matrix met 0 rijen van ∞ elementen is een matrix met ∞ rijen van 0 elementen. De functie zipWith zorgt er later wel voor dat deze oneindige lijst wordt ingekort op de gewenste lengte (het resultaat van zipWith op twee lijsten heeft de lengte van de kortste).

Deze wat ingewikkelde redenering levert een zeer elegante definitie voor transpose op:

```
transpose = foldr f e
  where
    f = zipWith (:)
    e = repeat [ ]
```

of zelfs kortweg

```
transpose = foldr (zipWith (:)) (repeat [])
```

Matrix op een vector toepassen

Een matrix is een representatie van een lineaire afbeelding tussen vectoren. De functie matApply voert deze lineaire afbeelding uit. Bijvoorbeeld:

```
matApply (Mat [ [1,2,3] , [4,5,6] ]) (Vec [x,y,z]) = Vec [ 1x+2y+3z , 4x+5y+6z ]
```

Het aantal *kolommen* van de matrix is gelijk aan het aantal coördinaten van de *originele* vector; het aantal *rijen* van de matrix is gelijk aan de dimensie van de *beeld*vector.

Voor elke coördinaat van de beeldvector zijn alleen maar de getallen uit de overeenkomstige rij van de matrix nodig. Het ligt dus voor de hand om matApply te schrijven als de map van een of andere functie op de (rijen van de) matix:

```
matApply (Mat m) v = Vec (map f m)
```

De functie f werkt daarbij op één rij van de matrix, en levert één element van de beeldvector op. Bijvoorbeeld op de tweede rij:

$$\mathsf{f}\ [4,5,6] \ = \ 4x + 5y + 6z$$

De functie f berekent dus het inproduct van zijn parameter met de vector v ($[x, y, z]$ in het voorbeeld). De complete functie matApply is dus:

```
matApply             :: Matrix -> Vector -> Vector
matApply (Mat m) v = Vec (map f m)
  where f rij = vecInprod (Vec rij) v
```

Twee dingen om op te letten in deze definitie:

- De constructorfunctie Vec wordt op de juiste plaatsen toegepast om het type correct te krijgen. De functie vecInprod verwacht twee vectoren. De parameter v is al een vector, maar rij is een ordinaire lijst, waarvan met Vec eerst een vector gemaakt moet worden.
- De functie f mag, behalve van zijn parameter rij, ook gebruik maken van v. Lokale functies mogen altijd gebruik maken van de parameters van de functie waarbinnen ze gedefinieerd worden.

De identiteitsmatrix

In elke dimensie is er een identiteitsafbeelding. Deze wordt beschreven door een vierkante matrix met een 1 op de diagonaal, en een 0 op alle andere plaatsen. Om een functie te

schrijven die voor elke dimensie de juiste identiteitsmatrix oplevert, is het handig om eerst een oneindig grote identiteitsmatrix te definiëren, dus de matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \ddots \end{pmatrix}$$

De eerste rij daarvan is een oneindige lijst nullen met een 1 op kop, dus 1 : repeat 0. De overige rijen worden bepaald door steeds een extra 0 op kop te zetten. Dit kan met de functie iterate:

```
matIdent :: Matrix
matIdent = Mat (iterate (0.0:) (1.0 : repeat 0.0))
```

Een identiteitsmatrix van dimensie n is nu te verkrijgen door n rijen van deze oneindige matrix te nemen, en elke rij af te breken op n elementen:

```
matId    :: Int -> Matrix
matId n = Mat (map (take n) (take n xss))
  where
    (Mat xss) = matIdent   -- uitpakken matrix
```

Matrixvermenigvuldiging

Het product van twee matrices beschrijft de afbeelding die de samenstelling is van de afbeeldingen die bij de twee matrices horen. Om te bekijken hoe het product berekend kan worden, passen we de matrices $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ en $\left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)$ na elkaar toe op de vector $\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$. (We noteren $\otimes$ voor matrixproduct en $\odot$ voor toepassing van een matrix op een vector. Let op het verschil tussen matrices en vectoren.)

$$\begin{aligned}
& \left(\begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right) \odot \begin{pmatrix} x \\ y \end{pmatrix} \\
= \;& \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \left(\begin{pmatrix} e & f \\ g & h \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix}\right) \\
= \;& \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} ex+fy \\ gx+hy \end{pmatrix} \\
= \;& \begin{pmatrix} a(ex+fy)+b(gx+hy) \\ c(ex+fy)+d(gx+hy) \end{pmatrix} \\
= \;& \begin{pmatrix} (ae+bg)x+(af+bh)y \\ (ce+dg)x+(cf+dh)y \end{pmatrix} \\
= \;& \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix}
\end{aligned}$$

Het product van twee matrices wordt dus als volgt berekend: elk element is het *inproduct* tussen een *rij* van de linker matrix en een *kolom* van de rechter matrix. De lengte van een rij (het aantal kolommen) van de linker matrix moet gelijk zijn aan de lengte van een kolom (het aantal rijen) van de rechter matrix. (Dat was aan het eind van de vorige paragraaf ook al opgemerkt).

Blijft de vraag hoe matrixvermenigvuldiging als functie geschreven kan worden. Daartoe kijken we nog eens naar het voorbeeld uit de vorige paragraaf:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}$$

Het aantal rijen van het resultaat is hetzelfde als het aantal rijen van de linker matrix. We proberen matProd daarom te schrijven als map op de linker matrix:

```
matProd (Mat a) (Mat b) = Mat (map f a)
```

Daarbij werkt de functie f op één rij van de linker matrix, en levert één rij van het resultaat. Bijvoorbeeld de tweede rij:

```
f [2,1,0] = [2,5,8,5]
```

Hoe worden de getallen $[2, 5, 8, 5]$ berekend? Door het inproduct van $[2, 1, 0]$ met alle *kolommen* van de rechter matrix te bepalen.

Matrices worden echter opgeslagen als *rijen*. Om de kolommen te krijgen, moet de transpose-functie worden toegepast. Het resultaat is dus:

```
matProd (Mat a) (Mat b) = Mat (map f a)
  where
    f aRij = map (vecInprod (Vec aRij)) bKols
    bKols = map Vec (transpose b)
```

De functie transpose werkt op een "kale" lijst van lijsten (dus niet op matrices). Hij levert weer een lijst van lijsten op. Met map Vec wordt daar een lijst van vectoren van gemaakt. Van al deze vectoren kan het inproduct met de rijen van a (beschouwd als vectoren) berekend worden.

7.2.3 Determinant en inverse

Nut van de determinant

Alleen bijectieve (één-op-één en "op") afbeeldingen zijn inverteerbaar. Als er beeldpunten zijn met meer dan één origineel, is het namelijk onduidelijk waarheen de inverse afbeelding hem moet terugsturen. Ook als er punten in de beeldruimte zijn zonder origineel, kan de inverse afbeelding niet worden gedefinieerd.

Als een matrix niet vierkant is, is hij dus niet inverteerbaar (de beeldruimte heeft dan immers een lagere dimensie (waardoor er beeldpunten zijn met meer originelen) of een hogere dimensie (waardoor er beeldpunten zijn zonder origineel)). Maar zelfs vierkante matrices stellen niet altijd bijectieve afbeeldingen voor. De matrix $\left(\begin{smallmatrix}0&0\\0&0\end{smallmatrix}\right)$ beeldt bijvoorbeeld elk punt af op de oorsprong, en heeft dus geen inverse. De matrix $\left(\begin{smallmatrix}1&1\\2&2\end{smallmatrix}\right)$ beeldt elk punt af op een punt van de lijn $y = 2x$, en heeft dus ook geen inverse. Ook de matrix $\left(\begin{smallmatrix}1&2\\3&6\end{smallmatrix}\right)$ beeldt alle punten af op een lijn.

Alleen als de tweede rij van een 2-dimensionale matrix geen veelvoud is van de eerste, is de matrix inverteerbaar. Een matrix $\left(\begin{smallmatrix}a&b\\c&d\end{smallmatrix}\right)$ is dus alleen inverteerbaar als $\frac{a}{c} \neq \frac{b}{d}$, oftewel $ad - bc \neq 0$. Deze waarde wordt de *determinant* van de matrix genoemd. Als de determinant nul is, is de matrix niet inverteerbaar; als hij ongelijk aan nul is, is de matrix wel inverteerbaar.

Ook voor (vierkante) matrices van hogere dimensie kan een determinant berekend worden. Voor een 3×3-matrix gaat dat als volgt:

$$\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \det\begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \det\begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \det\begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

Je begint dus met de matrix te splitsen in een eerste rij (a, b, c) en overige rijen $\left(\begin{smallmatrix}d&e&f\\g&h&i\end{smallmatrix}\right)$. Dan bereken je de determinanten van de 2×2-matrices die je krijgt door uit de "overige rijen" steeds één kolom weg te laten ($\left(\begin{smallmatrix}e&f\\h&i\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}d&f\\g&i\end{smallmatrix}\right)$ en $\left(\begin{smallmatrix}d&e\\g&h\end{smallmatrix}\right)$). Die vermenigvuldig je met de overeenkomstige elementen uit de eerste rij. Tenslotte tel je ze op, waarbij om de andere term een minteken krijgt. Dit werkt ook in hogere dimensies dan 3.

Definitie van de determinant

Van deze informele beschrijving van de determinant gaan we een functiedefinitie maken. Er is sprake van het afsplitsen van de eerste rij van de matrix, dus de definitie heeft de vorm

```
det (Mat (ry : rys)) = ...
```

Uit de restrijen rys moeten kolommen worden weggelaten. Om van rijen kolommen te maken moeten ze getransponeerd worden. De definitie wordt dus zoiets als

```
det (Mat (ry : rys)) = ...
  where
    kols = transpose rys
```

Uit de lijst van kolommen moet op alle mogelijke manieren één kolom worden weggelaten. Dat kan met de combinatorische functie gaps uit opgave 7.8. Het resultaat is een lijst van n lijsten van lijsten. Die moeten weer teruggetransponeerd worden, en er moeten met Mat kleine matrixjes van gemaakt worden:

p. 170

```
det (Mat (ry : rys)) = ...
  where
    kols  = transpose rys
    mats = map (Mat.transpose) (gaps kols)
```

Van al deze matrixjes moet de determinant berekend worden. Dat gebeurt natuurlijk door de functie recursief aan te roepen. De determinanten die het resultaat zijn, moeten vermenigvuldigd worden met de overeenkomstige elementen van de eerste rij:

```
det (Mat (ry : rys)) = ...
  where
    kols  = transpose rys
    mats  = map (Mat . transpose) (gaps kols)
    prods = zipWith (*) ry (map det mats)
```

De producten prods die hierdoor worden opgeleverd, moeten worden opgeteld, waarbij de termen afwisselend een plus- en een minteken krijgen. Dat kan bijvoorbeeld met behulp van de functie altsum (voor "alternerende som") die gedefinieerd kan worden met behulp van een oneindige lijst:

```
altsum xs = sum (zipWith (*) xs plusMinEen)
  where
    plusMinEen = 1.0 : -1.0 : plusMinEen
```

De functiedefinitie wordt dus:

```
det (Mat (ry : rys)) = altsum prods
  where
    kols  = transpose rys
    mats  = map (Mat . transpose) (gaps kols)
    prods = zipWith (*) ry (map det mats)
```

Dit kan nog wat vereenvoudigd worden. Het terugtransponeren van de matrixjes is namelijk niet nodig, omdat de determinant van een getransponeerde matrix hetzelfde is als van de matrix zelf. Bovendien is het niet nodig om de tussenresultaten (kols, mats en prods) een naam te geven. De definitie kan dus luiden:

```
det (Mat (ry : rys)) =
  altsum (zipWith (*) ry (map det (map Mat (gaps (transpose rys)))))
```

Omdat det een recursieve functie is, moeten we niet vergeten een niet-recursief basisgeval toe te voegen. Daarvoor kan het 2×2-geval gebruikt worden, maar nog makkelijker is het om het 1×1-geval te definiëren:

```
det (Mat [[x]]) = x
```

Het eindresultaat, waarin nog wat haakjes zijn weggewerkt door functiesamenstelling te gebruiken, is als volgt:

```
det                  :: Matrix -> Float
det (Mat [[x]])      = x
det (Mat (ry : rys)) =
  (altsum . zipWith (*) ry . map det . map Mat . gaps . transpose) rys
```

Inverse van een matrix

De determinant is niet alleen van nut om te bepalen of de inverse bestaat, maar ook om de inverse daadwerkelijk uit te rekenen. De determinant moet dan natuurlijk wel ongelijk aan nul zijn.

De inverse van een 3×3-matrix kan als volgt worden uitgerekend: bepaal een matrix van negen 2×2-matrixjes, die onstaan door een rij en een kolom weg te laten uit de matrix. Bereken overal de determinant van, zet afwisselend plus- en mintekens, en deel alles door de determinant van de gehele matrix (die $\neq 0$ moet zijn!).

Een voorbeeld is waarschijnlijk duidelijker. De inverse van $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$ wordt als volgt uitgerekend:

$$\frac{\begin{pmatrix} +\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ \cancel{b} & e & h \\ \cancel{c} & f & i \end{pmatrix} & -\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ \cancel{c} & f & i \end{pmatrix} & +\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & e & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\ -\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & \cancel{e} & h \\ c & \cancel{f} & i \end{pmatrix} & +\det\begin{pmatrix} a & \cancel{d} & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & \cancel{f} & i \end{pmatrix} & -\det\begin{pmatrix} a & \cancel{d} & g \\ b & \cancel{e} & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\ +\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & e & \cancel{h} \\ c & f & \cancel{i} \end{pmatrix} & -\det\begin{pmatrix} a & d & \cancel{g} \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & f & \cancel{i} \end{pmatrix} & +\det\begin{pmatrix} a & d & \cancel{g} \\ b & e & \cancel{h} \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \end{pmatrix}}{\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}$$

De doorgestreepte elementen staan in dit voorbeeld alleen maar om aan te geven welke elementen weggelaten moeten worden; er staan dus negen 2×2-matrixjes.

Let goed op welke elementen worden weggelaten: in de r-de rij van de grote matrix wordt steeds de r-de *kolom* van de matrixjes weggelaten, terwijl in de k-de kolom van de grote matrix juist de k-de *rij* van de matrixjes wordt weggelaten.

De matrixinverse functie matInv kan nu op vergelijkbare wijze als de functie det geschreven worden, dat wil zeggen door op de juiste momenten gebruik te maken van gaps, transpose, Mat, enzovoort. Het wordt aan de lezer overgelaten om de details uit te werken (zie opgave 7.11). p. 170

7.3 Polynomen

7.3.1 Representatie

Een *polynoom* is een som van *termen*, waarbij elke term bestaat uit het product van een reëel getal en een natuurlijke macht van een variabele.

$$x^2 + 2x + 1$$
$$4.3x^3 + 2.5x^2 + 0.5$$
$$6x^5$$
$$x$$
$$3$$

De hoogste macht die voorkomt heet de *graad* van het polynoom. In bovenstaande voorbeelden is de graad dus achtereenvolgens 2, 3, 5, 1 en 0. Het lijkt misschien raar om 3 een polynoom te noemen; het getal 3 is echter gelijk aan $3x^0$, en is dus inderdaad een product van een getal en een natuurlijke macht van x.

Met polynomen kun je rekenen: je kunt polynomen bij elkaar optellen, van elkaar aftrekken en met elkaar vermenigvuldigen. Het product van de polynomen $x+1$ en x^2+3x is bijvoorbeeld $x^3 + 4x^2 + 3x$. Als je twee polynomen echter door elkaar deelt is het resultaat niet altijd een polynoom. Het komt nu goed uit dat getallen ook polynomen zijn: zo is het resultaat van het optellen van $x + 1$ en $-x$ het polynoom 1.

In deze paragraaf wordt een datatype *Poly* ontworpen, waarmee polynomen kunnen worden gerepresenteerd. In de volgende paragraaf worden een aantal functies gedefinieerd, die op dat soort polynomen werken:

```
pPlus    :: Poly -> Poly -> Poly
pMin     :: Poly -> Poly -> Poly
pMaal    :: Poly -> Poly -> Poly
pEq      :: Poly -> Poly -> Bool
pGraad   :: Poly -> Int
pEval    :: Float -> Poly -> Float
polyString :: Poly -> String
```

Een mogelijke representatie voor polynomen is “functie van float naar float”. Het nadeel daarvan is echter dat je het resultaat van het vermenigvuldigen van twee polynomen niet meer als polynoom kunt inspecteren; je hebt dan een functie die je alleen nog maar op waarden kunt loslaten. Ook is het dan niet mogelijk om een gelijkheidsoperator te schrijven; het is dus niet mogelijk om te testen of het product van de polynomen x en $x + 1$ gelijk is aan het polynoom $x^2 + x$.

Het is dus beter om een polynoom te representeren als een datastructuur met getallen. Daarbij ligt het voor de hand om een polynoom voor te stellen als lijst termen, waarbij elke term gekenmerkt wordt door een *Float* (de coëfficiënt) en een *Int* (de exponent).

Een polynoom kan dus worden gerepresenteerd als een lijst tweetupels. We maken er echter meteen maar een *data*type van met de volgende definitie:

data *Poly* = *Poly* [*Term*]
data *Term* = *Term* (*Float*, *Int*)

Let op: de namen *Poly* en *Term* worden dus zowel als naam van het type gebruikt, als naam van de (enige) constructorfunctie. Dit is toegestaan, want het is uit de context altijd duidelijk welke van de twee bedoeld wordt. Het woord *Poly* is een type in typedeclaraties zoals

pEq :: *Poly* -> *Poly* -> *Bool*

maar het is een constructorfunctie in functiedefinities zoals

pGraad (*Poly* []) = ...

Een aantal voorbeelden van representaties van polynomen is:

$3x^5 + 2x^4$	*Poly* [*Term* (3.0, 5), *Term* (2.0, 4)]
$4x^2$	*Poly* [*Term* (4.0, 2)]
$2x + 1$	*Poly* [*Term* (2.0, 1), *Term* (1.0, 0)]
3	*Poly* [*Term* (3.0, 0)]
0	*Poly* []

Net als bij de rationale getallen uit paragraaf 4.3.3 hebben we hier weer het probleem dat er meerdere representaties zijn voor één polynoom. Het polynoom x^2+7 kan bijvoorbeeld worden gerepresenteerd door de volgende expressies: p. 110

Poly [*Term* (1.0, 2), *Term* (7.0, 0)]
Poly [*Term* (7.0, 0), *Term* (1.0, 2)]
Poly [*Term* (1.0, 2), *Term* (3.0, 0), *Term* (4.0, 0)]

Net als bij rationale getallen is het dus nodig om een polynoom te "vereenvoudigen" nadat er operaties op zijn uitgevoerd. Vereenvoudigen bestaat in dit geval uit:

- sorteren van de termen, zodat de termen met de hoogste exponent voorop staan;
- samenvoegen van termen met gelijke exponent;
- verwijderen van termen met coëfficiënt nul.

Een alternatieve methode is om de polynomen niet te vereenvoudigen, maar dan moet er extra werk gedaan worden in de functie pEq waarmee polynomen vergeleken worden.

7.3.2 Vereenvoudiging

Voor het vereenvoudigen van polynomen schrijven we een functie

pEenvoud :: *Poly* -> *Poly*

Deze functie voert de drie genoemde aspecten van het vereenvoudigen uit, en kan dus geschreven worden als functiesamenstelling:

```
pEenvoud (Poly xs) = Poly (eenvoud xs)
  where
    eenvoud = verwijderNul . samenvExpo . sortTerms
```

Blijft de taak over om de drie samenstellende functies te schrijven. Alledrie werken ze op lijsten van termen.

In paragraaf 4.1.5 werd een functie gedefinieerd die een lijst sorteert. Daarbij moesten de waarden echter ordenbaar zijn, en werd de lijst gesorteerd van klein naar groot. Er is een algemenere sorteerfunctie denkbaar, waarbij als extra parameter een criterium wordt meegegeven dat beslist in welke volgorde de elementen komen te staan. Deze functie zou hier goed van pas komen, want hij kan dan gebruikt worden met "heeft een grotere exponent" als sorteercriterium. p. 92

Daarom schrijven we eerst een functie sortVolgens, die dan gebruikt kan worden bij het schrijven van sortTerms.

De algemene sorteerfunctie sortVolgens, heeft als type:

```
sortVolgens :: (a -> a -> Bool) -> [a] -> [a]
```

Behalve de te sorteren lijst heeft de functie een functie als parameter. Die parameterfunctie levert *True* op als zijn eerste parameter vóór zijn tweede parameter moet komen. De definitie van sortVolgens lijkt sterk op die van isort in paragraaf 4.1.5. Het verschil is dat nu de als extra parameter meegegeven vergelijkfunctie wordt gebruikt in plaats van $<$. De definitie wordt dan: p. 92

```
sortVolgens komtVoor xs     = foldr (insertVolgens komtVoor) [] xs

insertVolgens komtVoor e [] = [e]
insertVolgens komtVoor e (x : xs)
   | e `komtVoor` x         = e : x : xs
   | otherwise              = x : insertVolgens komtVoor e xs
```

Voorbeelden van het gebruik van sortVolgens zijn:

```
? sortVolgens (<) [1, 3, 2, 4]
[1, 2, 3, 4]
? sortVolgens (>) [1, 3, 2, 4]
[4, 3, 2, 1]
```

Bij het sorteren van termen op grond van hun exponent kan sortVolgens nu worden gebruikt. Deze functie krijgt als komtVoor-functie een functie mee die kijkt of de exponent groter is:

```
sortTerms :: [Term] -> [Term]
sortTerms = sortVolgens expoGroter
  where
    Term (c1, e1) `expoGroter` Term (c2, e2) = e1 > e2
```

De tweede functie die nodig is, is de functie die termen met gelijke exponenten samenvoegt. Deze functie mag er van uitgaan dat de termen al zijn gesorteerd op exponent. Termen met gelijke exponent staan dus naast elkaar. De functie laat lijsten met nul of één element ongemoeid. Bij lijsten met twee of meer elementen zijn er twee mogelijkheden:

- de exponenten van de eerste twee elementen zijn gelijk; de elementen worden samengevoegd, het nieuwe element wordt op kop van de rest gezet, en de functie wordt opnieuw aangeroepen, zodat het nieuwe element eventueel met nog meer elementen samengevoegd kan worden.
- de exponenten van de eerste twee elementen zijn *niet* gelijk; het eerste element komt dan onveranderd in het resultaat, de rest wordt aan een nadere inspectie onderworden (misschien is het tweede element wel gelijk aan het derde).

Dit alles komt terug in de definitie:

```
samenvExpo    :: [Term] -> [Term]
samenvExpo [] =  []
samenvExpo [t] = [t]
samenvExpo (Term (c1, e1) : Term (c2, e2) : ts)
   | e1 == e2  =  samenvExpo (Term (c1 + c2, e1) : ts)
   | otherwise =  Term (c1, e1) : samenvExpo (Term (c2, e2) : ts)
```

De derde benodigde functie is eenvoudig te maken:

```
verwijderNul :: [Term] -> [Term]
verwijderNul = filter coefNietNul
  where
    coefNietNul (Term (c, e)) = c /= 0.0
```

Desgewenst kunnen de drie functies lokaal gedefinieerd worden in pEenvoud:

```
pEenvoud (Poly xs) = Poly (eenvoud xs)
  where
    eenvoud                              = vN . sE . sT
    sT                                   = sortVolgens expoGroter
    sE []                                = []
    sE [t]                               = [t]
    sE (Term (c1, e1) : Term (c2, e2) : ts)
       | e1 == e2                        = sE (Term (c1 + c2, e1) : ts)
       | otherwise                       = Term (c1, e1) : sE (Term (c2, e2) : ts)
    vN                                   = filter coefNietNul
    coefNietNul (Term (c, e))            = c /= 0.0
    Term (c1, e1) `expoGroter` Term (c2, e2) = e1 > e2
```

De functie pEenvoud verwijdert *alle* termen waarvan de coëfficiënt nul is. Het nulpolynoom wordt dus gerepresenteerd door *Poly* [ ], een lege lijst termen. Daarmee verschilt het nulpolynoom van andere polynomen waarin de variabele niet voorkomt. Het polynoom "3" wordt bijvoorbeeld gerepresenteerd door *Poly* [*Term* (3.0, 0)].

7.3.3 Rekenkundige operaties

Het optellen van twee polynomen is eenvoudig. De lijsten van termen kunnen gewoon geconcateneerd worden. Daarna zorgt pEenvoud ervoor dat de termen gesorteerd worden, gelijke exponenten samengenomen worden, en nultermen verwijderd worden:

```
pPlus                       :: Poly -> Poly -> Poly
pPlus (Poly xs) (Poly ys) = pEenvoud (Poly (xs ++ ys))
```

Voor het aftrekken van twee polynomen tellen we het eerste polynoom op bij het tegengestelde van het tweede:

```
pMin       :: Poly -> Poly -> Poly
pMin p1 p2 = pPlus p1 (pNeg p2)
```

Blijft natuurlijk de vraag hoe het tegengestelde van een polynoom berekend wordt: daartoe moet het tegengestelde van elke term berekend worden.

```
pNeg              :: Poly -> Poly
pNeg (Poly xs)    = Poly (map tNeg xs)

tNeg              :: Term -> Term
tNeg (Term (c, e)) = Term (-c, e)
```

Vermenigvuldigen van polynomen is wat moeilijker. Daarvoor moet elke term van het eerste polynoom vermenigvuldigd worden met elke term van het andere polynoom. Dat vraagt om een hogere-orde functie: "doe iets met elk element van een lijst in combinatie met elk element van een andere lijst". Deze functie noemen we cpWith. De `cp` staat voor *cross product*, de `With` is naar analogie van de functie zipWith. Als de eerste lijst leeg is, valt er niets samen te stellen. Als de eerste lijst de vorm x : xs heeft, moet het eerste element x met alle elementen van de tweede lijst samengesteld worden, en moet bovendien het cross-product van xs en de tweede lijst nog bepaald worden. Dit geeft de definitie:

```
cpWith              :: (a -> b -> c) -> [a] -> [b] -> [c]
cpWith f [] ys      = []
cpWith f (x : xs) ys = map (f x) ys ++ cpWith f xs ys
```

Deze definitie kan meteen gebruikt worden bij het vermenigvuldigen van polynomen:

```
pMaal                       :: Poly -> Poly -> Poly
pMaal (Poly xs) (Poly ys) = pEenvoud (Poly (cpWith tMaal xs ys))
```

Hierin wordt de functie tMaal gebruikt, die twee termen vermenigvuldigd. Zoals uit het voorbeeld $3x^2$ maal $5x^4$ is $15x^6$ blijkt, moeten daartoe de coëfficiënten worden vermenigvuldigd, en de exponenten opgeteld:

```
tMaal                                  :: Term -> Term -> Term
tMaal (Term (c1, e1)) (Term (c2, e2)) = Term (c1 * c2, e1 + e2)
```

Doordat we polynomen steeds vereenvoudigen, en dus steeds de term met de hoogste exponent voorop staat, is de graad van een polynoom gelijk aan de exponent van de eerste term. Alleen voor het nulpolynoom hebben we een aparte definitie nodig.

```
pGraad                          :: Poly -> Int
pGraad (Poly [])                = 0
pGraad (Poly (Term (c, e) : ts)) = e
```

Twee vereenvoudigde polynomen zijn gelijk als alle termen gelijk zijn. Twee termen zijn gelijk als de coëfficiënt en de exponent overeenstemmen. Dit alles laat zich gemakkelijk naar functies vertalen:

```
pEq                                  :: Poly -> Poly -> Bool
pEq (Poly xs) (Poly ys)              = length xs == length ys
                                         && and (zipWith tEq xs ys)

tEq                                  :: Term -> Term -> Bool
tEq (Term (c1, e1)) (Term (c2, e2)) = eqFloat c1 c2 && e1 == e2
```

De functie pEval moet een polynoom uitrekenen met een specifieke waarde voor x ingevuld. Daartoe moeten alle termen geëvalueerd worden, en de resultaten opgeteld:

```
pEval                :: Float -> Poly -> Float
pEval w (Poly xs)    = sum (map (tEval w) xs)

tEval                :: Float -> Term -> Float
tEval w (Term (c, e)) = c * w ^ e
```

Tenslotte schrijven we een functie voor de weergave van een polynoom als string. Hiervoor moeten de termen worden weergegeven als string, en ertussen moet een +-teken komen:

```
polyString               :: Poly -> String
polyString (Poly [])     = "0"
polyString (Poly [t])    = termString t
polyString (Poly (t : ts)) = termString t ++ " + "
                               ++ polyString (Poly ts)
```

Bij de weergave van een term laten we de coëfficiënt en de exponent weg als die 1 is. Als de exponent 0 is, wordt de variabele weggelaten, maar de coëfficiënt nooit. De exponent

duiden we aan met een ^-teken, net zoals dat in Haskell-expressies gebruikelijk is. De functie wordt daarmee:

```
termString                  :: Term -> String
termString (Term (c, 0))    = floatString c
termString (Term (1.0, 1))  = "x"
termString (Term (1.0, e))  = "x^" ++ intString e
termString (Term (c, e))    = floatString c ++ "x^" ++ intString e
```

De functie floatString is nog niet gedefinieerd. Met een boel moeite is dat wel mogelijk, maar dat is niet nodig: in paragraaf 10.2.4 wordt hiervoor de functie show geïntroduceerd. p. 200

Opgaven

7.1 Hoe wordt de volgorde van de elementen van segs $[1, 2, 3, 4]$ als de parameters van $++$ in de definitie van segs worden omgewisseld?

7.2 Schrijf segs als combinatie van inits, tails en standaardfuncties. De volgorde van de elementen in het resultaat hoeft niet hetzelfde te zijn als op blz. 144.

7.3 Gegeven is een lijst xs met n elementen. Bepaal het aantal elementen van inits xs, segs xs, subs xs, perms xs, en combs k xs.

7.4 Schrijf de functie inits met behulp van een foldr.

7.5 Waarom laat de functie tails zich niet direct m.b.v. een foldr uitdrukken?

7.6 Kun je de functie segs ook uitdrukken met behup van een foldr?

7.7 Schrijf een functie bins :: *Int* -> [[*Char*]] die alle getallen in het tweetallig stelsel (als strings nullen en enen) bepaalt met het gegeven aantal cijfers. Vergelijk de functie met de functie subs.

7.8 Schrijf een functie gaps die alle mogelijkheden geeft om één element uit een lijst weg te laten. Bijvoorbeeld:

```
gaps [1,2,3,4,5] = [ [2,3,4,5] , [1,3,4,5] , [1,2,4,5] , [1,2,3,5] , [1,2,3,4] ]
```

7.9 Vergelijk de voorwaarden wat betreft de afmetingen van de matrices bij matrixvermenigvuldiging met het type van de functiesamenstellingsoperator (.).

7.10 Ga na dat de recursieve definitie van matrixdeterminant det overeenkomt met de expliciete definitie voor determinanten van 2×2-matrices uit paragraaf 7.2.3. p. 160

7.11 Schrijf de functie matInv.

7.12 In paragraaf 7.2.2 werd de functie transpose beschreven als generalisatie van zip. p. 156 In paragraaf 4.3.4 werd zip geschreven als zipWith maak2tupel. De functie cp p. 112 wordt nu gedefinieerd als cpWith maak2tupel. Schrijf een functie crossprod die een generalisatie is van cp zoals transpose een generalisatie is van zip. Hoe kan length (crossprod xs) berekend worden zonder crossprod te gebruiken?

7.13 Een andere mogelijke representatie van polynomen is als een lijst van coëfficiënten. De exponenten worden dus niet opgeslagen. De prijs daarvan is dat "ontbrekende" termen als 0.0 opgeslagen moeten worden. Het is het handigst om de termen met de kleinste exponent aan het begin van de lijst op te slaan. Dus bijvoorbeeld:

$x^2 + 2x$	$[0.0, 2.0, 1.0]$
$4x^3$	$[0.0, 0.0, 0.0, 4.0]$
5	$[5.0]$

Schrijf functies voor de graad van een polynoom en voor de som en het product van twee polynomen in deze representatie.

8 Case study: symbolische berekeningen

8.1 Rekenkundige expressies

Datadeclaraties worden gebruikt om de vorm van datastructuren te beschrijven. Een veel voorkomende niet-lijstvormige datastructuur is de *ontleedboom*. Een ontleedboom is een symbolische beschrijving van een expressie. Een expressie wordt in een ontleedboom dus in de vorm van een datastructuur opgeslagen.

Numerieke expressies worden bijvoorbeeld beschreven door bomen die zijn opgebouwd volgens de volgende datadeclaratie:

```
data Expr = Con Float
          | Var String
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
```

De datadeclaratie beschrijft de opbouw van rekenkundige expressies. Voor elk soort expressie (constante, variabele, optelling, aftrekking, vermenigvuldiging en deling) is er een constructor waarmee de representatie van de expressie kan worden opgebouwd.

Twee van de zes contructoren (*Con* en *Var*) hebben één parameter. De andere vier hebben twee parameters. Voor de duidelijkheid schrijven we ze als infixoperator, dus tussen de parameters in plaats van er voor. Operatoren die een constructor voorstellen in plaats van een gewone functie moeten met een dubbele punt beginnen. Voor de symmetrie (het oog wil ook wat) eindigen de operatoren in de gegeven datadeclaratie ook op een dubbele punt. De drie symbolen in :+: vormen één operator, en moeten dan ook zonder spatie ertussen geschreven worden.

De constructoroperatoren uit de datadeclaratie kunnen van een prioriteit en een associatievolgorde worden voorzien met behulp van een infixdeclaratie. Deze werd ingevoerd in paragraaf 3.1.4. Het is het handigste om de operatoren van dezelfde prioriteit te voorzien als de gewone rekenkundige operatoren: p. 59

```
infixl 7 :*:
infix  7 :/:
infixl 6 :+:, :-:
```

Na deze declaraties kan bijvoorbeeld de expressie $3x + 4y$ als datastructuur worden gerepresenteerd door de Haskell-expressie

```
Con 3.0 :*: Var "x" :+: Con 4.0 :*: Var "y"
```

Het is belangrijk om onderscheid te maken tussen expressies in Haskell, en rekenkundige expressies in het taaltje dat door de datadeclaratie wordt beschreven. Zo is *Var* `"x"` een Haskell-expressie die binnen Haskell de rekenkundige expressie x beschrijft.

Als we de waardes hebben van de variabelen die in een expressie voor komen, kunnen we de waarde van een *Expr* ook uitrekenen. We noemen een functie die variabelen op waarden afbeeldt vaak een *omgeving* (in Engels, *environment*). We maken weer gebruik van de structuur van de boom, en lopen die netjes af, telkens de omgeving meegevend.

```
type Env = String -> Float
eval :: Env -> Expr -> Float
eval env (Con f)   = f
eval env (Var s)   = env s
eval env (l :+: r) = (eval env l) + (eval env r)
eval env (l :-: r) = (eval env l) - (eval env r)
eval env (l :*: r) = (eval env l) * (eval env r)
eval env (l :/: r) = (eval env l) / (eval env r)
```

Als we geen zin hebben in zoveel tikwerk, kunnen we de eigenlijke evaluatiefunctie ook wel locaal definiëren:

```
eval env t = eval' t
  where eval' (Con f)   = f
        eval' (Var s)   = env s
        eval' (l :+: r) = eval' l + eval' r
        eval' (l :-: r) = eval' l - eval' r
        eval' (l :*: r) = eval' l * eval' r
        eval' (l :/: r) = eval' l / eval' r
```

Een veel voorkomende operatie is een waarbij we sommige variabelen in een expressieboom willen vervangen door een andere boom. We noemen dit een substitutie (*substitution*). We definieren nu substitutie m.b.v. de functie `lookup` uit de prelude, waarbij de te vervangen variabelen met hun nieuwe waarde in de een tabel zijn gerepresenteerd:

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b   -- uit de prelude
subst :: [(String, Expr)] -> Expr -> Expr
subst env t = subst' t
  where subst' (Con f) = Con f
        subst' (Var s) = case lookup s env of
                           Nothing -> Var s
                           Just v  -> v
        subst' (l :+: r) = subst' l :+: subst' r
        subst' (l :-: r) = subst' l :-: subst' r
        subst' (l :*: r) = subst' l :*: subst' r
        subst' (l :/: r) = subst' l :/: subst' r
```

8.2 Symbolisch differentiëren

Het voordeel van de representatie van expressies als datastructuren is dat we functies kunnen schrijven die op expressies werken, en het resultaat kunnen bekijken. Dit wordt *symbolische manipulatie* van expressies genoemd. Een goed voorbeeld van symbolische manipulatie is het *differentiëren* van een expressie. Als we de expressie *Var* `"x"`:∗: *Var* `"x"` differentiëren, dan komt daar de expressie *Con* 2.0 :∗: *Var* `"x"` uit, of iets wat daar equivalent aan is.

Het symbolische differentiëren van een expressie biedt veel meer mogelijkheden dan het numeriek differentiëren, waarvoor in paragraaf 3.4.2 de volgende functie werd geschreven: p. 72

```
diff f = f'
  where
    f' x = (f (x + h) − f x) / h
    h    = 0.0001
```

Een numeriek gedifferentiëerde functie is immers een functie; het enige wat we met die functie kunnen doen is hem op een parameter toepassen. Het is niet mogelijk om het functievoorschrift van de numeriek gedifferentieerde functie te zien te krijgen. Bij het gebruik van expressiebomen en symbolisch differentiëren is dat wèl mogelijk.

De symbolische differentieerfunctie heeft behalve een expressie een *String* als parameter die aangeeft naar welke variabele de expressie gedifferentieerd moet worden (dit is ook iets wat bij numeriek differentiëren niet mogelijk was). In de definitie wordt voor elk van de zes constructoren van expressies aangegeven hoe de expressie gedifferentieerd kan worden. Daarbij kunnen de bekende rekenregels voor differentiëren gevolgd worden: de afgeleide van een som is bijvoorbeeld de som van de afgeleiden, en voor de afgeleide van een product geldt de "productregel" $((fg)' = fg' + gf')$.

```
afg                :: Expr -> String -> Expr
afg (Con c) dx = Con 0.0
afg (Var x) dx
   | x == dx   = Con 1.0
   | otherwise = Con 0.0
afg (f :+: g) dx = afg f dx :+: afg g dx
afg (f :−: g) dx = afg f dx :−: afg g dx
afg (f :∗: g) dx =  f :∗: afg g dx :+: g :∗: afg f dx
afg (f :/: g) dx = (g :∗: afg f dx :−: f :∗: afg g dx)
                   :/:
                   (g :∗: g)
```

Uit de definitie blijkt verder dat de afgeleide van een constante de constante 0 is. De afgeleide van een variabele is de constante 1 als het de variabele betreft waarnaar gedifferentieerd wordt (in de definitie suggestief dx genoemd). Andere variabelen gedragen zich als constanten.

8.3 Andere expressiebomen

Naast de datadeclaratie die rekenkundige expressies beschrijft, zijn ook andere soorten expressies te beschrijven met datadeclaraties. Onderstaande datadeclaratie beschrijft bijvoorbeeld Boolese expressies oftewel *proposities*:

```
data Prop = Cons Bool
          | Vari String
          | Not Prop
          | Prop :∧: Prop
          | Prop :∨: Prop
          | Prop :⇒: Prop
```

De propositie $b \vee \neg b$ wordt bijvoorbeeld gerepresenteerd door de datastructuur

```
Vari 'b' :∨: Not (Vari 'b')
```

Functies op het type *Prop* worden natuurlijk weer geschreven door voor alle vijf constructoren een patroon te gebruiken. De functie verw bijvoorbeeld, verwijdert alle voorkomens van :∨: en :⇒: uit een propositie. Daarvoor worden rekenregels uit de propositielogica, zoals de wet van De Morgan toegepast. We definiëren ook een zogenaamde *smart*-constructor not die het herhaaldelijk toepassen van *Not* tijdens het bouwen van het resultaat elimineert.

```
verw (Cons b)  = Cons b
verw (Vari x)  = Vari x
verw (Not p)   = not (verw p)
                 where not (Not p) = p
                       not p = Not p
verw (p :∧: q) = verw p :∧: verw q
verw (p :∨: q) = Not (Not (verw p) :∧: Not (verw q))
verw (p :⇒: q) = verw (Not p :∨: q)
```

Met datadeclaraties kunnen naast expressies ook taalconstructies uit programmeertalen worden beschreven. Statements uit een imperatieve taal kunnen bijvoorbeeld worden beschreven door de volgende datadeclaratie:

```
data Stat = Assign Char Expr
          | If Prop Stat Stat
          | While Prop Stat
          | Repeat Stat Prop
          | Compound [Stat]
```

Door functies te schrijven die op dit soort datastructuren werken, is het mogelijk om programma's (althans de boomrepresentaties daarvan) te transformeren volgens "rekenregels" die daarvoor gelden.

8.4 Stringrepresentatie van een boom

We keren weer terug naar de expressiebomen uit paragraaf 8.1. De Haskell-expressies p. 172
die nodig zijn om een rekenkundige expressie als boom te representeren, lijken sterk op die expressies zelf. Zo wordt bijvoorbeeld de expressie $x * x + 1$ gerepresenteerd door *Var* `"x"` :*: *Var* `"x"` :+: *Con* 1.0. Het is alleen jammer dat op elke constante eerst de constructor *Con* moet worden toegepast, op elke variabele *Var*, en dat elke operator van dubbele punten moet worden voorzien. Makkelijker zou het zijn als de expressie direct kan worden ingetikt. De simpelste manier om een datastructuur te maken waar Haskell mee uit de voeten kan, is om de rekenkundige expressie in een string te zetten: `"x*x+1"`.

In een ander vak (Talen en Compilers) kun je leren hoe je een functie

```
ontleed :: String -> Expr
```

kunt schrijven, die zo'n string omzet in de overeenkomstige boomstructuur. Hier bekijken we het eenvoudigere, omgekeerde probleem: een functie weerg die een expressieboom weergeeft als string. Als beide functies beschikbaar zijn, dan kunnen we bijvoorbeeld de differentieerfunctie afg "inpakken" zodat het een functie tussen strings wordt:

```
afgel              :: String -> String -> String
afgel exprstr dx = weerg (afg (ontleed exprstr) dx)
```

De ingepakte functie is eenvoudiger te gebruiken dan de oorspronkelijke functie afg. Een sessie kan er bijvoorbeeld als volgt uitzien:

```
? afgel "x*x+1" "x"
x*1+1*x+0
```

Dit gebruik van de symbolische expressiemanipulatiefuncties is natuurlijk eenvoudiger dan

```
? afg (Var "x" :*: Var "x" :+: Con 1.0) "x"
Var "x":*:Con 1:+:Con 1:*:Var "x":+:Con 0
```

De functie weerg werkt op expressiebomen, en wordt daarom met zes patronen gedefinieerd voor alle constructoren van *Expr*.

```
weerg              :: Expr -> String
weerg (Con n)  = show n
weerg (Var x)  = x
weerg (a :+: b) = "(" ++ weerg a ++ "+" ++ weerg b ++ ")"
weerg (a :-: b) = "(" ++ weerg a ++ "-" ++ weerg b ++ ")"
weerg (a :*: b) =        weerg a ++ "*" ++ weerg b
weerg (a :/: b) =        weerg a ++ "/" ++ weerg b
```

De functie show is een standaardfunctie die een stringrepresentatie geeft van onder andere *Float* waarden. De functie weerg wordt waar nodig recursief aangeroepen; de resulterende strings worden samengevoegd tot één lange string, waarin ook nog het symbool voor de betreffende operator wordt opgenomen. Bij het optellen en aftrekken worden er ook nog haakjes in het resultaat gezet, anders zou de boomexpressie (*Var* `"a"` :+: *Var* `"b"`) :*: (*Var* `"c"` :+: *Var* `"d"`) worden omgezet in de string `"a+b*c+d"`, die de verkeerde interpretatie heeft.

Opgaven

8.1 Breid de datadeclaratie van *Expr* uit zodat er vier nieuwe expressievormen ontstaan: de sinus van een expressie, de cosinus van een expressie, de exponentiële functie (e tot de macht een expressie), en de natuurlijke logaritme van een expressie. Breid vervolgens de definitie van afg uit voor deze vier nieuwe expressievormen. Verwerk daarin de "kettingregel" voor het differentiëren: $(f \circ g)' = (f' \circ g) * g'$.

8.2 Schrijf een functie norep die gegeven een *Stat* alle *Repeat*-statements daaruit verwijdert, door ze te vervangen door een equivalente *While*-statements.

8.3 Kun je een functie:

```
foldExpr :: (Int -> a)
         -> (String  -> a)
         -> (a -> a -> a)
         -> (a -> a -> a)
         -> (a -> a -> a)
         -> (a -> a -> a)
         -> Expr    -> a
```

schrijven, met behulp waarvan je de functie weerg dan definieert?

8.4 Schrijf een functie die nagaat of een propositie een tautologie is, d.w.z. dat hij voor elke mogelijk toekenning van waarheidswaarden aan de in de expressie voor komende variabelen *True* oplevert. Gebruik eventueel de module *Data.Set*.

8.5 (*)[1] We extend and simply our expression data type in order to represent functions and function applications. We furthermore have a single built-in operator for concatenating the only basic values (*String*) we have.

```
data Expr = Con String   -- a constant string
          | Var String   -- the use of a variable
          | Concat       -- the only basic operation
          | Apply  Expr  Expr
          | Lambda String Expr
```

[1]The exercises with a (*) tend to be challenging.

Now write a function eval :: *Expr* -> *Expr*, which simplifies an expression until its top-level constructor is not an Apply node. Do not try to write efficiënt code! You may want to ignore the so-called variable capture problem in a first approach.

9 Case study: Huffman-codering

Bij het verzenden van grote hoeveelheden gegevens, streven we ernaar die gegevens zo efficiënt mogelijk te coderen. Een populaire methode om dat te bewerkstelligen is gebruik te maken van zogenaamde *Huffman-coderingen*. In deze case study presenteren we een Haskell-implementatie van deze coderingen.

Een Huffman-codering beeldt elk symbool in een te verzenden boodschap af op een specifieke bitreeks. In tegenstelling tot conventionele coderingstechnieken kunnen de lengtes van deze bitreeksen onderling verschillen. Door symbolen die vaak in de boodschap voorkomen af te beelden op korte bitreeksen en symbolen die minder vaak voorkomen af te beelden op langere bitreeksen kan een kortere codering voor de totale boodschap bewerkstelligd worden dan wanneer alle symbolen afgebeeld worden op bitreeksen van gelijke lengte.

9.1 Prefixvrijheid

Een potentieel probleem van het afbeelden op bitreeksen van verschillende lengtes doet zich voor wanneer een gecodeerde boodschap terugvertaald moet worden naar de oorspronkelijke symboolreeks. Bij een afbeelding op bitreeksen van gelijke lengte volstaat het immers om de gecodeerde boodschap op te delen in blokken van bits die in grootte overeenkomen met een enkel symbool in de oorspronkelijke boodschap en deze blokken vervolgens één voor één (of zelfs parallel) te decoderen; het opdelen van de gecodeerde boodschap heeft echter meer voeten in de aarde als de bitreeksen voor individuele symbolen van verschillende lengte zijn.

Huffman-coderingen omzeilen dit probleem door af te dwingen dat de afbeelding van symbolen op bitreeksen *prefixvrij* is. Dat wil zeggen dat voor elk symbool geldt dat geen van de prefixen van de bitreeks waarop het afgebeeld wordt, gelijk is aan een van de bitreeksen waarop de overige symbolen afgebeeld worden.

Voorbeeld. Als we gebruikmaken van de volgende codering,

$$\begin{aligned} \texttt{e} &\mapsto \texttt{00} \\ \texttt{x} &\mapsto \texttt{01} \\ \texttt{t} &\mapsto \texttt{1}, \end{aligned}$$

en de boodschap `text` dus coderen als `100011`, dan maakt prefixvrijheid dat de gecodeerde boodschap zich eenvoudig laat decoderen tot de orginele symboolreeks. Immers, de enige

bitreeks in het bereik van de codering die begint met `1` is de reeks `1` zelf. Verder zijn er zijn weliswaar twee bitreeksen die beginnen met `0`, maar de enige reeks waarin `0` gevolgd wordt door `0` is `00`. Evenzo, de enige reeks waarin `0` gevolgd wordt door `1` is `01`. De gecodeerde boodschap `100011` laat zich dus fragmenteren in de reeksen `1`, `00`, `01` en `1`, welke met behulp van de coderingstabel eenvoudig zijn om te zetten naar de symbolen `t`, `e`, `x` en `t` van de oorspronkelijke boodschap. ∎

Voorbeeld. De volgende codering,

$$\begin{aligned} \texttt{e} &\mapsto \texttt{0} \\ \texttt{x} &\mapsto \texttt{01} \\ \texttt{t} &\mapsto \texttt{1}, \end{aligned}$$

is niet prefixvrij: een van de prefixen van de bitreeks `01`, waarop het symbool `x` wordt afgebeeld, is `0` en dus gelijk aan de afbeelding van het symbool `e`. Het belang van prefixvrijheid wordt duidelijk als we deze niet-prefixvrije codering loslaten op de boodschap `text`. De op die manier verkregen codering `10011` is *ambigu*. Ze kan op twee manieren opgedeeld worden, maar slechts één opdeling leidt bij terugvertaling tot de oorspronkelijke boodschap:

(1) de opdeling `1`, `0`, `0`, `1` en `1` geeft de decodering `teett` en

(2) de opdeling `1`, `0`, `01` en `1` geeft de decodering `text`. ∎

9.2 Optimale coderingen

Stel dat een boodschap opgebouwd is uit een verzameling symbolen S (het *alfabet*) en dat voor elk symbool $s \in S$ de frequentie in de boodschap gegeven is door een natuurlijk getal f_s.

Voorbeeld. Voor de boodschap `text` hebben we het alfabet

$$S = \{\texttt{t}, \texttt{e}, \texttt{x}\}$$

en de frequenties $f_{\texttt{t}} = 2$, $f_{\texttt{e}} = 1$ en $f_{\texttt{x}} = 1$. ∎

Ons doel is nu om een prefixvrije codering af te leiden die elk symbool $s \in S$ afbeeldt op een bitreeks van lengte ℓ_s, zodanig dat

$$\sum_{s \in S} f_s \ell_s \qquad (*)$$

minimaal is.

9.3 Frequenties

In onze Haskell-implementatie van Huffman-codering zullen we boodschappen eenvoudigweg representeren als lijsten van lettertekens:

```
type Symbol  = Char
type Message = [Symbol].
```

Gebruikmakend van een tweetal functies uit de standaardmodule *Data*.List,

```
sort  :: (Ord a) => [a] -> [a]
group :: (Eq a)  => [a] -> [[a]],
```

waarbij group de functie is die opeenvolgende gelijken in een lijst samenvoegt, als in bijvoorbeeld

```
group "onmiddellijk" ⇝ ["o","n","m","i","dd","e","ll","i","j","k"],
```

kunnen we nu het alfabet en de frequenties van een boodschap bepalen met

```
freqs :: Message -> [(Symbol, Int)]
freqs = map freq . group . sort
  where
    freq syms@(sym : _) = (sym, length syms).
```

Voor de boodschap `text` hebben we bijvoorbeeld

```
sort "text" ⇝ "ettx"
```

en

```
group "ettx" ⇝ ["e","tt","x"],
```

en dus

```
freqs "text" ⇝ [('e',1),('t',2),('x',1)].
```

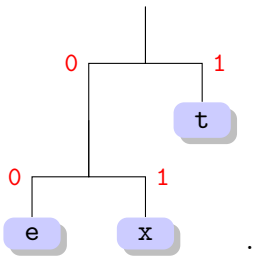
Merk op dat de gegeven implementatie van de functie freqs nog ruimte voor verbetering biedt: na het sorteren van de boodschap wordt een groepering opgebouwd, die vervolgens meteen weer afgebroken wordt om de lengte van de groepen te bepalen. Het is iets efficiënter om deze operaties samen te voegen tot een enkele gang over de lijst met behulp van een lokale functie collate:

```
freqs :: Message -> [(Symbol, Int)]
freqs = collate . sort
  where
    collate []                          = []
    collate [sym]                       = [(sym, 1)]
    collate (sym1 : syms@(sym2 : _))
      | sym1 == sym2                    = let (sym, n) : frqs = collate syms
                                          in  (sym, n + 1) : frqs
      | otherwise                       = (sym1, 1) : collate syms.
```

9.4 Huffman-bomen

Gegeven het alfabet en de frequenties van een boodschap kunnen we nu een zogenaamde *Huffman-boom* construeren waaruit eenvoudig af te lezen is hoe symbolen op bitreeksen worden afgebeeld.

Een voorbeeld van zo'n Huffman-boom, voor de boodschap `text` met alfabet {`t`, `e`, `x`}, is



.

De bitreeks die bij een bepaald symbool hoort wordt gevonden door het pad te volgen dat loopt vanaf de wortel van de boom tot het met het betreffende symbool gemarkeerde blad: elke vertakking die naar links gevolgd wordt, komt overeen met een bit `0`; elke vertakking die naar rechts gevolgd wordt, komt overeen met een bit `1`. In bovenstaande boom, bijvoorbeeld, leidt het pad van de wortel tot aan het met `x` gemarkeerde blad langs een vertakking naar links gevolgd door een vertakking naar rechts; en zo vinden we als afbeelding voor `x` de bitreeks `01`.

In onze Haskell-implementatie zullen we Huffman-bomen representeren met waarden van het type *Tree* van binaire bomen met lettertekens in de bladeren,

```
data Tree = Leaf Symbol | Node Tree Tree.
```

De hierboven afgebeelde boom laat zich dan schrijven als

```
Node (Node (Leaf 'e') (Leaf 'x')) (Leaf 't').
```

9.5 Coderen

Bitreeksen worden gerepresenteerd als lijsten van waarden van het type *Bit*:

```
data Bit  = Zero | One
type Bits = [Bit].
```

Gegeven een Huffman-boom kunnen we voor een symbool de bijbehorende bitreeks als volgt bepalen:

```
encode_Symbol :: Tree -> Symbol -> Bits
encode_Symbol  tree      sym    = case enc tree of
                                    Nothing -> undefined
                                    Just bits -> bits
  where
    enc (Leaf sym')
      | sym == sym' = Just []
      | otherwise   = Nothing
    enc (Node l r)  = case enc l of
                        Nothing -> case enc r of
                                     Nothing -> Nothing
                                     Just bits -> Just (One : bits)
                        Just bits -> Just (Zero : bits).
```

Of, iets korter, zonder tussenkomst van het *Maybe*-type,

```
encode_Symbol :: Tree -> Symbol -> Bits
encode_Symbol  tree      sym    = enc tree id undefined
  where
    enc (Leaf sym') prefix bits
      | sym == sym'  = prefix []
      | otherwise    = bits
    enc (Node l r)  prefix bits = enc l (prefix . (Zero:)) (enc r (prefix . (One:)) bits)
```

en zelfs

```
encode_Symbol :: Tree -> Symbol -> Bits
encode_Symbol  tree      sym    = enc tree id undefined
  where
    enc (Leaf sym') prefix
      | sym == sym'  = const (prefix [])
      | otherwise    = id
    enc (Node l r)  prefix = enc l (prefix . (Zero:)) . enc r (prefix . (One:)).
```

Merk op dat als het te coderen symbool niet in de boom voorkomt, we de speciale foutwaarde undefined ("ongedefinieerd") opleveren (dit levert dan een run-time exceptie

op). Een goed alternatief is het produceren van een nette foutboodschap (met behulp van de functie error).

Het coderen van een complete boodschap is nu eenvoudig:

```
encode :: Tree -> Message -> Bits
encode  tree  = concatMap . encode_Symbol
```

9.6 Boomconstructie

Wat rest is het construeren van een Huffman-boom op basis van een gegeven frequentietabel. Om de boom zo te construeren dat de resulterende codering optimaal is, passen we het volgende algoritme toe:

(1) we slaan elk symbool op in een boom die uit alleen een blad bestaat;

(2) we voegen de twee bomen met de laagste totale frequentie samen;

(3) we herhalen stap 2 totdat er nog één boom over is.

Figuur 9.1 toont hoe met behulp van bovenstaand algoritme de Huffman-boom voor de boodschap `onmiddellijk` geconstrueerd wordt; dat wil zeggen, de boom voor het symboolalfabet $\{\mathtt{o}, \mathtt{n}, \mathtt{m}, \mathtt{i}, \mathtt{d}, \mathtt{e}, \mathtt{l}, \mathtt{j}, \mathtt{k}\}$ en de frequenties

$$f_{\mathtt{o}} = f_{\mathtt{n}} = f_{\mathtt{m}} = f_{\mathtt{e}} = f_{\mathtt{j}} = f_{\mathtt{k}} = 1$$

en

$$f_{\mathtt{i}} = f_{\mathtt{d}} = f_{\mathtt{l}} = 2.$$

In deelfiguur 9.1(a) is te zien hoe voor elk symbool een blad aangemaakt is. In deelfiguur 9.1(b)–(i) worden vervolgens telkens de twee bomen met het laagste totale frequentie samengevoegd tot een nieuwe boom. In elk van de deelfiguren zijn de bomen gesorteerd op oplopend frequentietotaal en zijn alle knopen en bladen gemarkeerd met het bijbehorende subtotaal.

De volgende Haskell-functie levert een redelijk directe implementatie van het algoritme:

```
huffman :: [(Symbol, Int)] -> Tree
huffman  frqs              = let frqs' = sortBy (compare `on` snd) frqs
                             in  huff [(Leaf sym, n) | (sym, n) <- frqs']
  where
    huff []                        = undefined
    huff [(tree, _)]               = tree
    huff ((l, n_l) : (r, n_r) : pairs) =
      huff (insertBy (compare `on` snd) (Node l r, n_l + n_r) pairs)
```

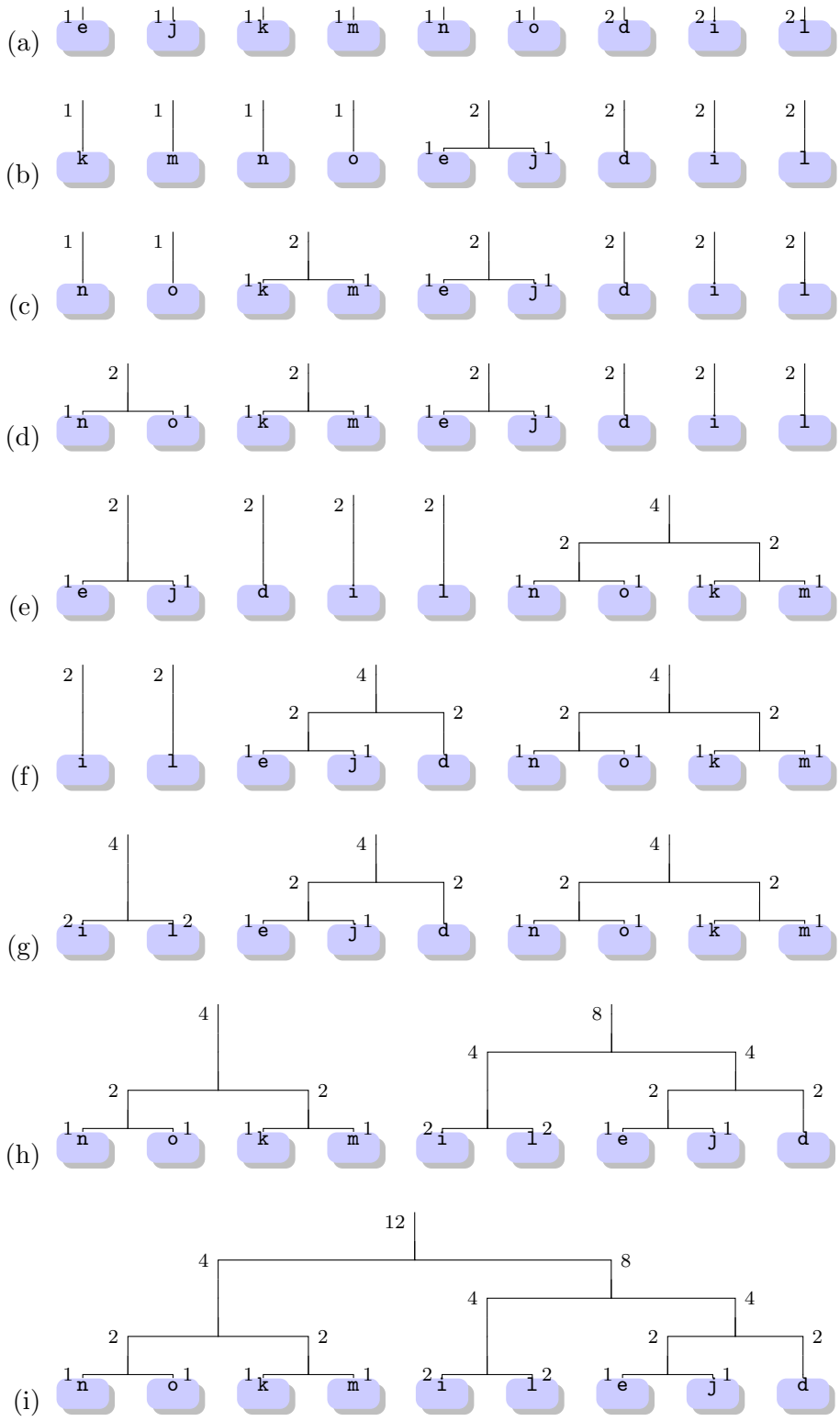


Fig. 9.1: Constructie van een Huffman-boom

De functie huffman begint het bouwen van de boom met het op frequentie sorteren van de paren in de frequentietabel frqs. Daarvoor wordt gebuikgemaakt van een variatie op de functie sort, namelijk

$$\mathsf{sortBy} :: (a \mathrel{->} a \mathrel{->} \mathit{Ordering}) \mathrel{->} [a] \mathrel{->} [a],$$

die ook gedefinieerd is in de standaardmodule List en lijsten sorteert op basis van een gegeven ordening voor paren van elementen. Deze ordening heeft de vorm van een functieparameter die twee elementen als argument neemt en een waarde van het in de Prelude gedefinieerde type *Ordering*,

$$\textbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT},$$

als resultaat oplevert: LT als het eerste element volgens de ordening kleiner is dan het tweede, EQ als de twee elementen volgens de ordening gelijk zijn en GT als het eerste element volgens de ordening groter is dan het tweede. Voor elk type in de klasse *Ord* is zo'n ordening beschikbaar via de Prelude-functie compare:

$$\mathsf{compare} :: (\mathit{Ord}\ a) \Rightarrow a \mathrel{->} a \mathrel{->} \mathit{Ordering}.$$

De ordening op de paren in de frequentietabel wordt gemaakt door een hulpfunctie on,

$$\begin{array}{l}\mathsf{on} :: (b \mathrel{->} b \mathrel{->} c) \mathrel{->} (a \mathrel{->} b) \mathrel{->} a \mathrel{->} a \mathrel{->} c\\ \mathsf{on}\quad (\oplus)\qquad\quad \mathsf{f}\qquad\quad \mathsf{x}\qquad \mathsf{y}\ =\ \mathsf{f\,x} \oplus \mathsf{f\,y},\end{array}$$

los te laten op de functies compare en snd.

Van alle paren $(\mathsf{sym}, \mathsf{n})$ die in de gesorteerde frequentietabel frqs' voorkomen, wordt het symbool sym opgeslagen in een blad en wordt de aldus verkregen boom *Leaf* sym getupeld met de frequentie n. De lokale functie huff neemt nu telkens de eerste twee elementen $(\mathsf{l}, \mathsf{n_l})$ en $(\mathsf{r}, \mathsf{n_r})$ van de lijst van boomfrequentieparen en voegt ze samen tot een nieuwe boom *Node* l r, die getupeld wordt met zijn frequentietotaal $\mathsf{n_l} + \mathsf{n_r}$. Dit nieuwe boom-frequentiepaar wordt op de juiste plaats in de lijst pairs van overgebleven bomen en frequenties opgeslagen. Hiervoor gebruiken we de functie

$$\mathsf{insertBy} :: (a \mathrel{->} a \mathrel{->} \mathit{Ordering}) \mathrel{->} a \mathrel{->} [a] \mathrel{->} [a]$$

die, net als sortBy, een ordeningsfunctie als argument neemt. Op de op deze manier verkregen lijst van paren wordt vervolgens weer de functie huff aangeroepen. Als er nog slechts één boomfrequentiepaar over is, levert huff de eerste component van dit paar op: dit is de Huffman-boom voor de gegeven frequentietabel.

Je kunt bewijzen dat een met de functie huffman verkregen boom inderdaad overeenkomt met een minimale prefixvrije symbool-voor-symboolcodering van een gegeven alfabet met de daarbij horende frequenties.

Opgaven

Opgave 1. Schrijf een functie

encoding :: *Message* -> *Bits*

die een boodschap codeert in een minimale prefixvrije symbool-voor-symboolcodering.

Opgave 2. Schrijf een functie

cost :: [(*Symbol*, *Int*)] -> *Tree* -> *Int*

die, gegeven een frequentietabel, de kosten van een Huffman-boom bepaalt (cf. de sommatie (∗) op pagina 180).

Opgave 3. Schrijf een functie

decode :: *Tree* -> *Bits* -> *Message*

die, gegeven een Huffman-boom, en een gecodeerde boodschap de oorspronkelijke boodschap produceert. (Hint: definieer een functie decodes :: *Tree* -> *Bits* -> (*Symbol*, *Bits*) die een enkel symbool decodeert en dat symbool samen met de overgebleven bits oplevert.)

Opgave 4. Geef een definitie voor de functie group (pagina 181).

Opgave 5.

(i) Geef een definitie voor de functie sortBy (pagina 186).

(ii) Geef een definitie van de functie sort :: (*Ord a*) => [*a*] -> [*a*] die gebruikmaakt van sortBy.

(iii) Geef een definitie voor de functie insertBy (pagina 186).

(iv) Geef een definitie van de functie insert :: (*Ord a*) => *a* -> [*a*] -> [*a*] die gebruikmaakt van insertBy.

Opgave 6.

(i) Geef een type voor de lokale functie freq (pagina 181).

(ii) Geef een type voor de lokale functie collate (pagina 182).

(iii) Geef een type voor de lokale functie huff (pagina 184).

Opgave 7.

(i) Geef voor elk van de drie gegeven definities van de functie encode_{Symbol} het type van de gebruikte lokale functie enc.

(ii) Geef een definitie van de functie encode_{Symbol} die gebruikmaakt van een lokale functie enc van type *Tree* -> [*Bits*]. Geef ook een definitie van zo'n lokale functie enc.

Opgave 8. De functie encode_{Symbol} levert een foutwaarde op als het te encoderen symbool niet in de Huffman-boom voorkomt. Implementeer een nieuwe versie van encode_{Symbol}, maar ditmaal van type *Tree* -> *Symbol* -> *Maybe Bits*, die, afhankelijk van of het symbool wel of niet in de boom voorkomt, respectievelijk een *Just*- of een *Nothing*-waarde oplevert. Pas de functie encode overeenkomstig aan, dat wil zeggen schrijf een versie van type *Tree* -> *Message* -> *Maybe Bits*.

Opgave 9. De functie huffman is ongedefinieerd voor lege frequentietabellen. Breid de definitie van het type *Tree* uit met een constructor voor lege bomen en gebruik deze nieuwe definitie van *Tree* in een functie huffman die ook gedefinieerd is op lege tabellen.

Opgave 10. Hierboven hebben we ervoor gekozen om boodschappen te representeren als lijsten van lettertekens, maar kunnen onze implementatie zonder al te veel moeite zo aanpassen dat we ook boodschappen bestaande uit bijvoorbeeld gehele getallen kunnen coderen.

(i) Pas de implementatie zo aan dat boodschappen kunnen bestaan uit symbolen van een willekeurig type *a* uit de klasse *Ord*:

type *Message a* = [*a*].

(ii) Pas de implementatie zo aan dat boodschappen kunnen bestaan uit symbolen van een willekeurig type uit de klasse *Eq*.

10 Klassen en hun instanties

10.1 Numerieke types

10.1.1 Overloading

Veel functies en operatoren kunnen op waarden van verschillend type worden toegepast. Er zijn twee mechanismen waardoor dat mogelijk is:

- *Polymorfie*. Een polymorfe functie werkt op een bepaalde datastructuur (bijvoorbeeld lijsten), zonder gebruik te maken van eigenschappen van de elementen. De functie kan dus op datastructuren met een willekeurig elementtype worden toegepast. Voorbeelden van polymorfe functies: length, concat en map.
- *Overloading*. Een overloaded functie kan op een aantal verschillende types werken met voor ieder een eigen onafhankelijke implementatie. De operator + werkt bijvoorbeeld zowel op type *Int* als op type *Float*. De operator <= kan op *Int* en *Float* werken, en daarnaast ook op *Char*, tweetupels en lijsten.

In paragraaf 2.5.5 is aangegeven dat overloading mogelijk is dankzij het bestaan van klassen van types (*classes*). Een klasse is een groep types waarop een bepaalde operator kan worden toegepast. De types *Int* en *Float* zitten beide in *Num*, de klasse van numerieke types. De operator + is op alle types in de klasse *Num* gedefinieerd. Dit komt tot uiting in het type van de operator +: p. 52

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

De tekst *Num a* kan gelezen worden als "type *a* zit in klasse *Num*". Het geheel heeft de betekenis: "+ heeft het type $a \rightarrow a \rightarrow a$ mits *a* in klasse *Num* zit".

Andere klassen die veel gebruikt worden zijn *Eq* en *Ord*. De klasse *Eq* is de klasse van types waarvan de elementen vergeleken kunnen worden; *Ord* is de klasse van ordenbare types. Operatoren die op types uit deze klassen gedefinieerd zijn, zijn bijvoorbeeld:

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(<=) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Let op het verschil tussen de pijltjes: het pijltje met enkele stok kan meer dan eens voorkomen in een type, en wordt gebruikt in de betekenis "functie van... naar...". Het pijltje met dubbele stok kan maar één keer voorkomen in een typedeclaratie. Links ervan staat vermeld dat een bepaalde typevariabele in een bepaalde klasse zit, rechts ervan staat een type waar deze typevariabele in gebruikt wordt.

10.1.2 Classes en instances

Het is mogelijk om zelf nieuwe klassen te definiëren, naast de reeds bestaande klassen *Num*, *Eq* en *Ord*. Ook is het mogelijk om nieuwe types toe te voegen aan een klasse (zowel aan de drie bestaande klassen als aan zelfgedefinieerde).

Het definiëren van een klasse heet een *klassedeclaratie* (*class declaration*), het toevoegen van een type aan een klasse een *instantiedeclaratie* (*instance declaration*). De drie standaardklassen zijn niet ingebouwd in Haskell. Ze worden in de prelude gedefinieerd door middel van gewone klassedeclaraties. Ook het feit dat *Int* en *Float* deel uitmaken van de klasse *Num* wordt in de prelude gedefinieerd door middel van instantiedeclaraties. Er is dus niets speciaals aan de drie standaardklassen.

De definitie van de klasse *Num* is een goed voorbeeld van een klassedeclaratie. Deze ziet er, iets vereenvoudigd, als volgt uit:

```
class Num a where
  (+), (−), (∗), (/) :: a -> a -> a
  negate             :: a -> a
```

Een klassedeclaratie bestaat dus uit de volgende onderdelen:

- het (speciaal voor dit doel) gereserveerde woord **class**;
- de naam van de klasse (*Num* in het voorbeeld);
- een typevariabele (*a* in het voorbeeld);
- het gereserveerde woord **where**;
- typedeclaraties voor operatoren en functies, waarbij de genoemde typevariabele gebruikt mag worden.

In een instantiedeclaratie wordt voor de aldus gedefinieerde operatoren een definitie gegeven. De instantiedeclaratie waarmee wordt aangegeven dat *Int* in de klasse *Num* zit ziet er als volgt uit:

```
instance Num Int where
  (+)    = primPlusInt
  (−)    = primMinusInt
  (∗)    = primMulInt
  (/)    = primDivInt
  negate = primNegInt
```

Een instantiedeclaratie bestaat dus uit de volgende onderdelen:

- het gereserveerde woord **instance**;
- de naam van een klasse (*Num* in het voorbeeld);
- een type (*Int* in het voorbeeld);
- het gereserveerde woord **where**;
- definities voor de operatoren en functies die in de klassedeclaratie werden gedeclareerd.

Bij de instantiedeclaratie *Num Int* zijn de functiedefinities een beetje flauw, omdat simpelweg wordt aangegeven dat voor elke functie de betreffende ingebouwde functie op integers genomen moet worden. De instantiedeclaratie *Num Float* (uit te spreken als "*Float* is een instance van *Num*") is al even flauw:

```
instance Num Float where
  (+)    = primPlusFloat
  (−)    = primMinusFloat
  (∗)    = primMulFloat
  (/)    = primDivFloat
  negate = primNegFloat
```

Het leuke van deze declaraties is wel, dat alleen de functies prim... ingebouwd zijn; de operatoren +, ∗ enz. zijn, compleet met hun overloading, in de prelude gewoon in Haskell gedefinieerd.

10.1.3 Nieuwe numerieke types

Als je zelf een type hebt gedefinieerd, waarop de numerieke operatoren zouden moeten werken, dan kan het nieuwe type met een instantiedeclaratie lid gemaakt worden van de klasse *Num*. In paragraaf 4.3.3 werd bijvoorbeeld het type *Ratio* der rationale getallen p. 110
gedefinieerd (de verzameling **Q**). Rationale getallen werden opgeteld met de functie qPlus, vermenigvuldigd met qMaal, enzovoort. Maar het is natuurlijk veel handiger om optelling tussen *Ratio*'s gewoon als + te kunnen schrijven. Daartoe dient de volgende instantiedeclaratie:

```
instance Num Ratio where
  (+)    = qPlus
  (−)    = qMin
  (∗)    = qMaal
  (/)    = qDeel
  negate = qMin (0, 1)
```

In plaats van de functies eerst qPlus, qMaal enz. te noemen, kan de functiedefinitie ook direct in de instantiedeclaratie geschreven worden. Vaak worden de instantiedeclaraties direct na de typedeclaratie geschreven, zoals hieronder:

```
type Ratio      = (Int, Int)

instance Num Ratio where
  (x, y) + (p, q) = eenvoud (x ∗ q + y ∗ p, y ∗ q)
  (x, y) − (p, q) = eenvoud (x ∗ q − y ∗ p, y ∗ q)
  (x, y) ∗ (p, q) = eenvoud (x ∗ p, y ∗ q)
  (x, y) / (p, q) = eenvoud (x ∗ q, y ∗ p)
  negate (x, y)   = (negate x, y)
```

Het is overigens verstandig om geen "kale" tupels tot instance van een klasse te maken, maar ze te beschermen met een beschermd datatype (zie paragraaf 6.3). Je gebruikt p. 130 daartoe **data** in plaats van **type**, en patronen in de definitie van de functies:

```
data Ratio                = Rat (Int, Int)
instance Num Ratio where
  Rat (x, y) + Rat (p, q) = eenvoud (Rat (x * q + y * p, y * q))
  Rat (x, y) - Rat (p, q) = eenvoud (Rat (x * q - y * p, y * q))
  Rat (x, y) * Rat (p, q) = eenvoud (Rat (x * p, y * q))
  Rat (x, y) / Rat (p, q) = eenvoud (Rat (x * q, y * p))
  negate (Rat (x, y))     = Rat (negate x, y)
```

Aan de hand van de typering bepaalt de interpreter welke versie van de operatoren gebruikt moet worden. Bij de operator $*$ op het type *Ratio* redeneert de interpreter ongeveer als volgt:

> Dit is een definitie van de operator $*$. Volgens de klassedeclaratie van *Num* heeft die operator het type $a \to a \to a$, waarbij a het type is van een instance van *Num*. In deze instantiedeclaratie is dat *Ratio*. Dus de parameters van $*$ zijn in deze definitie van type *Ratio*. Een *Ratio* is een (beschermd) tupel van twee integers. Dus x, y, p en q hebben het type *Int*. Volgens de definitie moeten x*p en y*q uitgerekend worden. Eens kijken, kan $*$ toegepast worden op integers? Ja, want *Int* behoort volgens de definitie in de prelude ook tot de klasse *Num*. Dan weet ik dus ook hoe x en p vermenigvuldigd moeten worden...

Dankzij de typering ziet de interpreter dus dat x*p niet een recursieve aanroep is van het vermenigvuldigen van *Ratio*'s, maar dat hier sprake is van het gebruik van de operator $*$ uit één van de andere instances van *Num*.

10.1.4 Numerieke constanten

Door het klassemechanisme kan voor optelling de operator + gebruikt worden, ongeacht of de op te tellen waardes integers zijn, *Float*'s, of zelfgedefinieerde numerieke types, zoals *Ratio* of *Complex*. Lastig is echter, dat het voor de notatie van constanten wèl belangrijk is wat het gewenste type is. Zo moet voor de waarde "drie" geschreven worden:

met het type *Int*:	3
met het type *Float*:	3.0
met het type *Ratio*:	*Rat* $(3, 1)$
met het type *Complex*:	*Comp* $(3.0, 0.0)$

Als bijvoorbeeld is gedefinieerd: half = *Rat* $(1, 2)$, dan kun je niet schrijven:

```
3 * half
```

De waarde half heeft immers het type *Ratio*, terwijl 3 het type *Int* heeft.

Dit is vooral vervelend bij het definiëren van functies die op alle types in een klasse moeten kunnen werken. Het is bijvoorbeeld wel mogelijk om een overloaded functie verdubbel te schrijven, maar een functie halveer lukt niet. Het enige wat er op zou zitten is om hier verschillende versies van te maken:

```
verdubbel    ::  Num a => a -> a
verdubbel x  =   x + x

halveerInt   ::  Int -> Int
halveerInt n =   n / 2

halveerFloat ::  Float -> Float
halveerFloat x = x / 2.0
```

enzovoort. Een oplossing zou kunnen zijn om de functie halveer in de klasse *Num* te specificeren, en in elke instance te definiëren. Maar dan kan je wel aan de gang blijven, want waarom wel een functie halveer maar geen functie deelInVieren?

Om het probleem beter op te lossen, is er in de prelude voor gekozen om nog één functie aan de klasse *Num* toe te voegen: de functie fromInteger. De volledige klassedeclaratie luidt dus:

```
class Num a where
  (+), (−), (∗), (/) :: a -> a -> a
  negate             :: a -> a
  fromInteger        :: Int -> a
```

Hiermee wordt gespecificeerd dat er voor elke instantie van *Num* een conversiefunctie moet zijn van *Int* naar dat type. Die conversiefunctie moet gedefinieerd worden in de instantiedeclaratie. Voor het type *Int* is dat gemakkelijk:

```
instance Num Int where
  ...
  fromInteger n = n
```

Voor het type *Float* zit er niets anders op dan een ingebouwde functie te gebruiken

```
instance Num Float where
  ...
  fromInteger = primIntToFloat
```

Voor zelfgedefinieerde types is het echter wel mogelijk om fromInteger zonder prim-magie te definiëren, bijvoorbeeld:

```
instance Num Ratio where
  ...
  fromInteger n = Rat (n, 1)
```

Functies zoals halveer kunnen nu gedefinieerd worden door:

```
halveer  :: Num a => a -> a
halveer x = x / fromInteger 2
```

Omdat het in de praktijk tamelijk vervelend is om op iedere getalconstante de functie fromInteger toe te passen, is het in Haskell mogelijk om dit automatisch te laten doen. Nadat aan de interpreter de opdracht `:set +i` is gegeven (zie paragraaf 2.2.3), wordt p. 31 voortaan op elke constante van type *Int* direct de functie fromInteger toegepast. Daarmee wordt wel een raar mengsel van "ingebouwde" en "voorgedefinieerde" faciliteiten gebruikt: de functie fromInteger is in de prelude netjes in Haskell gedefinieerd, maar het automatisch toepassen ervan op elke *Int*-constante is een niet in Haskell definieerbaar, en daarom ingebouwd mechanisme.

Handig is het wel. Functies die op *Float*'s werken, zoals sqrt, lijken nu ook op constanten van type *Integer* te kunnen werken:

```
? sqrt 2
1.41421
```

"Lijken te werken", want wat er eigenlijk uitgerekend wordt is sqrt (fromInteger 2).

10.2 Ordening en gelijkheid

10.2.1 Defaultdefinities

In de prelude wordt een klasse *Eq* gedefinieerd. De instances van deze klasse zijn de types waarvan de elementen met elkaar vergeleken kunnen worden. De klassedeclaratie is als volgt:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

Bij elke instantiedeclaratie voor deze klasse moeten dus de operatoren == (gelijkheid) en /= (ongelijkheid) worden gedefinieerd. De vier standaardtypes *Int*, *Float*, *Char* en *Bool* worden in de prelude alle als instance van *Eq* gedefinieerd:

```
instance Eq Int where
  x == y = primEqInt x y
  x /= y = not (x == y)

instance Eq Float where
  x == y = primEqFloat x y
  x /= y = not (x == y)
```

```
instance Eq Char where
  x == y          = ord x == ord y
  x /= y          = not (x == y)

instance Eq Bool where
  True == True  = True
  False == False = True
  True == False = False
  False == True = False
  x /= y          = not (x == y)
```

Waarden van type *Int* en *Float* worden vergeleken door aanroep van een ingebouwde functie. Characters worden vergeleken door hun ISO/ASCII-codes te vergelijken met de zojuist gedefinieerde operator == op integers. Gelijkheid op *Bool*'s tenslotte wordt direct door middel van patronen gedefinieerd.

In alle vier de gevallen wordt ongelijkheid (/=) gedefinieerd door het resultaat van == om te keren met not. De definitie is in alle gevallen precies hetzelfde (behalve natuurlijk dat telkens de == uit een andere instance wordt gebruikt). Dit soort definities mag ook reeds in de klassedeclaratie worden gezet. Het is dan niet nodig om ze in iedere instance te herhalen. Zo'n definitie van een operator heet een *default*-definitie: een definitie die bij ontbreken van een definitie in de instantiedeclaraties wordt gebruikt. Wordt een functie waarvoor een defaultdefinitie bestaat tòch in de instantiedeclaratie gedefinieerd, dan gaat die definitie voor.

De klassedeclaratie *Eq* zoals die werkelijk in de prelude staat is dus als volgt:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y       = not (x == y)
```

De definitie van /= in de instantiedeclaraties is weggelaten, omdat de defaultdefinitie voldoet.

Ook zelfgedefinieerde types kunnen tot instance van *Eq* gemaakt worden. Voor het type *Ratio* kan de gelijkheidsdefinitie uit opgave 6.18 gebruikt worden: p. 115

```
instance Eq Ratio where
  Rat (x, y) == Rat (p, q) = x * q == y * p
```

De gelijkheid die in de rechterkant van de definitie gebruikt wordt is de gelijkheid tussen integers. De Haskell-interpreter kan dat afleiden aan de hand van de typering. Een definitie van /= kan, net als in de instantiedeclaraties in de prelude, achterwege blijven. De defaultdefinitie is ook in dit geval immers bruikbaar.

10.2.2 Klassen met voorwaarden

De types waarvan de elementen ordenbaar zijn (met operatoren zoals <=) zijn instances van de klasse *Ord*. In de klassedeclaratie voor *Ord* wordt aangegeven dat een type ook een instance van *Eq* moet zijn, wil het ordenbaar zijn:

```
class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

Alle operatoren en functies in deze klasse met uitzondering van <= hebben een default-definitie. De enige operator die in instantiedeclaraties gedefinieerd hoeft te worden is dus <=. Het is vanwege deze defaultdefinities dat geëist wordt dat instances van *Ord* ook instances van *Eq* zijn. In de defaultdefinitie van < wordt namelijk de operator /= gebruikt. De defaultdefinities luiden:

```
x < y          = x <= y && x/= y
x >= y         = y <= x
x > y          = y < x

max x y | x >= y = x
        | y >= x = y
min x y | x <= y = x
        | y <= x = y
```

De instantiedeclaratie *Ord Int* en *Ord Float* doen voor de definitie van <= een beroep op een ingebouwde functie. Voor characters wordt de ordening bepaald door de integerordening van hun ISO/ASCII codes:

```
instance Ord Char where
  x <= y = ord x <= ord y
```

Net als *Ord* eist de klassedeclaratie voor *Num* in de prelude dat de instances ook een instance van de klasse *Eq* zijn. De, nu helemaal complete klassedeclaratie voor *Num* luidt derhalve:

```
class Eq a => Num a where
  (+), (-), (*), (/) :: a -> a -> a
  negate             :: a -> a
  fromInteger        :: Int -> a
```

De vergelijkbaarheid van de elementen van instances van *Num* wordt echter niet gebruikt in defaultdefinities, zoals dat bij *Ord* het geval was. De enige reden dat *Eq a* wordt geëist als voorwaarde voor *Num a* is dat "numerieke types", waarvan de elementen niet eens vergelijkbaar zijn, als onzinnig beschouwd worden.

10.2.3 Instances met voorwaarden

Ook bij instantiedeclaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een bepaalde klasse. Deze constructie wordt o.a. gebruikt om in één keer alle denkbare lijsten tot instance van *Eq* te maken:

```
instance Eq a => Eq [a] where
  [] == []         = True
  [] == (y : ys)   = False
  (x : xs) == []   = False
  (x : xs) == (y : ys) = x == y && xs == ys
```

De eerste regel van deze instantiedeclaratie kan zo gelezen worden: "als a een instance is van *Eq*, dan is ook $[a]$ een instance van *Eq*". Het vierde geval in de definitie van == is opmerkelijk: aan de rechterkant komt twee maal een aanroep van == voor. De eerste stelt gelijkheid op (de eerste) elementen van de lijst voor (dat kan, want dat was immers de voorwaarde van de instantiedeclaratie). De tweede aanroep van == is een recursieve aanroep van gelijkheid op lijsten.

Dankzij deze declaratie kunnen lijsten van integers vergeleken worden, lijsten van floats, lijsten van characters en lijsten van booleans. Maar ook lijsten van lijsten van integers (want lijsten van integers zijn ondertussen ook vergelijkbaar). En daarom dus ook lijsten van lijsten van lijsten van integers, enzovoort...

In paragraaf 4.1.4 werd al opgemerkt dat lijsten een ordening kennen: de lexicografische ordening. Deze ordening wordt gedefinieerd door een instantiedeclaratie in de prelude: p. 89

```
instance Ord a => Ord [a] where
  [] <= ys         = True
  (x : xs) <= []   = False
  (x : xs) <= (y : ys) = x < y || (x == y && xs <= ys)
```

Met deze definitie wordt aangegeven dat de lege lijst de kleinste lijst is. Voor niet-lege lijsten is het eerste element bepalend; als het eerste element van de twee lijsten gelijk is, wordt de rest van de lijsten recursief vergeleken. De andere ordeningsoperatoren hoeven niet gedefinieerd te worden: daarvoor worden de defaultdefinities gebruikt.

Een instantiedeclaratie kan meer dan één voorwaarde hebben. Die moeten dan tussen haakjes genoteerd worden, met komma's ertussen. Hiermee kan bijvoorbeeld de gelijkheid van tweetupels gedefinieerd worden, zoals in de prelude gebeurt:

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x, y) == (u, v) = x == u && y == v
```

De elementen van een tweetupel (a, b) zijn dus vergelijkbaar mits zowel a als b een instance is van *Eq*. Twee tweetupels zijn volgens deze definitie alleen maar gelijk, als beide elementen gelijk zijn (volgens de gelijkheidsdefinitie van hun respectievelijke types).

In de prelude wordt alleen een definitie gegeven van de gelijkheid van tweetupels. Drie- en meertupels zijn niet vergelijkbaar. In voorkomende gevallen kan zo'n gelijkheid natuurlijk wel zelf gedefinieerd worden.

Het pijltje met dubbele stok (=>) kan gebruikt worden in typedeclaraties, in instantiedeclaraties en in klassedeclaraties. Let op het verschil in betekenis van deze drie vormen:

- f :: *Num a* => *a* -> *a* is een typedeclaratie: f is een functie met type *a* -> *a* mits *a* een type is in de klasse *Num*.
- **instance** *Eq a* => *Eq* [*a*] is een instantiedeclaratie: [*a*] is een instance van *Eq* mits *a* dat ook is.
- **class** *Eq a* => *Ord a* is een klassedeclaratie: alle instances van de nieuwe klasse *Ord* moeten ook instances zijn van *Eq*.

10.2.4 Standaardklassen

In de prelude worden de volgende klassen gedefinieerd:

- *Eq*, de klasse van vergelijkbare types,
- *Ord*, de klasse van ordenbare types,
- *Num*, de klasse van numerieke types,
- *Enum*, de klasse van opsombare types,
- *Ix*, de klasse van indextypes,
- *Show*, de klasse van afdrukbare types, en
- *Read*, de klasse van types die naar een *String* geconverteerd kunnen worden.

De eerste drie werden al eerder besproken. Hieronder volgt een korte beschrijving van de anderen.

De klasse *Enum*

De klasse *Enum* is als volgt gedefinieerd:

```
class Ord a => Enum a where
  enumFrom         :: a -> [a]
  enumFromThen     :: a -> a -> [a]
  enumFromTo       :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

De bedoeling van de functie enumFrom is dat de lijst van waarden "vanaf" een bepaalde waarde wordt opgeleverd. De functie enumFromThen krijgt een beginwaarde en een tweede waarde; de functie enumFromTo krijgt een beginwaarde en een eindwaarde; de functie enumFromThenTo tenslotte krijgt ze alledrie.

Standaardinstanties van deze klasse zijn *Int*, *Float* en *Char*. Een paar voorbeelden van het resultaat van de functies voor verschillende types:

```
enumFrom 4                  =  [4,5,6,7,8,...
enumFromTo 'c' 'f'          =  ['c','d','e','f']
enumFromThenTo 1.0 1.5 3.0  =  [1.0,1.5,2.0,2.5,3.0]
```

Deze vier functies worden door de interpreter gebruikt om de speciale notaties [x..], [x,y..], [x..y] en [x,y..z] uit te rekenen, die in paragraaf 4.1.1 werden besproken. Deze notaties kunnen dus evenals de vier functies gebruikt worden voor verschillende types, bijvoorbeeld ['c'..'f']. Zou je zelf types definiëren als instance van *Enum*, dan kan ook voor die types de notatie [x..y] gebruikt worden. p. 81

De instantiedeclaratie *Enum Int* luidt als volgt:

```
instance Enum Int where
  enumFrom n       = iterate (1+)       n
  enumFromThen n m = iterate ((m - n)+) n
```

Voor het type *Float* is de instantiedeclaratie hetzelfde, maar dan met 1.0 in plaats van 1. Voor characters luidt de declaratie:

```
instance Enum Char where
  enumFrom c       = map chr (enumFrom (ord c))
  enumFromThen c d = map chr (enumFromThen (ord c) (ord d))
```

De in de definitie gebruikte functies zijn natuurlijk geen recursieve aanroepen, maar de overeenkomstige functies van de *Int*-instance.

Voor de functies enumFromTo en enumFromThenTo is er een defaultdefinitie in de klassedeclaratie:

```
class Ord a => Enum a where
  ...
  enumFromTo n m = takeWhile (m >=) (enumFrom n)
  enumFromThenTo n n' m
    | n' > n       = takeWhile (m >=) (enumFromThen n n')
    | otherwise    = takeWhile (m <=) (enumFromThen n n')
```

Omdat in deze definities de ordeningsoperatoren gebruikt worden, is het noodzakelijk dat elke instance van *Enum* ook een instance is van *Ord*. Instances van *Enum* hoeven echter niet noodzakelijk een instance van *Num* te zijn. Dat de instantiedeclaraties voor *Int* en *Float* de operator + gebruiken is hun zaak; er zijn instances van *Enum* denkbaar die geen numerieke operatoren nodig hebben (*Char* is daar een voorbeeld van).

De klasse *Ix*

De klasse *Ix* lijkt op *Enum*. De klassedeclaratie is als volgt:

```
class Ord a => Ix a where
  range   :: (a,a) -> [a]
  index   :: (a,a) -> a -> Int
  inRange :: (a,a) -> a -> Bool
```

De functie range is vergelijkbaar met enumFromTo. Er wordt nu echter ook een functie index gevraagd, die het rangnummer van een waarde in een bepaald interval geeft. Daarom kan het "continue" type *Float* geen instance zijn van *Ix*. De "discrete" types *Int* en *Char* zijn wel een instance van *Ix*, bijvoorbeeld:

```
instance Ix Int where
  range   (m,n)   = [m..n]
  index   (m,n) i = i - m
  inRange (m,n) i = m <= i && i <= n
```

De klassen *Show* en *Read*

De klasse *Show* introduceert functies die een waarde naar een *String* kunnen converteren. Deze worden veelal gebruikt om waarden in de terminal te kunnen afdrukken.

```
class Show a where
  showsPrec :: Int -> a      -> ShowS
  show      ::        a      -> String
  showList  ::        [a]    -> ShowS

  showsPrec _ x  s = show x ++ s
  show x           = shows x ""
  showList ls    s = showList__ shows ls s

showList__ :: (a -> ShowS) -> [a] -> ShowS
showList__ _     []     s  = "[]" ++ s
showList__ showx (x:xs) s = '[' : showx x (showl xs)
  where showl []     = ']' : s
        showl (y:ys) = ',' : showx y (showl ys)
```

Als je de behoefte hebt aan een speciale stringrepresentatie van je datatypes, zul je je eigen show willen definiëren. Meestal worden de functies van *Show* automatisch voor je afgeleid.

Om de omgekeerde weg te bewandelen levert de klasse *Read* de functie:

```
read :: Read a => String -> a
```

Het schrijven van *Show*, *Read*, maar ook van *Eq* instanties wordt al snel een geestdodende aangelegenheid. In veel gevallen kun je dit aan de compiler over laten.

```
data Person = APerson String String Int
  deriving (Eq, Show, Read)
```

zal een aantal defaultdefinities genereren voor bovenstaand type. In veel gevallen voldoen deze aan wat je wil. Het voordeel is dat je op deze manier zonder zelf iets te hoeven doen waarden van het type APerson in `ghci` kunt laten zien. Dat kan handig zijn tijdens het debuggen. Als je niet tevreden bent met het gedrag van de automatisch afgeleide definities, zul je ze zelf moeten schrijven.

10.2.5 Problemen met klassen

Bij het analyseren van een expressie of een file met definities kan de interpreter een aantal fouten melden die te maken hebben met het gebruik van klassen. Hieronder worden drie soorten fouten besproken.

"Cannot derive instance"

Deze foutmelding is het gevolg als je een operator uit een klasse gebruikt met parameters waarvoor er geen instance is gedeclareerd. Bijvoorbeeld:

```
? (1,2,3) == (4,5,6)
ERROR: Cannot derive instance in expression
*** Expression        : (1,2,3) == (4,5,6)
*** Required instance : Eq (Int,Int,Int)
```

In de prelude staat geen declaratie waarmee drietupels tot instance van *Eq* gemaakt worden (wel voor tweetupels en lijsten). Daarom kan de operator == niet zonder meer op drietupels toegepast worden. Een oplossing van dit probleem is de ontbrekende instantiedeclaratie aan het programma toe te voegen.

Deze foutmelding wordt ook gegeven als je functies probeert te vergelijken. Functietypes zijn immers geen instance van *Eq*:

```
? tail == drop 1
ERROR: Cannot derive instance in expression
*** Expression        : tail == drop 1
*** Required instance : Eq ([a]->[a])
```

Dat wij met de technieken uit sectie 13 zelf in staat zijn om de gelijkheid van twee functies te bewijzen, wil nog niet zeggen dat deze functies ook in de taal Haskell vergeleken mogen worden. De interpreter zou in zo'n geval immers de twee functies op alle mogelijke parameters moeten toepassen (wat oneindig lang duurt) of een inductief bewijs moeten leveren (waar hij niet creatief genoeg voor is). p. 234

"Overlapping instances"

Het is niet mogelijk om twee declaraties te geven waarmee een type instance wordt van dezelfde klasse. Bij gebruik van een operator op een waarde van dat type zou de interpreter dan namelijk niet kunnen kiezen uit de twee definities.

Hetzelfde probleem treedt op als het type in een instantiedeclaratie een speciaal geval is van een type waarvoor al een andere instantiedeclaratie bestaat. Bijvoorbeeld: in de prelude worden tweetupels gedeclareerd als instance van *Eq*:

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)$ **where**
$(\mathsf{x}, \mathsf{y}) == (\mathsf{u}, \mathsf{v}) = \mathsf{x} == \mathsf{u}\ \&\&\ \mathsf{y} == \mathsf{v}$

Als rationale getallen als tweetupel van twee integers worden gedefinieerd, zou je daarop wel een andere gelijkheid willen definiëren:

type $Ratio = (Int, Int)$

instance $Eq\ Ratio$ **where**
$(\mathsf{x}, \mathsf{y}) == (\mathsf{u}, \mathsf{v}) = \mathsf{x} * \mathsf{v} == \mathsf{u} * \mathsf{y}$

Bij een aanroep van $(1, 2) == (2, 4)$ kan de interpreter nu niet kiezen: wordt de standaard tupelgelijkheid bedoeld of de *Ratio*-gelijkheid? Bij het analyseren van de instantiedeclaratie wordt daarom een foutmelding gegeven:

```
ERROR "file" (line 12): Overlapping instances for class "Eq"
*** This instance   : Eq (Int,Int)
*** Overlaps with   : Eq (a,a)
*** Common instance : Eq (Int,Int)
```

Dit probleem kan worden opgelost door types waar een "rare" gelijkheid op gedefinieerd moet worden als beschermd type te definiëren, dus met gebruikmaking van **data** in plaats van **type**:

data $Ratio = Rat\ (Int, Int)$

Dan kan *Ratio* rustig tot instance van *Eq* gemaakt worden, omdat *Ratio* een ander type is dan (Int, Int).

"Unresolved overloading"

Bij het gebruik van een overloaded operator besluit de interpreter op grond van de types van de parameters welke definitie gekozen moet worden. Zo wordt in $1 + 2$ de integer-versie van $+$ gebruikt, en in $1.0 + 2.0$ de float-versie. Maar als de parameters zelf het resultaat zijn van een overloaded functie, kan het zijn dat er meerdere mogelijkheden zijn. Dat is bijvoorbeeld het geval in de volgende expressie:

```
? fromInteger 1 + fromInteger 2
ERROR: Unresolved overloading
*** type        : Num a => a
```

Voor fromInteger kan de integer-versie of de float-versie gekozen worden. In het eerste geval moet ook de integer-versie van $+$ gebruikt worden, in het tweede geval de float-versie. Als er verder geen context is waardoor de keuze gemaakt kan worden (bijvoorbeeld de hele expressie is parameter van de functie sqrt), dan volgt er een *unresolved overloading* foutmelding.

Deze foutmelding treedt vooral op als de optie "pas fromInteger toe op elk getal" aan staat (zoals beschreven in paragraaf 10.1.4). Een onschuldig ogende expressie als $1 + 2$ heeft dan immers al een unresolved overloading tot gevolg. p. 192

Meestal is deze fout te herstellen door de interpreter een extra hint te geven over het gewenste type. Dat kan bijvoorbeeld door een typedeclaratie te geven voor de kritieke functies, of door de dubieuze expressie direct te typeren:

```
? fromInteger 1 + fromInteger 2 :: Int
3
```

10.3 Klassen en eigenschappen

De namen van de diverse klassen en de operatoren daarin suggereren dat die operatoren aan allerlei eigenschappen voldoen. Er is echter niemand die er op toeziet dat dit inderdaad het geval is. Er volgt bijvoorbeeld geen foutmelding als je zou definiëren:

```
type Bliep          = (Int, [Char])
instance Ord Bliep where
  (n, xs) <= (k, ys) = n + k == length ys
```

(om maar eens iets onzinnigs te noemen). Toch is het niet gangbaar om dit soort definities een "ordening" te noemen. Maar waarom is deze definitie niet "zinvol", en de toch ook niet voor de hand liggende definitie van $\leqslant$ op rationale getallen (zie hieronder) wel?

```
instance Ord Ratio where
  Rat (x, y) <= Rat (u, v) = x * v <= u * y
```

Het antwoord is: normaliter wordt er van uitgegaan dat operatoren zoals $\leqslant$ aan bepaalde eigenschappen voldoen. Van die eigenschappen wordt gebruik gemaakt in andere functies. Sorteerfuncties maken bijvoorbeeld gebruik van het feit dat als $x \leqslant y$ en $y \leqslant z$, dat dan ook $x \leqslant z$.

Eigenschappen waar operatoren in een klasse aan dienen te voldoen, kunnen vastgelegd worden in *wetten*. Deze wetten zouden als commentaar bij de klassedeclaratie toegevoegd kunnen worden. Helemaal mooi zou het zijn als Haskell voor elke instance zou controleren of hij aan de gegeven wetten voldoet. Helaas... dat is een beetje te veel gevraagd. Wel zou je als programmeur kunnen *bewijzen* dat de definities die je in een bepaalde instance geeft, aan de vereiste wetten voldoet. Zo'n bewijs kan dienen om de *correctheid* van een implementatie aan te tonen.[1]

[1] Er zijn enkele pogingen gedaan om een programmeertaal te ontwerpen waarbij wetten automatisch gecontroleerd worden. Hoewel er aardige resultaten zijn geboekt, heeft deze benadering nog niet tot grote doorbraken geleid.

Wil de operator == zijn naam "gelijkheidsoperator" waardig zijn, dan moet hij aan de volgende wetten voldoen (voor alle f, x, y en z):

reflexiviteit	er geldt $x = x$;
symmetrie	als $x = y$, dan $y = x$;
transitiviteit	als $x = y$ en $y = z$, dan $x = z$;
congruentie	als $x = y$, dan $f\ x = f\ y$.

De laatste wet geeft problemen als die inderdaad wordt geëist voor *alle* functies f. Gelijkheid op rationale getallen voldoet bijvoorbeeld niet aan de congruentiewet voor de functie gemeen:

$$\mathsf{gemeen}\ (\mathit{Rat}\ (\mathsf{t}, \mathsf{n})) = \mathsf{t} + \mathsf{n}$$

want hoewel $\mathit{Rat}\ (1,2) == \mathit{Rat}\ (2,4)$, is $\mathsf{gemeen}\ (\mathit{Rat}\ (1,2)) /= \mathsf{gemeen}\ (\mathit{Rat}\ (2,4))$. Als het om beschermde datatypes gaat, wordt de congruentieëis daarom meestal afgezwakt; de wet hoeft alleen maar te gelden voor een bepaalde verzameling functies en combinaties daarvan. Bij de rationale getallen zijn dat bijvoorbeeld qPlus, qMin, qMaal en qDeel. Voor functies zoals gemeen, die met patroonherkenning direct gebruik maken van de representatie van het datatype, hoeft de congruentiewet niet te gelden.

De wetten waar de ordeningsoperator <= aan pleegt te voldoen, zijn de volgende (voor alle x, y en z):

reflexiviteit	er geldt $x \leqslant x$;
antisymmetrie	als $x \leqslant y$ en $y \leqslant x$, dan $x = y$;
transitiviteit	als $x \leqslant y$ en $y \leqslant z$, dan $x \leqslant z$.

De in de vorige paragrafen gedefinieerde instances van *Ord* voldoen inderdaad aan deze drie wetten. Voor de types *Int* en *Float* is dat moeilijk na te gaan, omdat die ingebouwde operatoren gebruiken (je zou kunnen zeggen dat die per definitie aan deze wetten voldoen). Voor de zelfgedefinieerde instances, zoals de rationale getallen en de lexicografische ordening op lijsten, zijn de wetten inderdaad te bewijzen. Een ordening die aan deze wetten voldoet heet een *partiële ordening*. Het is namelijk niet nodig dat elk element van het type met elk ander element vergelijkbaar is. Daarom staat er in de defaultdefinitie van min en max in paragraaf 10.2.2 niet otherwise in de tweede regel. Als twee elementen p. 196 onderling niet geordend zijn, is de waarde van min en max ongedefinieerd.

Numerieke types moeten, willen ze met recht zo genoemd worden, voldoen aan de wetten uit paragraaf 13.8. De bewijzen in die paragraaf zijn in feite het bewijs dat het type Nat p. 256 een waarlijk numeriek type is. Naast deze wetten zijn er nog meer wetten waaraan numerieke types moeten voldoen. Bijvoorbeeld een wet die het gedrag van $-$ definiëert:

$-$ is de inverse van $+$ $\qquad y + (x - y) = x$

Een overeenkomstige wet voor de delingsoperator / geeft echter problemen:

/ is de inverse van $*$ $\qquad$ als $y \neq 0$ dan $y * (x/y) = x$

Deze wet is bijvoorbeeld niet geldig voor de ingebouwde deling op gehele getallen. Boven-

dien is er sprake van een getal 0, en wat moeten we daarvoor nemen in een potentieel numeriek type?

In Haskell zijn alle numerieke types een instance van de klasse *Num*. De klassedeclaratie en de instantiedeclaraties die daarvoor nodig zijn staan in de prelude, en werden in paragraaf 10.1.4 besproken. Voor een precieze beschrijving van het onderscheid tussen verschillende numerieke types is een indeling in meerdere klassen eigenlijk geschikter. p. 192

10.4 The *Functor* class

In the latest Haskell definition a few more standard classes were introduced. Where the previous standard classes are intuitively rather self-evident, these new classes may seem at first less so. The main theme they address is to capture common control structures instead of common functions. The *Monad* type class is another example of a type of this kind. However, we devote the entire chapter 11 to the discussion of monads. p. 209

We have been using the function map :: (*a* -> *b*) -> [*a*] -> [*b*] quite a lot and one of the things one might wonder about is why there is no function map :: (*a* -> *b*) -> *Tree a* -> *Tree b* as well (and similarly for all other "container" types). Why should we only want to apply a function to all the elements in a list, and not to all the elements occurring in some other data type? Because the name map was used for a long time to apply only to lists, even in Haskell's predecessors, it was decided to call this overloaded function fmap and to introduce a class *Functor*. Although the term finds its origin in category theory, and has a broad scope, it suffices for the time being to think of it as a type constructor, which takes a type and returns a type.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Of course our list constructor is the first one to become a member of this class, since here the new name fmap is just our familiar map function:

```
instance Functor [] where
  fmap = map
```

The laws which should hold for all the instances of *Functor* that you declare are:

```
fmap id          = id
fmap f . fmap g = fmap (f . g)
```

Instead of defining a function mapTree as we have done in exercise 6.8 we can now make the data type *Tree* (and similarly all its variants) an instance of *Functor*: p. ??

```
data Tree a = Leaf a
            | Bin (Tree a) (Tree a)

instance Functor Tree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Bin l r)  = Bin (fmap f l) (fmap f r)
```

Yet another instance is the functor *Maybe* (note we already start calling *Maybe* a functor, instead of a type constructor?):

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing = Nothing
```

Now one might be inclined to think that only parameterised data types "containing" their parameter type can be declared to be an instance of the class *Functor*. The concept of a functor however goes quite bit further.

```
instance Functor IO where
  fmap f io = do v <- io
                 return (f v)
```

A suprising instance might be the following:

```
instance Functor ((->) c) where
  fmap = (.)
```

But what is this $(\rightarrow)\ c$? The answer is: a partially applied $\rightarrow$-constructor, which constructs a type which still expects the result type of the function type. Substituting $(\rightarrow)\ c$ in the type of fmap we get:

```
fmap :: (a -> b) -> (c ->) a -> (c ->) b
```

or, by removing the partial applications of $(\rightarrow)$:

```
fmap :: (a -> b) -> (c -> a) -> (c -> b)
```

But this is precisely the type of the function composition (.). Let us check whether the functor law holds for this definition:

```
fmap f . fmap g
 ==  {-definition of fmap -}
((.) f) . ((.) g)
 ==  {-applications of (.) -}
(f .) . (g .)
 ==  {-introduce extra parameter in order to be able to unfold (.) -}
\x -> ((f .) . (g .)) x
 ==  {-unfold (.) -}
\x -> (f .) (g . x)
 ==  {-application of partially applied (.) -}
\x -> f . (g . x)
 ==  {-associativity of (.) -}
\x -> (f . g) . x
 ==  {-partially unapplying (.) -}
\x -> ((f . g) .) x
 ==  {-removing the parameter x -}
((f . g) .)
 ==  {-folding definition of fmap -}
fmap (f . g)
```

Opgaven

10.1 Schrijf een declaratie waardoor de operatoren $+$, $-$, $*$ en / ook op complexe getallen (zie opgave 6.19) gebruikt kunnen worden. p. 115

10.2 Definieer een type *Set a* dat verzamelingen voorstelt met elementen uit *a*. Gebruik lijsten om dit type te implementeren. Definieer een functie

```
subset :: Set a -> Set a -> Bool
```

die controleert of een verzameling een deelverzameling is van een andere. Schrijf vervolgens een instantiedeclaratie waarmee *Set a* een instance van *Eq* wordt (met als voorwaarde dat *a* een instance is van *Eq*). Bedenk dat bij verzamelingen, anders dan bij lijsten, volgorde en verdubbelingen van elementen geen rol speelt. Waarom moet het type *Set a* gedefinieerd worden als beschermd datatype, dus met behulp van **data** en niet met **type**?

10.3 Definieer een klasse Finite ("eindig"). Deze klasse bezit geen operatoren en functies, maar wel een constante: de lijst van alle elementen van het type. Het is de bedoeling dat die lijst eindig is, vandaar de naam. Definieer de volgende types als instances van Finite:

- *Bool*;
- *Char*;

- (a, b), mits a en b eindig zijn;
- *Set* a (zoals gedefinieerd in de vorige opgave), mits a eindig is;
- $(a \mathrel{->} b)$, mits a en b eindig zijn en *Eq* a (moeilijk, voor de liefhebber).

Maak nu $(a \mathrel{->} b)$ tot instance van *Eq* mits a eindig is. Zijn er nog meer voorwaarden voor deze instantiedeclaratie?

10.4 Define instances of the class *Functor* for the types $(,)$ c and *Either* c.

10.5 The class Contravariant as defined in *Data.Functor*.Contravariant is defined as:

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
```

Just as functors come with a law which should hold for all instances, so does Contravariant.

```
contramap id              = id
contramap f . contramap g = contramap (g . f)
```

Now we show how the type constructor $(\mathrel{->} c)$ can be made an instance of Contravariant. Unfortunately this notation is not accepted by GHC, so we have to do some extra work. Complete the following code:

```
newtype Op b a = Op (a -> b)   -- note the order of the arguments.

instance Contravariant (Op c) where
  ...
```

10.6 The newtype Equivalence is defined as:

```
newtype Equivalence a = Equivalence (a -> a -> Bool)
```

Make this type an instance of Contravariant.

11 Monads: programming with effects

Credits: the text for this chapter was written by Graham Hutton (Version January 2014), and is reproduced here with his permission. I have omitted IO since we discuss that elsewhere in the notes, and I have omitted the further reading section.

11.1 Introduction

Shall we be pure or impure?

The functional programming community divides into two camps:

- "Pure" languages, such as Haskell, are based directly upon the mathematical notion of a function as a mapping from arguments to results.
- "Impure" languages, such as ML, are based upon the extension of this notion with a range of possible effects, such as exceptions and assignments.

Pure languages are easier to reason about and may benefit from lazy evaluation, while impure languages may be more efficient and can lead to shorter programs.

One of the primary developments in the programming language community in recent years (starting in the early 1990s) has been an approach to integrating the pure and impure camps, based upon the notion of a "monad". This chapter introduces the use of monads for programming with effects in Haskell.

Abstracting programming patterns

Monads are an example of the idea of abstracting out a common programming pattern as a definition. Before considering monads, let us review this idea, by means of two simple functions:

```
inc    :: [Int] -> [Int]
inc []  =      []
inc (n : ns) = n + 1 : inc ns

sqr    :: [Int] -> [Int]
sqr []  =      []
sqr (n : ns) = n ^ 2 : sqr ns
```

Both functions are defined using the same programming pattern, namely mapping the empty list to itself, and a non-empty list to some function applied to the head of the list and the result of recursively processing the tail of the list in the same manner. Abstracting this pattern gives the library function called map

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

using which our two examples can now be defined more compactly:

```
inc = map (+1)

sqr = map (^2)
```

11.2 A simple evaluator

Consider the following simple language of expressions that are built up from integer values using a division operator:

```
data Expr = Val Int | Div Expr Expr
```

Such expressions can be evaluated as follows:

```
eval          :: Expr -> Int
eval (Val n) =    n
eval (Div x y) = eval x `div` eval y
```

However, this function doesn't take account of the possibility of division by zero, and will produce an error in this case. In order to deal with this explicitly, we can use the *Maybe* type

```
data Maybe a = Nothing | Just a
```

to define a "safe" version of division

```
safediv :: Int -> Int -> Maybe Int
safediv n m = if m == 0 then Nothing else Just (n `div` m)
```

and then modify our evaluator as follows:

```
eval          :: Expr -> Maybe Int
eval (Val n) =    Just n
eval (Div x y) = case eval x of
                      Nothing -> Nothing
                      Just n -> case eval y of
                                     Nothing -> Nothing
                                     Just m -> safediv n m
```

As before, we can observe a common pattern, namely performing a case analysis on a value of a *Maybe* type, mapping Nothing to itself, and Just x to some result depending upon x. (Aside: we could go further and also take account of the fact that the case analysis is performed on the result of an eval, but this would lead to the more advanced notion of a monadic fold.)

How should this pattern be abstracted out? One approach would be to observe that a key notion in the evaluation of division is the sequencing of two values of a *Maybe* type, namely the results of evaluating the two arguments of the division. Based upon this observation, we could define a sequencing function

```
seqn                    :: Maybe a -> Maybe b -> Maybe (a, b)
seqn Nothing _          = Nothing
seqn _       Nothing    = Nothing
seqn (Just x) (Just y)  = Just (x, y)
```

using which our evaluator can now be defined more compactly:

```
eval (Val n) = Just n
eval (Div x y) = apply f (eval x `seqn` eval y)
  where f (n, m) = safediv n m
```

The auxiliary function apply is an analogue of application for *Maybe*, and is used to process the results of the two evaluations:

```
apply           :: (a -> Maybe b) -> Maybe a -> Maybe b
apply f Nothing  = Nothing
apply f (Just x) = f x
```

In practice, however, using seqn can lead to programs that manipulate nested tuples, which can be messy. For example, the evaluation of an operator Op with three arguments may be defined by:

```
eval (Op x y z) = apply f (eval x `seqn` (eval y `seqn` eval z))
  where f (a, (b, c)) = ...
```

11.3 Combining sequencing and processing

The problem of nested tuples can be avoided by returning to our original observation of a common pattern: "performing a case analysis on a value of a *Maybe* type, mapping *Nothing* to itself, and *Just* x to some result depending upon x". Abstracting this pattern directly gives a new sequencing operator that we write as $\gg\!=$, and read as "then":

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= f = case m of
            Nothing -> Nothing
            Just x -> f x
```

Replacing the use of case analysis by pattern matching gives a more compact definition for this operator:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= _ = Nothing
(Just x) >>= f = f x
```

That is, if the first argument is *Nothing* then the second argument is ignored and *Nothing* is returned as the result. Otherwise, if the first argument is of the form *Just* x, then the second argument is applied to x to give a result of type *Maybe b*.

The $\gg=$ operator avoids the problem of nested tuples of results because the result of the first argument is made directly available for processing by the second, rather than being paired up with the second result to be processed later on. In this manner, $\gg=$ integrates the sequencing of values of type *Maybe* with the processing of their result values. In the literature, $\gg=$ is often called "bind", because the second argument binds the result of the first. Note also that $\gg=$ is just apply with the order of its arguments swapped.

Using $\gg=$, our evaluator can now be rewritten as:

```
eval (Val n)   = Just n
eval (Div x y) = eval x >>= (\ n ->
                 eval y >>= (\ m ->
                 safediv n m))
```

The case for division can be read as follows: evaluate x and call its result value n, then evaluate y and call its result value m, and finally combine the two results by applying safediv. In fact, the scoping rules for lambda expressions mean that the parentheses in the case for division can freely be omitted.

Generalising from this example, a typical expression built using the $\gg=$ operator has the following structure:

```
m1 >>= \ x1 ->
m2 >>= \ x2 ->
...
mn >>= \ xn ->
f x1 x2 ... xn
```

That is, evaluate each of the expression $m1, m2, \ldots, mn$ in turn, and combine their result values $x1, x2, \ldots, xn$ by applying the function f. The definition of $\gg=$ ensures that such

an expression only succeeds (returns a value built using *Just*) if each mi in the sequence succeeds. In other words, the programmer does not have to worry about dealing with the possible failure (returning *Nothing*) of any of the component expressions, as this is handled automatically by the $\gg=$ operator.

Haskell provides a special notation for expressions of the above structure, allowing them to be written in a more appealing form:

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

Hence, for example, our evaluator can be redefined as:

```
eval (Val n)   = Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m
```

Exercises

11.1 Show that the version of eval defined using $\gg=$ is equivalent to our original version, by expanding the definition of $\gg=$.

11.2 Redefine seqn x y and eval (Op x y z) using the **do** notation.

11.4 Monads in Haskell

The do notation for sequencing is not specific to the *Maybe* type, but can be used with any type that forms a "monad". The general concept comes from a branch of mathematics called category theory. In Haskell, however, a monad is simply a parameterised type (or *type constructor*) m, together with two functions of the following types:

```
return :: a -> m a

(>>=) :: m a -> (a -> m b) -> m b
```

(Aside: the two functions are also required to satisfy some simple properties, but we will return to these later.) For example, if we take m as the parameterised type *Maybe*, return as the function *Just* :: a -> *Maybe* a, and $\gg=$ as defined before, then we obtain our first example, called the maybe monad.

In fact, we can capture the notion of a monad as a new class declaration. In Haskell, a class is a collection of types that support certain overloaded functions. For example, the class *Eq* of equality types can be declared as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x/= y = not (x == y)
```

The declaration states that for a type a to be an instance of the class *Eq*, it must support equality and inequality operators of the specified types. In fact, because a default definition has already been included for $/=$, declaring an instance of this class only requires a definition for $==$. For example, the type *Bool* can be made into an equality type as follows:

```
instance Eq Bool where
  False == False = True
  True == True  = True
  _     == _     = False
```

The notion of a monad can now be captured as follows:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

That is, a monad is a parameterised type "m" that supports return and $\gg=$ functions of the specified types. The fact that m must be a parameterised type, rather than just a type, is inferred from its use in the types for the two functions. Using this declaration, it is now straightforward to make *Maybe* into a monadic type:

```
instance Monad Maybe where
    -- return :: a -> Maybe a
  return x =        Just x

    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _ = Nothing
  (Just x) >>= f =  f x
```

(Aside: types are not permitted in instance declarations, but we include them in comments here for reference.) It is because of this declaration that the **do** notation can be used to sequence *Maybe* values. More generally, Haskell supports the use of this notation with any monadic type. Below, we give some further examples of types that are monadic, and the benefits that result from recognising and exploiting this fact.

11.4.1 The list monad

The maybe monad provides a simple model of computations that can fail, in the sense that a value of type *Maybe a* is either *Nothing*, which we can think of as representing failure, or has the form *Just* x for some x of type a, which we can think of as success.

The list monad generalises this notion, by permitting multiple results in the case of success. More precisely, a value of $[a]$ is either the empty list $[]$, which we can think of as failure, or has the form of a non-empty list $[\mathsf{x1}, \mathsf{x2}, \ldots, \mathsf{xn}]$ for some xi of type a, which we can think of as success. Making lists into a monadic type is straightforward:

```
instance Monad [] where
    -- return :: a -> [a]
  return x = [x]

    -- (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concat (map f xs)
```

(Aside: in this context, $[]$ denotes the list type $[a]$ without its parameter.) That is, return simply converts a value into a successful result containing that value, while $\gg\!=$ provides a means of sequencing computations that may produce multiple results: $\mathsf{xs} \gg \mathsf{f}$ applies the function f to each of the results in the list xs to give a nested list of results, which is then concatenated to give a single list of results.

As a simple example of the use of the list monad, a function that returns all possible ways of pairing elements from two lists can be defined using the **do** notation as follows:

```
pairs :: [a] -> [b] -> [(a, b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x, y)
```

That is, consider each possible value x from the list xs, and each value y from the list ys, and return the pair (x, y). It is interesting to note the similarity to how this function would be defined using the list comprehension notation:

```
pairs xs ys = [(x, y) | x <- xs, y <- ys]
```

In fact, there is a formal connection between the **do** notation and the comprehension notation. Both are simply different shorthands for repeated use of the $\gg\!=$ operator for lists. Indeed, the language Gofer that was one of the precursors to Haskell permitted the comprehension notation to be used with any monad. For simplicity however, Haskell only allows the comprehension notation to be used with lists.

11.4.2 The state monad

Now let us consider the problem of writing functions that manipulate some kind of state, represented by a type whose internal details are not important for the moment:

```
type State = ...
```

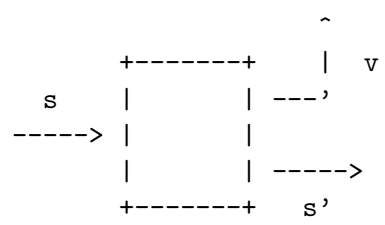
The most basic form of function on this type is a "state transformer" (abbreviated by ST), which takes the current state as its argument, and produces a modified state as its result, in which the modified state reflects any side effects performed by the function:

```
type ST = State -> State
```

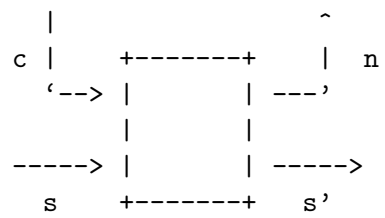
In general, however, we may wish to return a result value in addition to updating the state. For example, a function for incrementing a counter may wish to return the current value of the counter. For this reason, we generalise our type of state transformers to also return a result value, with the type of such values being a parameter of the ST type:

```
type ST a = State -> (a, State)
```

Such functions can be depicted as follows, where s is the input state, s′ is the output state, and v is the result value:



A state transformer may also wish to take argument values. However, there is no need to further generalise the ST type to take account of this, because this behaviour can already be achieved by exploiting currying. For example, a state transformer that takes a character and returns an integer would have type *Char* -> ST *Int*, which abbreviates the curried function type *Char* -> State -> (*Int*, State), depicted by:

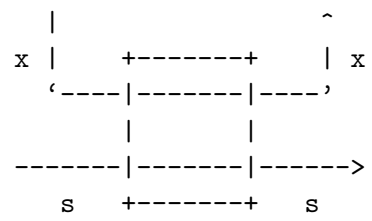


Returning to the subject of monads, it is now straightforward to make ST into an instance of a monadic type:

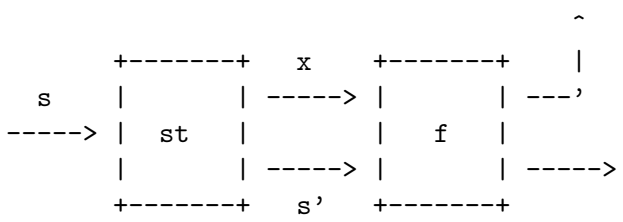
```
instance Monad ST where
   -- return :: a -> ST a
  return x = \s -> (x, s)

   -- (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f = \s -> let (x, s') = st s in f x s'
```

That is, return converts a value into a state transformer that simply returns that value without modifying the state:



In turn, $\gg\!=$ provides a means of sequencing state transformers: $\mathsf{st} \gg\!= \mathsf{f}$ applies the state transformer st to an initial state s, then applies the function f to the resulting value x to give a second state transformer $(\mathsf{f}\ \mathsf{x})$, which is then applied to the modified state s' to give the final result:



Note that return could also be defined by $\mathsf{return}\ \mathsf{x}\ \mathsf{s} = (\mathsf{x}, \mathsf{s})$. However, we prefer the above definition in which the second argument s is shunted to the body of the definition using a lambda abstraction, because it makes explicit that return is a function that takes a single argument and returns a state transformer, as expressed by the type $a \mathrel{->} \mathsf{ST}\ a$: A similar comment applies to the above definition for $\gg\!=$.

We conclude this section with a technical aside. In Haskell, types defined using the "**type**" mechanism cannot be made into instances of classes. Hence, in order to make ST into an instance of the class of monadic types, in reality it needs to be redefined using the "**data**" mechanism, which requires introducing a dummy constructor (called S for brevity):

```
data ST a = S (State -> (a, State))
```

It is convenient to define our own application function for this type, which simply removes the dummy constructor:

```
apply :: ST a -> State -> (a, State)
apply (S f) x = f x
```

In turn, ST is now defined as a monadic type as follows:

```
instance Monad ST where
    -- return :: a -> ST a
  return x = S (\ s -> (x, s))

    -- (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f  = S (\ s -> let (x, s') = apply st s in apply (f x) s')
```

Aside: the runtime overhead of manipulating the dummy constructor S can be eliminated by defining ST using the "**newtype**" mechanism of Haskell, rather than the "**data**" mechanism.

11.4.3 A state monad example

By way of an example of using the state monad, let us first define a type of binary trees whose leaves contains values of some type a:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Here is a simple example:

```
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

Now consider the problem of defining a function that labels each leaf in such a tree with a unique or "fresh" integer. This can be achieved by taking the next fresh integer as an additional argument to the function, and returning the next fresh integer as an additional result. In other words, the function can be defined using the notion of a state transformer, in which the internal state is simply the next fresh integer:

```
type State = Int
```

In order to generate a fresh integer, we define a special state transformer that simply returns the current state as its result, and the next integer as the new state:

```
fresh :: ST Int
fresh = S (\ n -> (n, n + 1))
```

Using this, together with the return and >>= primitives that are provided by virtue of ST being a monadic type, it is now straightforward to define a function that takes a tree as its argument, and returns a state transformer that produces the same tree with each leaf labelled by a fresh integer:

```
mlabel             :: Tree a -> ST (Tree (a, Int))
mlabel (Leaf x)    = do n <- fresh
                        return (Leaf (x, n))
mlabel (Node l r)  = do l' <- mlabel l
                        r' <- mlabel r
                        return (Node l' r')
```

Note that the programmer does not have to worry about the tedious and error-prone task of dealing with the plumbing of fresh labels, as this is handled automatically by the state monad.

Finally, we can now define a function that labels a tree by simply applying the resulting state transformer with zero as the initial state, and then discarding the final state:

```
label :: Tree a -> Tree (a, Int)
label t = fst (apply (mlabel t) 0)
```

For example, label tree gives the following result:

```
Node (Node (Leaf ('a', 0)) (Leaf ('b', 1))) (Leaf ('c', 2))
```

Exercises:

11.3 Define a function `app :: (State -> State) -> ST State`, such that fresh can be redefined by `fresh = app (+1)`.

11.4 Define a function `run :: ST a -> State -> a`, such that label can be redefined by `label t = run (mlabel t) 0`.

11.4.4 Some derived primitives

An important benefit of abstracting out the notion of a monad is that it then becomes possible to define a number of useful functions that work in an arbitrary monad. For example, the "map" function on lists can be generalised as follows:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f mx = do x <- mx
                return (f x)
```

Similarly, "concat" on lists generalises to:

```
join :: Monad m => m (m a) -> m a
join mmx = do mx <- mmx
              x <- mx
              return x
```

It is sometimes useful to sequence two monadic expressions, but discard the result value produced by the first:

```
(>>) :: Monad m => m a -> m b -> m b
mx >> my = do _ <- mx
              y <- my
              return y
```

For example, in the state monad the $\gg$ operator is just normal sequential composition, written as ; in most languages.

As a final example, we can define a function that transforms a list of monadic expressions into a single such expression that returns a list of results, by performing each of the argument expressions in sequence and collecting their results:

```
sequence    :: Monad m => [m a] -> m [a]
sequence []  =            return []
sequence (mx : mxs) = do x <- mx
                         xs <- sequence mxs
                         return (x : xs)
```

Exercises:

11.5 Define liftM and join more compactly by using $\gg=$.

11.6 Explain the behaviour of sequence for the maybe monad.

11.7 Define another monadic generalisation of map:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

11.8 Define a monadic generalisation of foldr:

```
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
```

11.5 The monad laws

Earlier we mentioned that the notion of a monad requires that the return and $\gg=$ functions satisfy some simple properties. The first two properties concern the link between return and $\gg=$:

```
return x >>= f = f x     -- (1)
mx >>= return = mx       -- (2)
```

Intuitively, equation (1) states that if we return a value x and then feed this value into a function f, this should give the same result as simply applying f to x. Dually, equation (2) states that if we feed the results of a computation mx into the function return, this should give the same result as simply performing mx. Together, these equations express — modulo the fact that the second argument to $\gg=$ involves a binding operation — that return is the left and right identity for $\gg=$.

The third property concerns the link between $\gg=$ and itself, and expresses (again modulo binding) that $\gg=$ is associative:

```
  (mx >>= f) >>= g
= (3)
  mx >>= (\ x -> (f x >>= g))
```

Note that we cannot simply write mx >>= (f >>= g) on the right hand side of this equation, as this would not be type correct.

As an example of the utility of the monad laws, let us see how they can be used to prove a useful property of the liftM function introduced earlier, namely that it distributes over the composition operator for functions, in the sense that:

```
liftM (f . g) = liftM f . liftM g
```

This equation generalises the familiar distribution property of map from lists to an arbitrary monad. In order to verify this equation, we first rewrite the definition of liftM using >>=:

```
liftM f mx = mx >>= \ x -> return (f x)
```

Now the distribution property can be verified as follows:

```
  (liftM f . liftM g) mx
= applying .
  liftM f (liftM g mx)
= applying the second liftM
  liftM f (mx >>= \ x -> return (g x))
= applying liftM
  (mx >>= \ x -> return (g x)) >>= \ y -> return (f y)
= equation (3)
  mx >>= (\ z -> (return (g z) >>= \ y -> return (f y)))
= equation (1)
  mx >>= (\ z -> return (f (g z)))
= unapplying .
  mx >>= (\ z -> return ((f . g) z)))
= unapplying liftM
  liftM (f . g) mx
```

Exercises:

11.9 Show that the maybe monad satisfies equations (1), (2) and (3).

11.10 Given the type

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
```

of expressions built from variables of type "a", show that this type is monadic by completing the following declaration:

```
instance Monad Expr where
    -- return :: a -> Expr a
  return x          = ...

    -- (>>=) :: Expr a -> (a -> Expr b) -> Expr b
  (Var a) >>= f     = ...
  (Val n) >>= f     = ...
  (Add x y) >>= f   = ...
```

Hint: think carefully about the types involved. With the aid of an example, explain what the $\gg=$ operator for this type does.

12 Basic IO

12.1 Introduction

As we have seen in the preface one of the distinguishing features of *pure* functional languages is the absence of side effects during the evaluation of a function call (that is what the word "pure" refers to), and thus especially the impossibility to change some global state. Thus far all functions we have seen when using the interpreter `ghci` just printed their result. But isn't this an *observable side-effect*? It is. Writing something to a screen and modifying something in the file system are both side effects which change the global state of the machine. Once the light emitted by the screen has been detected by our eyes and the information has been conveyed to our brain the world will never be the same again.

It is clear that a programming language in which we cannot manipulate the file system nor access other machines over the internet nor access data bases nor program games in, can hardly be called useful, and so the question addressed in this chapter is "How do we cope with input and output in a pure functional language".

Different pure functional languages (we will drop the word "pure" from now one, since when dealing with Haskell this is obvious) have taken different approaches to address this question. Despite the fact that they are not very different from a fundamental point of view, the solutions look differently in different langages and each comes with its specific style of programming.

There are two ways of presenting the solution to the issue of side effects; we can just present how things look in conventional Haskell programs and then explain the underlying structures, or we can gradually build up your understanding of how things are embedded inside Haskell, ending with how to use these stuctures eventually. In this chapter we take the second approach, since it also demonstrates a couple of basic programming techniques. This may make things look a bit more complicated than they are in everyday use, but prepares us better for understanding higher level abstractions.

First we show how the Haskell type system can help us to enforce the correct use of certain programming patterns. Having explained the basic principles we finally add some *syntactic sugar* in the form of the **do**-notation to make programs look nicer. In fact, they will look much like an imperative program. However, the type system of Haskell will enforce that we be honest about the side effects that we introduce in a program. Essentially, a side-effected computation that delivers the answer 5 will have a type that

is different from a pure expression that evaluates to 5.

12.2 Input and output

12.2.1 Modeling output

We start by showing how we can model textual output to a terminal in a functional language; the execution of the expression putChar 'a' displays the character 'a' in the terminal window (actually: append it to the standard output).

In imperative languages the expression putChar 'a' would be a command, with a side effect: the appearance of the character 'a' on the screen. Our first approach is to mimic this effect by looking upon putChar as a pure Haskell function, which takes as parameter the character to be printed and which constructs a list (actually in Haskell terms a new list) which represents the output. Since we will use the type *String* for different purposes we introduce a type synonym Output. This will make the rôles of the various components in the program more visible in the types. In this view the type and code of putChar may become:

```
type Output = String

putChar :: Char -> (Output -> Output)
putChar a output = output ++ [a]
```

In this example we have placed the Output -> Output part of the type between parentheses: it emphasizes that putChar 'a' as presented here *transforms* an Output value.

Now suppose we want to print the string "ab" using putChar 'a' and putChar 'b'. How do we compose the two printing commands?

```
... putChar 'a' ... putChar 'b' ...
```

A first attempt is:

```
putab output = putChar 'b' (putChar 'a' output)
```

or even:

```
putab = putChar 'b' . putChar 'a'
```

Unfortunately this does not look very natural since now the order in which the putChar calls appear in the program text is not the order in which the output is actually being produced. In order to alleviate this problem we introduce a new infix operator $\gg$, which executes both its arguments in the order in which they appear in the program text:

```
type O = Output -> Output

(≫) :: O -> O -> O
(≫) p q output = q (p output)
```

Using this operator we can now write:

```
putab = putChar 'a' ≫ putChar 'b'
```

If the Haskell designers hadn't used the semicolon ; for something else it would have made for a great binary operator since the operator ≫ precisely describes what a semicolon stands for: passing the final state of its left operand on to the right operand.

Once we have this operator it is only a small step to putStr a complete string to the output:

```
putStr :: String -> O
putStr ""       = id
putStr (s : ss) = putChar s ≫ putStr ss
```

Note that in the current definition we only get the final result once the commands have completed, and note furthermore that appending every next character to the end of the constructed string is very inefficient.

If we want to run this program we need a helper function which passes the empty string as initial value for the output:

```
runOutput p = p ""
result = runOutput (putStr "hello " ≫ putStr " world")
```

12.2.2 Modeling input

As a next step we shall model the input file. The situation is a bit more complicated here, because a function like getChar not only updates the input string by taking away its first element (e.g., by advancing a marker in the input), but it also has to make this first element available to the rest of the program; it does not make much sense to read a character and then throw it away immediately. We again introduce a type synonym for *String*s which act as input, and introduce a type I a for functions that transform the input (string), delivering a value of type a in the process:

```
type Input = String
type I a   = Input -> (a, Input)

getChar :: I Char
getChar (c : cs) = (c, cs)
```

Given the type I a, we cannot compose getChar operations with $\gg$, like we did with the calls to putChar, so we introduce a slightly more involved version of $\gg$, which we name $\gg=$. This function takes as its first parameter a function which takes an Input and returns the first character of the input and the unconsumed part. This character is passed as a parameter to the function that is its second argument and which continues to work on the unconsumed part:

```
(>>=) :: I b -> (b -> I a) -> I a
(ib >>= b2ia) input = let (b, input') = ib input
                      in  (b2ia b) input'
```

Using $\gg=$ and getChar it is now a straightforward exercise to write a function that reads characters from an Input up to the first newline character:

```
getLine :: I String
getLine = getChar >>= (\ c -> if c == '\n'
                              then (\ inp -> ("", inp))
                              else getLine >>= (\ cs -> return (c : cs)))

return ss = \ inp -> (ss, inp)
```

12.2.3 Modeling input and output at the same time

Now we have the functions putChar and getChar it seems we have the ingredients for writing a function that first reads a character and then prints it:

```
echo = getChar >>= (\ c -> putChar c)
```

Even shorter as:

```
echo = getChar >>= putChar
```

One can say that each of the functions we have presented worked on its own little piece of state, i.e. getChar operates on an input stream while advancing its read pointer by taking the tail of the Input and putChar updates an Output by appending characters to it. If we join these two streams together into a single value, and let each of the functions putChar and getChar work on its own component, while leaving the other part of the *World* untouched, we get precisely what we want. Since the new versions of getChar and putChar have the same type we make putChar return a value of type () to the rest of the computation. The type () contains exactly one value, denoted by () too. It is essentially a tuple with zero components, and it is often used to signal that something was computed, without the need to tell what the actual result was of the computation. It can therefore be likened to the role that `void` plays in many C-like languages.

We now use a type synonym to define the state *World* to be a pair that holds both the input and the output stream:

```
type World = (Input, Output)   -- the input and the output stream
type IO a  = World -> (a, World)
putChar :: Char -> IO ()
getChar ::         IO Char
putChar c  (ii,    oo) = ((), (ii, oo ++ [c]))
getChar    (i : ii, oo) = (i,  (ii, oo       ))
```

In the following definition we have used an identifier with the name b2ioa, which stands for a function of type "b to an IO of type a":

```
(>>=) :: IO b -> (b -> IO a) -> IO a
(iob >>= b2ioa) w = let (b, w') = iob w in b2ioa b w'

echo :: IO ()
echo = getChar >>= putChar
```

One way of looking at the Haskell main function now is as a function of type *World* -> *World*, which becomes a function from Input to Output when we run the program. We start with the empty output "" and when we are done we discard the unused part of the input using snd:

```
runIO :: (World -> World) -> Input -> Output
runIO prog input = snd (prog (input, ""))
```

12.2.4 The *IO* monad

We may conclude from the previous sections that it is nice to view all data in the outside world as a single large data structure of type *World* for which we have a collection of primitive (i.e. defined outside the standard libraries) functions to operate on. The Haskell language definition has the type *IO a* built-in as a primitive, which you may think of as being defined as:

```
type IO a = World -> (a, World)
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>)  :: IO a ->       IO b  -> IO b
```

The important thing here is that *IO* is an *opaque* type, which means that its internals are not visible. The direct consequence of this is that in our program we cannot get hold of values of this invisible type *World*. When the program is evaluated we always have a single value of this type, which is passed on as a "baton" in a relay race by the functions >> and >>=; only the function which "holds" the baton is allowed to perform input or

output actions. Since it is well defined in which order the baton is passed, the order in which these actions on the external world are executed is well-defined.

Another way of looking at this is to see a value of type *IO a* as a function which returns two kinds of values: a sequence of input/output actions which are implicitly executed and a value which can be used in the execution of the rest of the program. Once we take this view it should not come as a surprise that the main function in a Haskell program, i.e. the function which is actually evaluated when a Haskell program is executed, has type *IO* (). The result of evaluating this function is the combined result of all the actions stemming from this single call.

12.3 IO actions

The introduction of the type constructor *IO* has deep consequences for the structure of the program. As we said at the beginning, Haskell wants us to be honest in the side effects our program may have. If we want to compute a value of type, say, *Bool*, and we need side effects to take place during evaluation, then the code to compute it should have type *IO Bool* to reflect this. By choosing that option, we must use the operators $\gg=$ and $\gg$ to string expressions of this kind together. In the end, we shall have a main function (of *IO a* type for some *a*), of which the side-effects will be effectuated when the program is finally run. In the absence of side effects we can do without, and evaluate the code to compute the boolean value parts of the program.

Effectively, this means that when a function calls another function that returns of a value of type *IO a* for some *a*, it itself must also have a type *IO b* for some *b*. It is not allowed to forget that IO operations have taken place.[1]

12.3.1 Reading and writing a single character

We now come to the type of the functions getChar and putChar as they are defined in Haskell:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

We can now run these function from within `ghci`:

```
Prelude> 'K'
'K'
Prelude> putChar 'K'
KPrelude>
```

[1] To be honest, Haskell does provide operations for doing just that, but the use of these operations is very much frowned upon. We certainly do not want students using such functions in the assignments they deliver.

Notice that when we provide `ghci` with a non-*IO*-type value the result of this expression is printed as if it were an Haskell expression, whereas in the second case –where we provided `ghci` with a value of type *IO* ()– the effect of the actions is shown, but not the value which is passed on to the rest of the program. Hence `ghci` inspects the type of the expression given to it in order to decide what to do with it.

12.3.2 The `do` notation

Just as imperative programs contain many semicolons indicating sequential composition, our Haskell programs would contain a similar number of $\gg=$ and $\gg$ operations. This would soon lead to quite unreadable programs. In order to cope with this problem Haskell was extended with so-called **do**-notation, which serves as syntactic sugar for the calls to these primitive functions and the lambda-expressions we would need. As an example of its use we reformulate the echo function using **do**-notation:

```
echo :: IO ()
echo = do c <- getChar
          putChar c
```

The main advantage of this notation is that is is easy to see that the variable c in the above program contains both the result of the action getChar and serves as an input for the remainder of the action sequence. In the notation using $\gg=$ it was also a global variable, since the further occurrences of $\gg=$ were in its scope. Underneath it all, however, all uses of **do** are translated into $\gg=$ and $\gg$.

The **do**-notation follows similar layout rules as the **where**-clauses and the **let**-expressions. Whenever we have a line which indents precisely as far as the first statement of the sequence we start with a new *IO* expression.[2] The type of the complete **do**-sequence is the type of the last statement of the sequence. In the case above this is the type of the putChar c, and thus equals *IO* (). It is perfectly okay if other actions in the **do**-block have type *IO* t for some other type t. But they all have to have an *IO*-type.

12.3.3 Recursive actions

A function of type *IO a* (for some type *a*) may, just as any other function, be recursive. This allows us not only to print a single character, but also a whole sequence of characters:

```
putStr :: String -> IO ()
putStr (c : cs) = do putChar c
                     putStr  cs
```

[2]The precise layout rules of Haskell are complicated; for everyday use this formulation should suffice.

Now, you may have noticed that this definition is not complete yet, since the case for the empty string is still missing. What we need here is an empty effect. Fortunately, we have a special function available to deal with this case:

```
return :: a -> IO a
```

Given some value the function return constructs the empty side effect and returns the passed value on to the rest of the program. Our complete definition of putStr now becomes:

```
putStr :: String -> IO ()
putStr (c : cs) = do putChar c
                     putStr  cs
putStr []       = return ()
```

Although the use of the function return is quite similar to uses of the return statement in languages like Java and C++ there are differences too. In the aforementioned languages executing a return statement immediately makes execution return from the function at hand. In Haskell this is not the case; return just constructs a value of type $IO \ldots$ which, because it is an element in a sequence of statements in a **do**-construct, becomes part of a larger composite side-effect. The difference becomes clear by studying the following piece of code:

```
twoLetters' :: IO ()
twoLetters' = do putChar 'H'
                 _ <- return "this does not show up"
                 putChar 'i'
```

This function has precisely the same effect as putChar 'H' ≫ putChar 'i'. The call to return produces an empty side-effect and the value "this does not show up" gets bound to _, and is thus effectively discarded. In contrast to imperative languages, were the return statement would exit the twoLetters′ method, before putChar 'i' is executed.

The function putStr above is part of the prelude, just as its companion function putStrLn which places an end-of-line after the string has been printed.

12.3.4 Actions with results

Earlier we have seen a function getChar which reads a single character. We also have its companion function getLine available in the prelude, which reads a complete line from standard input, discarding the end-of-line character terminating that line:

```
getLine :: IO String
```

Some *IO* actions return a result that we want to use in the evaluation of the rest of a **do**-expression. In this case we use the following construct, where the result of executing the getLine command, is bound to the variable x:

```
x <- getLine
```

We use this construct in the following simple program, which asks for your name and prints a greeting:

```
groet :: IO ()
groet = do putStr "What is your name? "
           name <- getLine
           putStrLn ("Hello, " ++ name)
```

This closely resembles the assignment statement in imperative languages; we use however a left pointing arrow <- instead of an assignment symbol such as = or :=.

12.3.5 Actions on files

The functions we have seen thus far all deal with input and output from and to the standard channels. Of course the prelude also contains functions to read from and to write to the file system:

```
type FilePath = String
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

We conclude this section with a somewhat larger example. It tries to guess a natural number by repeatedly halving the interval in which the number has to lie following to the user's answers. We use a **case**-construct here, which is the Haskell-analogue of a structured switch statement (but without the breaks).

```
main :: IO ()
main = do
  putStrLn "Pick a number between 1 and 100"
  guess 1 100

guess :: Int -> Int -> IO ()
guess lower upper =
  let mid = (lower + upper) `div` 2
  in do putStrLn ("Is " ++ show mid ++ "g = greater, l = less, c = correct)"
        (a: _) <- getLine
        case a of
          'g' -> guess lower       (mid - 1)
          'l' -> guess (mid + 1) upper
          'c' -> putStrLn "Guessed"
          _   -> do putStrLn "Please type (g/l/c)!"
                    guess lower upper
```

12.4 Beyond Imperative Programming

Using **do**-notation, the part of the program which deals with input and output is not so different from what we are used to in imperative languages. If we however think a bit further we see that what we have is much more expressive: since values of type $IO\ldots$ are just values like any other values, we can: abstract from them, store them as part of other values, compute them, and pass them as as parameter. As you may have come to expect IO values are *first-class citizens*.

To see how we can exploit this let us take another look at the putStr function. The fact that a *String* is nothing else than a list of *Char*s and putStr merely applies putChar to each of the *String*'s elements raises the suspicion that it should be possible to implement putStr in terms of a map. What is the type of map putChar? Since putChar is of type $Char \to IO\ ()$, we have map putChar :: $String \to [IO\ ()]$. For executing a sequence of commands the prelude and the module *Control.Monad* both provide the function sequence_:

```
sequence_ :: [IO a] -> IO ()
sequence_ []       = return ()
sequence_ (x : xs) = x >> sequence_ xs
```

This function combines all the actions in its parameter list into a single large action. Using this function we can now give a concise definition of putStr:

```
putStr cs = sequence_ (map putChar cs)
```

Or even shorter:

```
putStr = sequence_ . map putChar
```

Since this combination of sequence_ with map occurs so often, the prelude defines a function:

```
mapM_ :: (a -> IO b) -> [a] -> IO ()
mapM_ f = sequence_ . map f
```

Exercises

12.1 Write the function getLine in terms of getChar using **do**-notation

12.2 Extend the guessing game such that you detect when the player does not stick to the rules.

12.3 Write a function sequence :: $[IO\ a] \to IO\ [a]$, which combines all the side effects of the individual actions in the list, and returns the list of all the individual results to the rest of the program. What do you think is the type of mapM, which is part of the prelude?

12.4 Write a function which prompts for a filename, reads the file with this name, splits the file into a number of lines, splits each line in a number of words separated by `' '`, and prints the total number of lines and words.

12.5 Given the function getInt :: *IO Int*, which reads an integer value from standard input, write a program that results in the following IO behaviour (the `3` has been typed in by the user):

```
Give a number: 3
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
...
10 * 3 = 30
Goodbye
```

12.6 Write a function of type [*String*] -> *String* -> *IO* () which concatenates a list of files to a specific target file: the first parameter is a list of filenames and the second parameter the name of the target file. Next write a program that first asks for the name of the target file, and then continues asking for names of files to be appended to that file until an empty line is entered. Note that the target files may be one of the source files! Do not use the function appendFile yet.

12.7 If we know that none of the source files equals the target file we may do a bit better using the function appendFile from the prelude or System.*IO*. Look up its definition using hoogle, and change the function you have written above using this function. What are the advantages and disadvantages of this approach?

12.8 With the function getArgs from the module System.Environment we can ask for the parameters which were entered form the command line when the program was called. Write a program CopyFiles, such that when called as ./CopyFiles a b c d the contents of the files a, b and c is copied into the file d.

13 Het bewijzen van eigenschappen van programma's

13.1 Wiskundige wetten

Wiskundige functies hebben de prettige eigenschap dat hun resultaat niet afhangt van de context van de berekening. De waarde van $2 + 3$ is altijd 5, of deze expressie nu deel uitmaakt van de expressie $4 \times (2 + 3)$ of bijvoorbeeld $(2 + 3)^2$.

Veel operatoren voldoen aan bepaalde rekenregels. Zo geldt bijvoorbeeld voor alle getallen x en y dat $x + y = y + x$. Zo'n rekenregel wordt een *wet* genoemd. Enkele rekenkundige wetten zijn:

commutatieve wet voor $+$	$x + y = y + x$
commutatieve wet voor $\times$	$x \times y = y \times x$
associatieve wet voor $+$	$x + (y + z) = (x + y) + z$
associatieve wet voor $\times$	$x \times (y \times z) = (x \times y) \times z$
distributieve wet	$x \times (y + z) = (x \times y) + (x \times z)$
wet voor herhaald machtsverheffen	$(x^y)^z = x^{(y \times z)}$

Dit soort rekenregels kun je goed gebruiken om expressies te tranformeren tot expressies die dezelfde waarde hebben. Daardoor kun je uitgaande van bestaande wetten nieuwe wetten afleiden. Het bekende merkwaardige product $(a + b)^2 = a^2 + 2ab + b^2$ volgt bijvoorbeeld uit bovenstaande wetten:

$(a+b)^2$
$=$ (definitie kwadraat)
$(a+b) \times (a+b)$
$=$ (distributieve wet)
$((a+b) \times a) + (a+b) \times b$
$=$ (commutatieve wet voor $\times$ (twee keer))
$(a \times (a+b)) + (b \times (a+b))$
$=$ (distributieve wet (twee keer))
$(a \times a + a \times b) + (b \times a + b \times b)$
$=$ (associatieve wet voor $+$)
$a \times a + (a \times b + b \times a) + b \times b$
$=$ (definitie kwadraat (twee keer))

$a^2 + (a \times b + b \times a) + b^2$
$=$ (commutatieve wet voor $\times$)
$a^2 + (a \times b + a \times b) + b^2$
$=$ (definitie "$(2\times)$")
$a^2 + 2 \times a \times b + b^2$

In elke tak van wiskunde worden nieuwe functies en operatoren gedefinieerd, waarvoor ook weer rekenregels gelden. In de propositielogica bijvoorbeeld gelden de volgende regels om te "rekenen" met boolse waarden:

commutatieve wet voor $\wedge$	$x \wedge y = y \wedge x$
associatieve wet voor $\wedge$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
distributieve wet	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
wet van de Morgan	$\neg(x \wedge y) = \neg x \vee \neg y$
wet van Howard	$(x \wedge y) \to z = x \to (y \to z)$

Het handige van wetten is dat je er een aantal achter elkaar kunt toepassen, zonder dat je je druk hoeft te maken om de betekenis van de tussenliggende stappen. Zo kun je bijvoorbeeld de wet $\neg((a \vee b) \vee c) \to \neg d = \neg a \to (\neg b \to (\neg c \to \neg d))$ afleiden met gebruik van bovenstaande wetten:

$\neg((a \vee b) \vee c) \to \neg d$
$=$ (wet van de Morgan)
$(\neg(a \vee b) \wedge \neg c) \to \neg d$
$=$ (wet van de Morgan)
$((\neg a \wedge \neg b) \wedge \neg c) \to \neg d$
$=$ (wet van Howard)
$(\neg a \wedge \neg b) \to (\neg c \to \neg d)$
$=$ (wet van Howard)
$\neg a \to (\neg b \to (\neg c \to \neg d))$

Zelfs als je niet zou weten wat $\wedge$, $\vee$ en $\neg$ betekenen kun je de geldigheid van de nieuwe wet inzien, mits je de geldigheid van de gebruikte wetten accepteert. Bij elke gelijkheid staat namelijk een aanwijzing ("hint") die aangeeftvolgens welke wet deze gelijkheid waar is. De hints zijn erg belangrijk. Als ze ontbreken, moet een lezer die twijfelt aan een stap zelf bedenken welke wet gebruikt is. De aanwezigheid van hints draagt in belangrijke mate bij aan de "leesbaarheid" van een afleiding.

13.2 Haskell-wetten

Haskell-functies hebben dezelfde eigenschap als wiskundige functies: een aanroep van een functie met dezelfde parameter levert altijd dezelfde waarde. Als je ergens in een expressie een deelexpressie vervangt door een andere deelexpressie die dezelfde waarde heeft, dan maakt dat voor het eindantwoord niet uit.

Voor allerlei functies is het mogelijk om wetten te formuleren waar ze aan voldoen. Zo

geldt bijvoorbeeld voor alle functies f en alle lijsten xs:

```
(map f . map g) xs = map (f . g) xs
```

Dit is geen *definitie* van map of . (die zijn al eerder gedefinieerd); het is een *wet* waar deze functies aan blijken te voldoen.

Wetten voor Haskell-functies kun je gebruiken bij het schrijven van een programma. Je kunt er programma's overzichtelijker of sneller mee maken. In de definitie van de determinantfunctie op matrices (zie paragraaf 7.2.3) komt bijvoorbeeld de volgende expressie p. 160 voor:

```
(altsum . zipWith (*.) ry . map det . map Mat . gaps . transpose) rys
```

Door bovenstaande wet hierop toe te passen blijkt dat dit ook geschreven kan worden als

```
(altsum . zipWith (*.) ry . map (det . Mat) . gaps . transpose) rys
```

Door gebruik te maken van wetten kunnen programma's worden geoptimaliseeerd door delen van het programma door equivalente maar goedkoper uit te rekenen alternatieven. Net als bij het rekenen met getallen of proposities is het weer niet nodig om alle tussenliggende programma's (of zelfs maar het uiteindelijke programma) te begrijpen. Als het eerste programma goed is, de wetten zijn geldig, en je maakt geen fouten bij het toepassen van de wetten, dan doet het uiteindelijke programma gegarandeerd hetzelfde als het eerste programma.

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

  ```
  f . (g . h) = (f . g) . h
  ```

- map f distribueert over ++, dus

  ```
  map f (xs ++ ys) = map f xs ++ map f ys
  ```

- de generalisatie hiervan naar een *lijst van* lijsten in plaats van *twee* lijsten:

  ```
  map f . concat = concat . map (map f)
  ```

- map distribueert over samenstelling:

  ```
  map (f . g) = map f . map g
  ```

- Als f associatief is (dus x `f` (y `f` z) = (x `f` y) `f` z), en e is het neutrale element van f (dus x `f` e = e `f` x = x voor alle x), dan geldt voor eindige lijsten xs:

  ```
  foldr f e xs = foldl f e xs
  ```

- Als bij foldr de beginwaarde het neutrale element van de operator is, dan is foldr over een singletonlijst de identiteit:

  ```
  foldr f e [x] = x
  ```

- Een element op kop zetten van een lijst kan verwisseld worden met map, mits je dan de functiewaarde van het element op kop zet:

  ```
  map f . (x:) = (f x:) . map f
  ```

- Elk gebruik van map kan ook geschreven worden als aanroep van foldr:

  ```
  map f xs = foldr g [] xs where g x ys = f x : ys
  ```

Veel van dit soort wetten komen overeen met wat men in imperatieve talen "programmeertrucs" zou noemen. De vierde wet in bovenstaand rijtje zou men in een imperatieve taal bijvoorbeeld beschrijven als "samenvoegen van twee loops". In imperatieve talen zijn de met de wetten overeenkomende programmatransformaties echter niet blindelings toe te passen: je moet er daar altijd op verdacht zijn dat functies onverwachte neveneffecten hebben. In functionele talen zoals Haskell mogen de wetten *altijd* toegepast worden; functies hebben immers altijd dezelfde waarde ongeacht de context.

13.3 Het bewijzen van wetten

Van een aantal wetten is het intuïtief duidelijk dat ze gelden. Van andere wetten is de geldigheid niet op het eerste gezicht duidelijk. Vooral in het laatste geval, maar ook in het eerste, is het nuttig om de wet te *bewijzen*. Daarbij kan gebruik gemaakt worden van de definities van functies, en van eerder bewezen wetten. In paragraaf 13.1 werden op die manier al de wetten $(a+b)^2 = a^2+2ab+b^2$ en $\neg((a \vee b) \vee c) \rightarrow \neg d = \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$ bewezen. p. 234

Ook het bewijs van wetten voor Haskell-functies ziet er zo uit. Het is handig wetten een naam te geven, zodat je er later (bij het bewijs van andere wetten) eenvoudig aan kunt refereren. De formulering van een wet, compleet met bewijs, ziet er dan bijvoorbeeld als volgt uit:

Wet *foldr over een singletonlijst*
Als e het neutrale element is van de operator f, dan geldt

```
foldr f e [x] = x
```

Bewijs:

```
foldr f e [x]
=   (notatie lijstopsomming)
foldr f e (x:[])
=   (def. foldr)
f x (foldr f e [])
=   (def. foldr)
f x e
=   (voorwaarde van de wet)
x
```

Als een wet de gelijkheid van twee *functies* beschrijft, dan kan deze bewezen worden door te bewijzen dat het resultaat van de functie gelijk is voor alle mogelijke parameters. Beide functies worden dus op een variabele `x` toegepast, waarna gelijkheid wordt aangetoond. Dit is bijvoorbeeld het geval in de volgende wet:

Wet *functiecompositie is associatief*
Voor alle functies `f`, `g` en `h` van het juiste type geldt:

`f . (g . h) = (f . g) . h`

Bewijs:

```
(f . (g.h)) x
=   (def. (.))
f ((g.h) x)
=   (def. (.))
f (g (h x))
=   (def. (.))
(f.g) (h x)
=   (def. (.))
((f.g) . h) x
```

Het nalezen van zo'n bewijs is niet altijd even spannend. Dat het schrijven van een bewijs lastig is, merk je pas als je het zelf moet bedenken. De eerste paar stappen zijn meestal niet zo moeilijk. Maar vaak kom je halverwege vast te zitten. Het is dan handig om ook een stukje van de andere kant te werken. Om aan te geven dat een bewijs op die manier opgebouwd is, zullen we het in het vervolg in twee kolommen weergeven:

	`f . ( g . h )`	`( f . g ) . h`
`x`	`(f . (g.h)) x`	`((f.g) . h) x`
	= (def. `(.)`)	= (def. `(.)`)
	`f ((g.h) x)`	`(f.g) (h x)`
	= (def. `(.)`)	= (def. `(.)`)
	`f (g (h x))`	`f (g (h x))`

In de twee kolommen van deze opzet worden de twee kanten van de wet bewezen gelijk te zijn aan dezelfde expressie. Twee dingen die aan dezelfde expressie gelijk zijn, zijn natuurlijk ook aan elkaar gelijk, en dat moest bewezen worden. In de eerste kolom van het schema is nog aangegeven op welke parameter (`x`) de linker- en de rechterfunctie worden toegepast.

Deze bewijsmethode kan ook gebruikt worden om een andere wet te bewijzen:

Wet *map na op-kop*
Voor alle waarden x en functies f van het juiste type geldt:

$$\mathsf{map\ f\ .\ (x:)} = \mathsf{((f\ x):)\ .\ map\ f}$$

Bewijs:

	`map f . (x:)`	`((f x):) . map f`

`xs`	`(map f . (x:)) xs`
	= (def. `(.)`)
	`map f ((x:) xs)`
	= (sectienotatie)
	`map f (x:xs)`
	= (def. `map`)
	`f x : map f xs`
		`( ((f x):) . map f) xs`
		= (def. `(.)`)
		`((f x):) (map f xs)`
		= (sectienotatie)
		`f x : map f xs`

Net als het vorige bewijs had dit bewijs natuurlijk ook als één lange afleiding geschreven kunnen worden. In deze tweekolomsnotatie is het echter duidelijker hoe het bewijs onstaan is: de twee kanten van de wet zijn gelijk bewezen aan een derde expressie, en daarmee ook aan elkaar.

13.4 Bewijzen met structurele inductie

Functies op lijsten hebben vaak een inductieve definitie. De functie wordt daarbij apart gedefinieerd voor [ ]. Dan wordt de functie gedefinieerd voor het patroon (x : xs), waarbij de functie recursief aangeroepen mag worden op xs.

Bij het bewijs van wetten waarin eindige lijsten een rol spelen, kan ook inductie worden gebruikt. De wet wordt daarbij apart bewezen voor het geval dat de lijst [ ] is. Vervolgens wordt de wet bewezen voor een lijst van de vorm (x:xs). Bij dat bewijs mag al aangenomen worden dat de wet geldig is voor de lijst xs. Een voorbeeld van een wet die met inductie bewezen kan worden is de distributie van map over ++.

Wet *map na ++*
Voor alle functies f en alle lijsten xs en ys van het juiste type geldt:

$$\mathsf{map\ f\ (xs\ ++\ ys)} = \mathsf{map\ f\ xs\ ++\ map\ f\ ys}$$

Bewijs met inductie naar xs:

IH xs	`map f (xs++ys)`	`map f xs ++ map f ys`

`[]`	`map f ([]++ys)`
	= (def. `++`)
	`map f ys`
		`map f [] ++ map f ys`
		= (def. `map`)
		`[] ++ map f ys`
		= (def. `++`)
		`map f ys`

`x:xs`	`map f ((x:xs)++ys)`
	= (def. `++`)
	`map f (x:(xs++ys))`
	= (def. `map`)
	`f x : map f (xs++ys)`
	= (IH xs)
	`f x : (map f xs ++ map f ys)`
		`map f (x:xs) ++ map f ys`
		= (def. `map`)
		`(f x : map f xs) ++ map f ys`
		= (def. `++`)
		`f x : (map f xs ++ map f ys)`

In dit bewijs wordt in het eerste gedeelte de wet geformuleerd voor de lijst xs. Dit heet de *inductiehypothese*, die we in de tabel afkorten met IH (ook in de toepassing van deze op de laatste regel). In het tweede gedeelte wordt de wet bewezen voor de lege lijst: hier is dus [] ingevuld waar in de originele wet xs stond. In het derde gedeelte wordt de wet bewezen met (x:xs) ingevuld voor xs. De laatste regel van de twee kolommen is weliswaar niet dezelfde expressie, maar omdat in dit gedeelte van het bewijs de inductiehypothese aangenomen mag worden, is de gelijkheid toch geldig.

In de wet "map na functiecompositie" worden twee functies gelijkgesteld. Om deze gelijkheid te bewijzen, worden linker- en rechterkant op een parameter xs toegepast. Daarna verloopt het bewijs met inductie naar xs:

Wet *map na functiecompositie*
Voor alle samenstelbare functies f en g geldt:

map (f . g) = map f . map g

Bewijs met inductie naar xs:

	`map (f.g)`	`map f . map g`

IH xs	`map (f.g) xs`	`(map f . map g) xs`
		= (def. `(.)`)
		`map f (map g xs)`

`[]`	`map (f.g) []`
	= (def. `map`)
	`[]`
		`map f (map g [])`
		= (def. `map`)
		`map f []`
		= (def. `map`)
		`[]`

`x:xs`	`map (f.g) (x:xs)`
	= (def. `map`)
	`(f.g) x : map (f.g) xs`
	= (def. `(.)`)
	`f(g x) : map (f.g) xs`
	= (IH)
	`f(g x) : map f (map g xs)`
		`map f (map g (x:xs))`
		= (def. `map`)
		`map f (g x : map g xs)`
		= (def. `map`)
		`f(g x) : map f (map g xs)`

In dit bewijs is de rechterkant van de stelling eerst nog vereenvoudigd, voordat de eigenlijke inductie begint; deze stap zou anders in beide gedeeltes van het inductieve bewijs

gedaan moeten worden. Dat gebeurt ook in het bewijs van de volgende wet:

Wet *map na concat*
Voor alle functies f geldt:

$$\mathsf{map\ f\ .\ concat} = \mathsf{concat\ .\ map\ (map\ f)}$$

Bewijs met inductie naar xss:
Dit is een generalisatie van de distributiewet van map over ++.

	`map f . concat`	`concat . map (map f)`

`IH xss`	`(map f . concat) xss`
	= (def. `(.)`)
	`map f (concat xss)`
		`(concat . map (map f)) xss`
		= (def. `(.)`)
		`concat (map (map f) xss)`

`[]`	`map f (concat [])`
	= (def. `concat`)
	`map f []`
	= (def. `map`)
	`[]`
		`concat (map (map f) [])`
		= (def. `map`)
		`concat []`
		= (def. `concat`)
		`[]`

`xs:xss`	`map f (concat (xs:xss))`
	= (def. `concat`)
	`map f (xs++concat xss)`
	= (distributiewet)
	`map f xs`
	`++ map f (concat xss)`
		`concat (map (map f) (xs:xss))`
		= (def. `map`)
		`concat (map f xs : map (map f) xss)`
		= (def. `concat`)
		`map f xs`
		`++ concat (map (map f) xss)`
		= (IH xss)
		`map f xs ++ map f (concat xss)`

In dit bewijs wordt behalve de definitie van functies en notaties ook een andere wet gebruikt, namelijk de eerder bewezen distributiewet van map over ++.

Niet altijd is het gevalsonderscheid []/(x:xs) voldoende om een wet te bewijzen. Datzelfde geldt trouwens voor de definitie van functies. In het bewijs van de volgende wet worden *drie* gevallen onderscheiden: de lege lijst, een singletonlijst, en een lijst met minstens twee elementen (x1 : x2 : xs).

Wet *dualiteitswet*
Als f een associatieve operator is (dus x `f` (y `f` z) = (x `f` y) `f` z), en e is het neutrale element van f (dus f x e = f e x = x voor alle x), dan geldt:

$$\mathsf{foldr\ f\ e} = \mathsf{foldl\ f\ e}$$

Bewijs met inductie naar xs:

	`foldr f e`	`foldl f e`
`IH xs`	`foldr f e xs`	`foldl f e xs`

`[]`	`foldr f e []`
	= (def. `foldr`)
	`e`
		`foldl f e []`
		= (def. `foldl`)
		`e`

`[x]`	`foldr f e [x]`
	= (def. `foldr`)
	`f x (foldr f e [])`
	= (def. `foldr`)
	`f x e`
	= (`e` neutraal element)
	`x`
		`foldl f e [x]`
		= (def. `foldl`)
		`foldl f (e‘f‘x) []`
		= (def. `foldl`)
		`e‘f‘x`
		= (`e` neutraal element)
		`x`

`x1:x2:xs`	`foldr f e (x1:x2:xs)`
	= (def. `foldr`)
	`x1 ‘f‘ foldr f e (x2:xs)`
	= (def. `foldr`)
	`x1 ‘f‘ (x2 ‘f‘ foldr f e xs)`
	= (`f` associatief)
	`(x1‘f‘x2) ‘f‘ foldr f e xs`
	= (def. `foldr`)
	`foldr f e ((x1‘f‘x2):xs)`
	= (IH voor `(x1‘f‘x2):xs)`)
	`foldl f e ((x1‘f‘x2):xs)`
		`foldl f e (x1:x2:xs)`
		= (def. `foldl`)
		`foldl f (e‘f‘x1) (x2:xs)`
		= (def. `foldl`)
		`foldl f ((e‘f‘x1)‘f‘x2) xs`
		= (`f` associatief)
		`foldl f (e‘f‘(x1‘f‘x2)) xs`
		= (def. `foldl`)
		`foldl f e ((x1‘f‘x2):xs)`

We mogen hier inderdaad de inductiehypothese toepassen omdat de lijst $(\mathsf{x1}\ \text{`}\mathsf{f}\text{`}\ \mathsf{x2}) : \mathsf{xs}$ korter is dan $\mathsf{x1} : \mathsf{x2} : \mathsf{xs}$. De wet mag voor deze lijst dus al aangenomen worden.

13.5 Verbetering van efficiëntie

Wetten kunnen gebruikt worden om functies te transformeren in efficiëntere functies. Twee expressies waarvan de gelijkheid bewezen is, hoeven immers niet even snel berekend te worden; in dat geval kan de langzamere definitie vervangen worden door de snellere. In deze paragraaf worden twee voorbeelden van deze techniek bekeken:

- de reverse-functie wordt verbeterd van $\mathcal{O}(n^2)$ tot $\mathcal{O}(n)$;
- de Fibonacci-functie, die al eerder was verbeterd van $\mathcal{O}(2^n)$ tot $\mathcal{O}(n)$ wordt verder verbeterd tot $\mathcal{O}(\log n)$.

reverse

Voor het verbeteren van de reverse-functie bewijzen we drie wetten:

- Naast het in de vorige paragraaf bewezen verband tussen foldr en foldl is er nog een verband: de *tweede dualiteitswet*:

  $$\mathsf{foldr\ f\ e\ (reverse\ xs)} = \mathsf{foldl\ (flip\ f)\ e\ xs}$$

- Het bewijs van bovenstaande wet lukt niet in één keer. Er is een andere wet bij nodig, die apart met inductie bewezen kan worden:

foldr f e (xs ++ [y]) = foldr f (f y e) xs

- Zo ongeveer de eenvoudigste wet die met inductie bewezen kan worden is:

  foldr (:) [] = id

We beginnen met de derde wet. Daarna volgt een bewijs van de hulpwet, en vervolgens de tweede dualiteitswet zelf. Met deze dualiteitswet verbeteren we tenslotte de reverse-functie.

Wet *foldr met constructorfuncties*
Als aan foldr de constructorfuncties van lijsten worden meegegeven, te weten (:) en [], is het resultaat de identiteit:

foldr (:) [] = id

Bewijs met inductie naar xs:

	`foldr (:) []`	`id`

`IH xs`	`foldr (:) [] xs`	`id xs`
		= (def. `id`)
		`xs`

`[]`	`foldr (:) [] []`
	= (def. `foldr`)
	`[]`
		`[]`

`x:xs`	`foldr (:) [] (x:xs)`
	= (def. `foldr`)
	`x : foldr (:) [] xs`
	= (IH xs)
	`x : xs`
		`x : xs`

Wet *hulpwet voor de volgende wet*

foldr f e (as ++ [b]) = foldr f (f b e) as

Bewijs met inductie naar **as**:

`IH as`	`foldr f e (as++[b])`	`foldr f (f b e) as`

`[]`	`foldr f e ([]++[b])`
	= (def. `++`)
	`foldr f e [b]`
	= (def. `foldr`)
	`f b (foldr f e [])`
	= (def. `foldr`)
	`f b e`
		`foldr f (f b e) []`
		= (def. `foldr`)
		`f b e`

`a:as`	`foldr f e ((a:as)++[b])`
	= (def. `++`)
	`foldr f e (a:(as++[b]))`
	= (def. `foldr`)
	`f a (foldr f e (as++[b]))`
		`foldr f (f b e) (a:as)`
		= (def. `foldr`)
		`f a (foldr f (f b e) as)`
		= (IH as)
		`f a (foldr f e (as++[b]))`

Wet *tweede dualiteitswet*
Voor alle functies f, waardes e en lijsten xs van het juiste type geldt:

foldr f e (reverse xs) = foldl (flip f) e xs

Bewijs met inductie naar xs:

IH xs	`foldr f e (reverse xs)`	`foldl (flip f) e xs`

`[]`	`foldr f e (reverse [])`
	= (def. `reverse`)
	`foldr f e []`
	= (def. `foldr`)
	`e`
		`foldl (flip f) e []`
		= (def. `foldl`)
		`e`

`x:xs`	`foldr f e (reverse (x:xs))`
	= (def. `reverse`)
	`foldr f e (reverse xs++[x])`
	= (hulpwet hierboven)
	`foldr f (f x e) (reverse xs)`
		`foldl (flip f) e (x:xs)`
		= (def. `foldl`)
		`foldl (flip f) (flip f e x) xs`
		= (def. `flip`)
		`foldl (flip f) (f x e) xs`
		= (IH met `f x e` voor `e`)
		`foldr f (f x e) (reverse xs)`

De inductiehypothese mag worden aangenomen voor *alle* e, dus ook met f x e ingevuld voor e. (De inductiehypothese mag daarentegen voor de variabele waarnaar de inductie verloopt, xs, alleen voor vaste xs aangenomen worden; anders valt de hele inductie in duigen.)

De functie reverse is in paragraaf 4.1.2 gedefinieerd als p. 85

```
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

of het daaraan equivalente

```
reverse = foldr post []
  where
    post x xs = xs ++ [x]
```

Op deze manier gedefinieerd kost de functie $\mathcal{O}(n^2)$ tijd, waarbij n de lengte van de om te keren lijst is. De operator ++ in post kost immers $\mathcal{O}(n)$ tijd, en dit moet vermenigvuldigd worden met de $\mathcal{O}(n)$ van de recursie (al of niet verborgen in foldr).

Maar uit de zojuist bewezen wetten kunnen we afleiden:

```
reverse xs
=   (def. id)
id (reverse xs)
=   (foldr met constructorfuncties)
foldr (:) [] (reverse xs)
=   (tweede dualiteitswet)
foldl (flip (:)) [] xs
```

De nieuwe definitie

```
reverse = foldl (flip (:)) []
```

kost slechts $\mathcal{O}(n)$ tijd. De operator die voor foldl gebruikt wordt, flip (:), kost immers slechts constante tijd.

Fibonacci

Ook wetten waarin natuurlijke getallen een rol spelen, kunnen soms met inductie worden bewezen. Daarbij wordt de wet apart bewezen voor het geval 0, en daarna voor het patroon $n+1$, waarbij de wet voor het geval n al gebruikt mag worden. In sommige bewijzen wordt een ander inductieschema aangehouden, bijvoorbeeld $0/1/n{+}2$; of $1/n{+}2$ als de wet niet hoeft te gelden voor het geval 0.

Inductie over natuurlijke getallen kunnen we goed gebruiken om een verbetering in de efficiëntie van de Fibonacci-functie te bereiken. De oorspronkelijke definitie daarvan was:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n − 2) + fib (n − 1)
```

De benodigde tijd voor het berekenen van fib n is $\mathcal{O}(2^n)$. Door memoïsatie is al een verbetering tot $\mathcal{O}(n)$ te bereiken, maar dankzij de volgende wet is een nog grotere verbetering mogelijk.

Wet *Fibonacci door machtsverheffen*
Stel $p = \frac{1}{2} + \frac{1}{2}\sqrt{5}$, $q = \frac{1}{2} - \frac{1}{2}\sqrt{5}$, en $c = 1/\sqrt{5}$. Dan geldt:

```
fib n = c ∗ (p ^ n − q ^ n)
```

De waarden p en q zijn de oplossingen van de vierkantsvergelijking $x^2 - x - 1 = 0$. Daarom geldt $p^2 = p + 1$ en $q^2 = q + 1$. Bovendien geldt $p - q = \sqrt{5} = 1/c$. Gebruik makend van deze eigenschappen bewijzen we de stelling. Het bewijs de stelling met inductie naar n vind je in Fig. 13.1.

Het opmerkelijke aan deze wet is dat ondanks al die wortels het eindantwoord toch weer geheeltallig is. Deze wet kan gebruikt worden om een $\mathcal{O}(\log n)$ versie van fib te maken. Machtsverheffen kan immers in $\mathcal{O}(\log n)$ tijd, met de halveringsmethode. Door fib te definiëren door

```
fib n = c ∗ (p ^. n − q ^. n)
  where
    c       = 1.0 / wortel5
    p       = 0.5 ∗ (1.0 + wortel5)
    q       = 0.5 ∗ (1.0 − wortel5)
    wortel5 = sqrt 5.0
```

kan ook fib in logaritmische tijd berekend worden. Een laatste optimalisatie is mogelijk door op te merken dat $q < 1$, en dat dus q^n, zeker voor grote n, verwaarloosd kan worden. Het is voldoende om $c \times p^n$ af te ronden op de dichtstbijzijnde integer.

13.6 Eigenschappen van functies

Behalve voor het verbeteren van de efficiëntie van bepaalde functies zijn wetten ook gewoon handig om meer inzicht te krijgen in de werking van bepaalde functies. Bij

IH n	`fib` n	$c \times (p^n - q^n)$
0	`fib 0` = (def. `fib`) 0 = (def. $\times$) $c \times 0$	$c \times (p^0 - q^0)$ = (def. machtsverheffen) $c \times (1 - 1)$ = (eigenschap $-$) $c \times 0$
1	`fib 1` = (def. `fib`) 1 = (def. /) $c \times (1/c)$	$c \times (p^1 - q^1)$ = (def. machtsverheffen) $c \times (p - q)$ = (eigenschap c) $c \times (1/c)$
n+2	`fib` $(n+2)$ = (def. `fib`) `fib` n `+ fib` $(n+1)$ = (IH n) $c \times (p^n - q^n)$+ `fib` $(n+1)$ = (IH voor $n+1$) $c \times (p^n - q^n)$+ $c \times (p^{n+1} - q^{n+1})$	$c \times (p^{n+2} - q^{n+2})$ = (eigenschap machtsverheffen) $c \times (p^n p^2 - q^n q^2)$ = (eigenschap p en q) $c \times (p^n(1+p) - q^n(1+q))$ = (distributie $\times$) $c \times ((p^n + p^{n+1}) - (q^n + q^{n+1}))$ = (commutativiteit en associativiteit +) $c \times ((p^n - q^n) + (p^{n+1} - q^{n+1}))$ = (distributie $\times$) $c \times (p^n - q^n) + c \times (p^{n+1} - q^{n+1})$

Fig. 13.1: Bewijs Fibonacci voor machtsverheffen

functies die een lijst opleveren is het bijvoorbeeld interessant om te weten hoe de lengte afhangt van de parameter van die functie. Hieronder volgen vier wetten over de lengte van het resultaat van een functie. Daarna volgen drie wetten over de som van de resultaatlijst van een functie. De wetten worden daarna gebruikt om iets te kunnen zeggen over de lengte van het resultaat van combinatorische functies.

Wetten over lengte

In deze paragraaf wordt de length-functie geschreven als len (om schrijfwerk te besparen).

Wet *lengte na op-kop*

Door een element op kop te zetten van een lijst wordt de lengte één groter:

$$\mathsf{len} \;.\; (\mathsf{x}:) = (1+) \;.\; \mathsf{len}$$

Deze wet volgt vrijwel direct uit de definitie. Er is geen inductie nodig:

	`len . (x:)`	`(1+) . len`

`xs`	`(len . (x:)) xs`
	= (def. `(.)`)
	`len (x:xs)`
	= (def. `len`)
	`1 + len xs`
		`((1+) . len) xs`
		= (def. `(.)`)
		`1 + len xs`

Wet *lengte na map*
Door het map-pen van een functie op een lijst blijft de lengte van de lijst onveranderd:

$$\mathsf{len} \;.\; \mathsf{map}\ \mathsf{f} = \mathsf{len}$$

Het bewijs verloopt met inductie naar xs:

	`len . map f`	`len`

`IH xs`	`(len . map f) xs`
	= (def. `(.)`)
	`len (map f xs)`
		`len xs`

`[]`	`len (map f [])`
	= (def. `map`)
	`len []`
		`len []`

`x:xs`	`len (map f (x:xs))`
	= (def. `map`)
	`len (f x : map f xs)`
	= (def. `len`)
	`1 + len (map f xs)`
		`len (x:xs)`
		= (def. `len`)
		`1 + len xs`
		= (IH xs)
		`1 + len (map f xs)`

Wet *lengte na ++*
De lengte van de concatenatie van twee lijsten is de som van de lengtes van die lijsten:

$$\mathsf{len}\ (\mathsf{xs} +\!+ \mathsf{ys}) = \mathsf{len}\ \mathsf{xs} + \mathsf{len}\ \mathsf{ys}$$

Het bewijs verloopt met inductie naar xs:

`IH xs`	`len (xs++ys)`	`len xs + len ys`

`[]`	`len ([]++ys)`
	= (def. `++`)
	`len ys`
		`len [] + len ys`
		= (def. `len`)
		`0 + len ys`
		= (def. `+`)
		`len ys`

`x:xs`	`len ((x:xs)++ys)`
	= (def. `++`)
	`len (x:(xs++ys))`
	= (def. `len`)
	`1 + len (xs++ys)`
		`len (x:xs) + len ys`
		= (def. `len`)
		`(1+len xs) + len ys`
		= (associativiteit +)
		`1 + (len xs + len ys)`
		= (IH xs)
		`1 + len (xs++ys)`

De volgende wet is een generalisatie hiervan: in deze wet komt een lijst van lijsten voor, in plaats van twee lijsten, en de operator + is dan ook vervangen door sum.

Wet *lengte na concatenatie*
De lengte van een concatenatie van een lijst van lijsten is de som van de lengtes van al die lijsten:

$$\mathsf{len} \;.\; \mathsf{concat} = \mathsf{sum} \;.\; \mathsf{map}\ \mathsf{len}$$

Het bewijs verloopt met inductie naar xss:

	`len . concat`	`sum . map len`
`IH xss`	`len (concat xss)`	`sum (map len xss)`

`[]`	`len (concat [])`
	= (def. `concat`)
	`len []`
	= (def. `len`)
	`0`
		`sum (map len [])`
		= (def. `map`)
		`sum []`
		= (def. `sum`)
		`0`

`xs:xss`	`len (concat (xs:xss))`
	= (def. `concat`)
	`len (xs++concat xss)`
	= (lengte na ++)
	`len xs + len (concat xss)`
		`sum (map len (xs:xss))`
		= (def. `map`)
		`sum (len xs : map len xss)`
		= (def. `sum`)
		`len xs + sum (map len xss)`
		= (IH xss)
		`len xs + len (concat xss)`

Wetten over sum

Net als voor len zijn er voor sum twee wetten om hem over concatenatie te distribueren (van twee lijsten of van een lijst van lijsten).

Wet *sum na ++*

De som van de concatenatie van twee lijsten is gelijk aan de sommen van die twee lijsten opgeteld:

$$\mathsf{sum}\ (\mathsf{xs} +\!+ \mathsf{ys}) = \mathsf{sum}\ \mathsf{xs} + \mathsf{sum}\ \mathsf{ys}$$

Bewijs met inductie naar xs:

`IH xs`	`sum (xs++ys)`	`sum xs + sum ys`

`[]`	`sum ([]++ys)`
	= (def. `++`)
	`sum ys`
		`sum [] + sum ys`
		= (def. `sum`)
		`0 + sum ys`
		= (def. `+`)
		`sum ys`

`x:xs`	`sum ((x:xs)++ys)`
	= (def. `++`)
	`sum (x:(xs++ys))`
	= (def. `sum`)
	`x + sum(xs++ys)`
		`sum (x:xs) + sum ys`
		= (def. `sum`)
		`(x+sum xs) + sum ys`
		= (associativiteit +)
		`x + (sum xs + sum ys)`
		= (IH xs)
		`x + sum(xs++ys)`

Net als bij len wordt deze wet gebruikt in het bewijs van de generalisatie naar lijsten van lijsten.

Wet *sum na concatenatie*

De som van de concatenatie van een lijst van lijsten is de som van de sommen van die lijsten:

$$\mathsf{sum}\ .\ \mathsf{concat} = \mathsf{sum}\ .\ \mathsf{map}\ \mathsf{sum}$$

Het bewijs verloopt met inductie naar xss:

	`sum . concat`	`sum . map sum`
`IH xss`	`sum (concat xss)`	`sum (map sum xss)`

`[]`	`sum (concat [])`
	= (def. `concat`)
	`sum []`
		`sum (map sum [])`
		= (def. `map`)
		`sum []`

`xs:xss`	`sum (concat (xs:xss))`
	= (def. `concat`)
	`sum (xs ++ concat xss)`
	= (sum na ++)
	`sum xs + sum (concat xss)`
		`sum (map sum (xs:xss))`
		= (def. `map`)
		`sum (sum xs : map sum xss)`
		= (def. `sum`)
		`sum xs + sum (map sum xss)`
		= (IH xss)
		`sum xs + sum (concat xss)`

Er is geen wet voor de sum van een map op een lijst, zoals die voor len gold. De som van de kwadraten van een getal is immers niet gelijk aan het kwadraat van de som of iets dergelijks. Wel is er een wet te formuleren voor het geval de gemapte functie de functie (1+) is.

Wet *sum na map-plus-1*
De som van een lijst opgehoogde getallen is de som van de oorspronkelijke lijst plus de lengte ervan:

$$\mathsf{sum}\ (\mathsf{map}\ (1+)\ \mathsf{xs}) = \mathsf{len}\ \mathsf{xs} + \mathsf{sum}\ \mathsf{xs}$$

Bewijs met inductie naar xs:

`IH xs`	`sum (map (1+) xs)`	`len xs + sum xs`

`[]`	`sum (map (1+) [])`
	= (def. `map`)
	`sum []`
		`len [] + sum []`
		= (def. `len`)
		`0 + sum []`
		= (def. `+`)
		`sum []`

`x:xs`	`sum (map (1+) (x:xs))`
	= (def. `map`)
	`sum (1+x : map (1+) xs)`
	= (def. `sum`)
	`(1+x) + sum (map (1+) xs)`
		`len (x:xs) + sum (x:xs)`
		= (def. len en sum)
		`(1+len xs) + (x+sum xs)`
		= (+ associatief en commutatief)
		`(1+x) + (len xs + sum xs)`
		= (IH xs)
		`(1+x) + sum (map (1+) xs)`

Wetten over combinatorische functies

Met behulp van een aantal hierboven genoemde wetten zijn wetten te bewijzen over combinatorische functies uit sectie 7.1. We bewijzen voor inits, segs en combs een wet die aangeeft hoeveel elementen het resultaat heeft (zie ook opgave 7.3): p. 144 p. 170

$$\begin{aligned}
&\mathsf{len}\ .\ \mathsf{inits} &&= (1+)\ .\ \mathsf{len}\\
&\mathsf{len}\ .\ \mathsf{segs} &&= \mathsf{f}\quad .\ \mathsf{len}\ \textbf{where}\ \mathsf{f}\ \mathsf{n} = 1 + (\mathsf{n} * \mathsf{n} + \mathsf{n})\ /\ 2\\
&\mathsf{len}\ .\ \mathsf{combs}\ \mathsf{k} &&= (\text{`boven`}\ \mathsf{k})\ .\ \mathsf{len}
\end{aligned}$$

Wet *aantal beginsegmenten*
Het aantal beginsegmenten van een lijst is één meer dan het aantal elementen van de lijst:

$$\mathsf{len} \mathbin{.} \mathsf{inits} = (1+) \mathbin{.} \mathsf{len}$$

Het bewijs verloopt met inductie naar xs:

	`len . inits`	`(1+) . len`
`IH xs`	`len (inits xs)`	`1 + len xs`

`[]`	`len (inits [])`
	= (def. `inits`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
		`1 + len []`

`x:xs`	`len (inits (x:xs))`
	= (def. `inits`)
	`len ([] : map (x:) (inits xs))`
	= (def. `len`)
	`1 + len (map (x:) (inits xs))`
	= (lengte na map)
	`1 + len (inits xs)`
		`1 + len (x:xs)`
		= (def. `len`)
		`1 + (1+len xs)`
		= (IH xs)
		`1 + len (inits xs)`

Wet *aantal segmenten*
Het aantal segmenten van een lijst is een kwadratische functie van het aantal elementen van de lijst:

$$\mathsf{len} \mathbin{.} \mathsf{segs} = \mathsf{f} \mathbin{.} \mathsf{len} \ \textbf{where} \ \mathsf{f}\ \mathsf{n} = 1 + (\mathsf{n} * \mathsf{n} + \mathsf{n}) \mathbin{/} 2$$

Het bewijs verloopt met inductie naar xs. We schrijven n voor len xs.

	`len . segs`	`f . len where f` n `=` $1+(n^2+n)/2$
`IH xs`	`len (segs xs)`	`f (len xs)`

`[]`	`len (segs [])`
	= (def. segs)
	`len [[]]`
	= (def. len)
	`1 + len []`
	= (def. len)
	`1 + 0`
		`f (len [])`
		= (def. len)
		`f 0`
		= (def. f)
		$1+(0^2+0)/2$
		= (uitrekenen)
		`1 + 0`

`x:xs`	`len (segs (x:xs))`
	= (def. segs)
	`len ( segs xs`
	`++ map (x:)(inits xs))`
	= (lengte na ++)
	`len (segs xs) +`
	`len (map (x:)(inits xs))`
	= (lengte na map)
	`len (segs xs) + len (inits xs)`
	= (aantal beginsegmenten)
	`len (segs xs) + 1 + len xs`
	= (IH xs)
	`f (len xs) + 1 + len xs`
	= (`len xs` is gelijk aan n)
	$f\ \ n+1+n$
	= (uitvouwen f)
	$1+(n^2+n)/2+1+n$
		`f (len (x:xs))`
		= (def. len)
		`f (1 +` n`)`
		= (def. f)
		$1+((1+n)^2+(1+n))/2$
		= (merkwaardig product)
		$1+((1+2n+n^2)+(1+n))/2$
		= (+ associatief en commutatief)
		$1+((n^2+n)+(2+2n))/2$
		= (distributie /)
		$1+(n^2+n)/2+1+n$

De definitie van boven in paragraaf 2.2.2 was niet volledig. De complete definitie luidt: p. 30

```
boven n k | n >= k = fac n / (fac k * fac (n - k))
          | n < k  = 0
```

Deze functie speelt een rol in de volgende wet.

Wet *aantal combinaties*
Het aantal combinaties van k elementen uit een lijst is de binomiaalcoëfficiënt "lengte van de lijst boven k":

```
len . combs k = (`boven` k) . len
```

In het bewijs gebruiken we de klassieke wiskundige notatie $n!$ voor fac n, en $\binom{n}{k}$ voor n `boven` k. We schrijven n voor len xs. Het bewijs verloopt met inductie naar k (met de gevallen 0 en k + 1). Het bewijs van de inductiestap (geval k + 1) verloopt opnieuw met inductie, ditmaal naar xs (met de gevallen [] en x : xs). Deze inductiestructuur komt

overeen met die van de definitie van combs.

	`len . combs k`	`(‘boven‘ k) . len`
`IH k,xs`	`len (combs k xs)`	$\binom{\mathsf{len\ xs}}{k}$

`0`
`xs`
	`len (combs 0 xs)`
	= (def. `combs`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
	= (def. `len`)
	`1 + 0`
	= (eigenschap +)
	`1`
		$\binom{\mathsf{len\ xs}}{0}$
		= (def. `boven`)
		$\frac{n!}{(n-0)!*0!}$
		= (def. fac en −)
		$\frac{n!}{n!*1}$
		= (def. / en ∗)
		1

`k+1`
`[]`
	`len (combs (k+1) [])`
	= (def. `combs`)
	`len []`
	= (def. `len`)
	`0`
		$\binom{\mathsf{len}\ []}{k+1}$
		= (def. `len`)
		$\binom{0}{k+1}$
		= (def. `boven`)
		0

`k+1`
`x:xs`
	`len (combs (k+1) (x:xs))`
	= (def. `combs`)
	`len (map (x:)(combs k xs)`
	`  ++ combs (k+1) xs)`
	= (lengte na ++)
	`len (map (x:)(combs k xs))`
	`+ len (combs (k+1) xs)`
	= (lengte na map)
	`len (combs k xs)`
	`+ len (combs (k+1) xs)`
	= (IH k,xs)
	$\binom{n}{k}$ `+ len (combs (k+1) xs)`
	= (IH k+1,xs)
	$\binom{n}{k}+\binom{n}{k+1}$
		$\binom{\mathsf{len\ (x:xs)}}{k+1}$
		= (def. `len`)
		$\binom{n+1}{k+1}$
		= (def. boven $(n \geqslant k)$)
		$\frac{(n+1)!}{((n+1)-(k+1))!*(k+1)!}$
		= (teller: def. fac; noemer: rekenen)
		$\frac{(n+1)*n!}{(n-k)!*(k+1)!}$
		= (teller: rekenen; noemer: def. fac $(n>k)$)
		$\frac{(k+1)*n!}{(n-k)!*(k+1)*k!}+\frac{(n-k)*n!}{(n-k)*(n-k-1)!*(k+1)!}$
		= (delen $(n>k)$)
		$\frac{n!}{(n-k)!*k!}+\frac{n!}{(n-(k+1))!*(k+1)!}$
		= (def. `boven`)
		$\binom{n}{k}+\binom{n}{k+1}$

De rechterkolom van het inductiestapbewijs is alleen geldig voor $n > k$. Voor $n = k$ verloopt het bewijs als volgt:

$$\binom{n+1}{k+1} \;=\; 1 \;=\; 1+0 \;=\; \binom{n}{k}+\binom{n}{k+1}$$

Voor $n < k$ luidt het bewijs:

$$\binom{n+1}{k+1} \;=\; 0 \;=\; 0+0 \;=\; \binom{n}{k} + \binom{n}{k+1}$$

13.7 Parametrische polymorfie

De volgende wetten zijn geldig voor alle functies f:

```
inits   . map f = map (map f) . inits
segs    . map f = map (map f) . segs
subs    . map f = map (map f) . subs
perms . map f = map (map f) . perms
```

Intuïtief is het wel duidelijk dat deze wetten gelden. Stel bijvoorbeeld dat je met inits de beginsegmenten van een lijst berekent, nadat je van alle elementen de f-waarde hebt berekend (door map f). Je had dan ook eerst de inits kunnen bepalen, en daarna in *elk* resulterend beginsegment op alle elementen f toepassen. In dat laatste geval (voorgesteld door de rechterkant van de wet) moet je f toepassen op de elementen van een lijst van lijsten, vandaar de dubbele map.

We bewijzen de eerste van de genoemde wetten; de andere zijn niet veel moeilijker.

Wet *beginsegmenten na map*
Voor all functies f op lijsten geldt:

```
inits . map f = map (map f) . inits
```

Het bewijs verloopt met inductie naar xs:

	`inits . map f`	`map (map f) . inits`
`IH xs`	`inits (map f xs)`	`map (map f) (inits xs)`

`[]`	`inits (map f [])`
	= (def. `map`)
	`inits []`
	= (def. `inits`)
	`[[]]`
		`map (map f) (inits [])`
		= (def. `inits`)
		`map (map f) [[]]`
		= (def. `map`)
		`[ map f [] ]`
		= (def. `map`)
		`[[]]`

`x:xs`	`inits (map f (x:xs))`
	= (def. `map`)
	`inits (f x:map f xs)`
	= (def. `inits`)
	`[] : map (f x:)`
	`(inits (map f xs))`
	= (IH xs)
	`[] : map (f x:)`
	`(map (map f)(inits xs))`
		`map (map f)(inits (x:xs))`
		= (def. `inits`)
		`map (map f)([]:map (x:)(inits xs))`
		= (def. `map`)
		`map f [] :`
		`map (map f)(map (x:)(inits xs))`
		= (def. `map`)
		`[] : map (map f)(map (x:)(inits xs))`
		= (map na functiecompositie)
		`[] : map (map f.(x:)) (inits xs)`
		= (map na op-kop)
		`[] : map ((f x:).map f) (inits xs)`
		= (map na functiecompositie)
		`[] : map (f x:)`
		`(map (map f)(inits xs))`

Een soortgelijke wet als de zojuist bewezene geldt voor *elke* combinatorische functie. Dat wil zeggen: als combinat een combinatorische functie is, dan geldt voor alle functies f dat

$$\mathsf{combinat}\ .\ \mathsf{map\ f} = \mathsf{map}\ (\mathsf{map\ f})\ .\ \mathsf{combinat}$$

Dat komt door de definitie van wat voor soort functies "combinatorische functie" genoemd worden: functies van lijsten naar lijsten van lijsten, die geen gebruik mogen maken van specifieke eigenschappen van elementen. Anders gezegd: combinatorische functies zijn polymorfe functies met als type

$$\mathsf{combinat} :: [a] \mathrel{->} [[a]]$$

Het is zelfs zo, dat bovengenoemde wet als *definitie* van combinatorische functies gebruikt kan worden. Dus: een functie combinat heet "combinatorisch" als voor alle functies f geldt:

$$\mathsf{combinat}\ .\ \mathsf{map\ f} = \mathsf{map}\ (\mathsf{map\ f})\ .\ \mathsf{combinat}$$

Met zo'n definitie, die een duidelijke omschrijving geeft met behulp van een wet, kun je meestal wat beter uit de voeten dan de enigszins vage omschrijving "mag geen gebruik maken van specifieke eigenschappen van elementen", die in sectie 7.1 werd gebruikt. p. 144

Er zijn wetten die lijken op deze "wet van de combinatorische functies". In paragraaf 13.4 p. 239

werd bijvoorbeeld de wet "map na concat" bewezen. Die wet stelt dat voor alle functies f geldt:

$$\mathsf{map\ f\ .\ concat} = \mathsf{concat\ .\ map\ (map\ f)}$$

De wet kun je natuurlijk ook andersom lezen. Dan staat er:

$$\mathsf{concat\ .\ map\ (map\ f)} = \mathsf{map\ f\ .\ concat}$$

In deze vorm lijkt de wet op de wet van de combinatorische functies. Het enige verschil is, dat de "dubbele map" nu aan de andere kant staat. Dat is ook niet zo gek, want het type van concat is:

$$\mathsf{concat} :: [[a]] \mathsf{\ ->\ } [a]$$

Neemt bij gebruik van combinatorische functies het aantal lijstnivo's toe, bij concat vermindert dat aantal juist. De dubbele map moet dan ook gebruikt worden *voordat* concat wordt toegepast; de enkele map er*na*.

De functie concat is geen combinatorische functie, om de eenvoudige reden dat hij niet aan de daarvoor geldende wet voldoet. Wel is de functie een *polymorfe* functie. In paragraaf 2.5.3 werd een polymorfe functie gedefinieerd als "een functie met een type waar typevariabelen in voorkomen". Net als het begrip "combinatorische functie" is het begrip "polymorfe functie" met een wet minder vaag te definiëren. Dat gaat als volgt: p. 50

> Een functie poly tussen lijsten heet een *polymorfe* functie als voor alle functies f geldt:
>
> $$\mathsf{poly\ .} \underbrace{\mathsf{map\ (\ldots map\ (\ldots (map\ f)))}}_{n\ \mathsf{map's}} = \underbrace{\mathsf{map\ (\ldots map\ (\ldots (map\ f))}}_{k\ \mathsf{map's}} \mathsf{\ .\ poly}$$
>
> Deze functie heeft dan het type:
>
> $$\mathsf{poly} :: \underbrace{[\ldots[\ldots[a]\ldots]}_{n\text{ dimensionale lijst}} \mathsf{\ ->\ } \underbrace{[\ldots[\ldots[a]\ldots]}_{k\text{ dimensionale lijst}}$$

Alle combinatorische functies zijn polymorf. De wet die voor combinatorische functies moet gelden is immers een speciaal geval van de wet voor polymorfe functies, met $n = 1$ en $k = 2$. Ook concat is polymorf: de wet "map na concat" heeft de geëiste vorm, met $n = 2$ en $k = 1$.

Ook voor andere datastructuren dan lijsten (bijvoorbeeld tupels, of bomen) kan het begrip "polymorfe functie" met behulp van een wet gedefinieerd worden. Er is dan een equivalent van map op de betreffende datastructuur nodig, die in de wet gebruikt kan worden in plaats van map.[1]

[1] De tak van wiskunde waarin deze constructie wordt uitgevoerd heet "categorietheorie". In de categorietheorie wordt een functie die aan deze wet voldoet een "natuurlijke transformatie" genoemd.

13.8 Bewijzen van rekenkundige wetten (in Haskell)

In paragraaf 13.1 is een aantal wiskundige wetten genoemd, zoals "vermenigvuldigen is associatief". Deze wetten kunnen ook bewezen worden. Bij een bewijs van een wet waarin een bepaalde functie een rol speelt, is echter de definitie van die functie nodig. Tot nu toe hebben we nog geen definitie gegeven van optellen en vermenigvuldigen; deze functies werden als "ingebouwd" beschouwd. p. 234

In theorie is het niet nodig dat de getallen, althans de natuurlijke getallen, in Haskell zijn ingebouwd. Het is namelijk mogelijk om ze te definiëren door middel van een datadeclaratie. In deze paragraaf zullen we de definitie van het type Nat (de natuurlijke getallen) geven, om twee redenen:

- om aan te tonen hoe krachtig het datadeclaratie mechanisme is (je kunt er zelfs de natuurlijke getallen mee definiëren!);
- om met inductie de rekenkundige operatoren te definiëren, waarna de rekenkundige wetten met inductie bewezen kunnen worden.

In de praktijk kun je de zo gedefinieerde natuurlijke getallen beter niet gebruiken, omdat de rekenkundige operaties niet erg efficiënt verlopen (vergeleken met de ingebouwde operatoren). Maar voor het gebruik bij het bewijzen van wetten voldoet de definitie uitstekend.

De definitie van natuurlijke getallen met een datadeclaratie verloopt volgens een procédé dat al in de vorige eeuw werd bedacht door Giuseppe Peano (al bediende hij zich natuurlijk niet van onze notaties). Het datatype `Nat` (voor "natuurlijk getal") luidt:

```
data Nat = Nul
         | Volg Nat
```

Een natuurlijk getal is dus òf het getal Nul, of het wordt opgebouwd door de constructorfunctie Volg toe te passen op een ander natuurlijk getal. Elk natuurlijk getal kan worden opgebouwd door maar vaak genoeg Volg toe te passen op Nul. Zo kan bijvoorbeeld gedefinieerd worden:

```
een  = Volg Nul
twee = Volg (Volg Nul)
drie = Volg (Volg (Volg Nul))
vier = Volg (Volg (Volg (Volg Nul)))
```

Dit is misschien een wat omslachtige notatie vergeleken bij 1, 2, 3, en 4, maar bij het definiëren van functies en het bewijzen van wetten heb je daar geen last van.

De functie "plus" kan nu met inductie naar één van de twee parameters gedefinieerd worden, bijvoorbeeld de linker:

```
Nul    + y = y
Volg x + y = Volg (x + y)
```

In de tweede regel wordt de te definiëren functie recursief aangeroepen. Dit is toegestaan, omdat x een kleinere datastructuur is dan Volg x. Met behulp van de plusfunctie kan, ook weer met inductie, een vermenigvuldigingsfunctie gedefinieerd worden:

$$\begin{aligned} \mathsf{Nul} \quad * y &= \mathsf{Nul} \\ \mathsf{Volg}\ x * y &= y + (x * y) \end{aligned}$$

Ook hier wordt de te definiëren functie recursief aangeroepen met een kleinere parameter. Met behulp van deze functie kan de machtsverheffingsfunctie worden gedefinieerd, ditmaal met inductie naar de tweede parameter:

$$\begin{aligned} x \hat{} \mathsf{Nul} \quad &= \mathsf{Volg}\ \mathsf{Nul} \\ x \hat{} \mathsf{Volg}\ y &= x * (x \hat{} y) \end{aligned}$$

Nu de operatoren gedefinieerd zijn, is het mogelijk om de rekenkundige wetten te bewijzen. Dat moet in de goede volgorde gebeuren, omdat voor het bewijs van sommige wetten andere wetten nodig zijn. Alle bewijzen verlopen met inductie naar één van de variabelen. Sommige worden zo vaak gebruikt dat ze een naam hebben; sommige worden hier alleen maar bewezen omdat ze in het bewijs van andere wetten nodig zijn. De bewijzen zijn niet moeilijk. Het enige lastige is, om tijdens de bewijzen niet per ongeluk een nog niet bewezen wet te gebruiken omdat het "natuurlijk zo is" – dan zou je wel meteen kunnen stoppen. Dit alles is misschien scherpslijperij, maar het is toch wel eens leuk om te zien dat de bekende wetten ook inderdaad bewezen kunnen worden.

Dit zijn de wetten die we zullen bewijzen:

1.	$x + \mathsf{Nul}$	$=$	x	
2.	$x + \mathsf{Volg}\ y$	$=$	$\mathsf{Volg}\ (x + y)$	
3.	$x + y$	$=$	$y + x$	$+$ is commutatief
4.	$(x + y) + z$	$=$	$x + (y + z)$	$+$ is associatief
5.	$x * \mathsf{Nul}$	$=$	Nul	
6.	$x * \mathsf{Volg}\ y$	$=$	$x + (x * y)$	
7.	$x * y$	$=$	$y * x$	$*$ is commutatief
8.	$x * (y + z)$	$=$	$x * y + x * z$	$*$ distribueert links over $+$
9.	$(y + z) * x$	$=$	$y * x + z * x$	$*$ distribueert rechts over $+$
10.	$(x * y) * z$	$=$	$x * (y * z)$	$*$ is associatief
11.	$x \hat{} (y + z)$	$=$	$x \hat{} y * x \hat{} z$	
12.	$(x * y) \hat{} z$	$=$	$x \hat{} z * y \hat{} z$	
13.	$(x \hat{} y) \hat{} z$	$=$	$x \hat{} (y * z)$	herhaald machtsverheffen

Bewijs van wet 1, met inductie naar x:

IH x	x + Nul	x
Nul	Nul + Nul = (def. +) Nul	Nul
Volg x	Volg x + Nul = (def. +) Volg (x+Nul) = (IH x) Volg x	Volg x

Bewijs van wet 2, met inductie naar x:

IH x	x + Volg y	Volg (x+y)
Nul	Nul + Volg y = (def. +) Volg y	Volg (Nul+y) = (def. +) Volg y
Volg x	Volg x + Volg y = (def. +) Volg (x + Volg y)	Volg (Volg x + y) = (def. +) Volg (Volg (x+y)) = (IH x) Volg (x + Volg y)

Bewijs van wet 3 (plus is commutatief), met inductie naar x:

IH x	x + y	y + x
Nul	Nul + y = (def. +) y	y + Nul = (wet 1) y
Volg x	Volg x + y = (def. +) Volg (x+y) = (IH x) Volg (y+x)	y + Volg x = (wet 2) Volg (y+x)

Bewijs van wet 4 (plus is associatief), met inductie naar x:

IH x	(x+y) + z	x + (y+z)
Nul	(Nul+y) + z = (def. +) y + z	Nul + (y+z) = (def. +) y + z
Volg x	(Volg x + y) + z = (def. +) Volg (x+y) + z = (def. +) Volg ((x+y) + z)	Volg x + (y+z) = (def. +) Volg (x + (y+z)) = (IH x) Volg ((x+y) + z)

Bewijs van wet 5, met inductie naar x:

IH x	x * Nul	Nul
Nul	Nul * Nul = (def. *) Nul	Nul
Volg x	Volg x * Nul = (def. *) Nul + (x*Nul) = (IH x) Nul + Nul	Nul = (def. +) Nul+Nul

Bewijs van wet 6, met inductie naar x:

IH x	x * Volg y	x + (x*y)
Nul	Nul * Volg y = (def. *) Nul	Nul + Nul*y = (def. *) Nul+Nul = (def. +) Nul
Volg x	Volg x * Volg y = (def. *) Volg y + (x*Volg y) = (def. +) Volg (y + (x*Volg y)) = (IH x) Volg (y + (x + (x*y)))	Volg x + (Volg x * y) = (def. *) Volg x + (y + (x*y)) = (def. +) Volg (x + (y + (x*y))) = (+ associatief) Volg ((x + y) + (x*y)) = (+ commutatief) Volg ((y + x) + (x*y)) = (+ associatief) Volg (y + (x + (x*y)))

Bewijs van wet 7 ($*$ is commutatief), met inductie naar x:

IH x	x * y	y * x
Nul	Nul * y = (def. *) Nul	y * Nul = (wet 5) Nul
Volg x	Volg x * y = (def. *) y + (x*y) = (IH x) y + (y*x)	y * Volg x = (wet 6) y + (y*x)

Bewijs van wet 8 ($*$ distribueert links over $+$), met inductie naar x:

IH x	x * (y+z)	x*y + x*z
Nul	Nul * (y+z) = (def. *) Nul	Nul*y + Nul*z = (def. *) Nul + Nul = (def. +) Nul
Volg x	Volg x * (y+z) = (def. *) (y+z) + (x*(y+z)) = (IH x) (y+z) + (x*y + x*z)	(Volg x*y) + (Volg x*z) = (def. *) (y+x*y) + (z + x*z) = (+ associatief) ((y+x*y)+z) + x*z = (+ associatief) (y+(x*y+z)) + x*z = (+ commutatief) (y+(z+x*y)) + x*z = (+ associatief) ((y+z)+x*y) + x*z = (+ associatief) (y+z) + (x*y + x*z)

Bewijs van wet 9 ($*$ distribueert rechts over $+$):

	(y+z) * x = (∗commutatief) x * (y+z) = (wet 8) x*y + x*z	y*x + z*x = (∗commutatief) x*y + x*z

Bewijs van wet 10 ($*$ is associatief), met inductie naar x:

IH x	(x*y) * z	x * (y*z)
Nul	(Nul*y) * z = (def. *) Nul * z = (def. *) Nul	Nul * (y*z) = (def. *) Nul
Volg x	(Volg x*y) * z = (def. *) (y+(x*y)) * z = (wet 9) (y*z) + ((x*y)*z)	Volg x * (y*z) = (def. *) (y*z) + (x*(y*z)) = (IH x) (y*z) + ((x*y)*z)

Bewijs van wet 11, met inductie naar y:

`IH y`	`x^(y+z)`	`x^y * x^z`

`Nul`	`x^(Nul+z)`
	= (def. `+`)
	`x^z`
		`x^Nul * x^z`
		= (def. `^`)
		`Volg Nul * x^z`
		= (def. `*`)
		`x^z + Nul*x^z`
		= (def. `*`)
		`x^z + Nul`
		= (wet 1)
		`x^z`

`Volg y`	`x ^ (Volg y+z)`
	= (def. `+`)
	`x ^ (Volg (y+z))`
	= (def. `^`)
	`x * x^(y+z)`
	= (IH y)
	`x * (x^y * x^z)`
		`x^Volg y * x^z`
		= (def. `^`)
		`(x*x^y) * x^z`
		= ($*$ associatief)
		`x * (x^y * x^z)`

Het bewijs van wet 12 en wet 13 wordt als opgave aan de lezer overgelaten.

Opgaven

13.1 In paragraaf 13.6 wordt de wet p. 249

$$\mathsf{sum}\ (\mathsf{map}\ (1+)\ \mathsf{xs}) = \mathsf{length}\ \mathsf{xs} + \mathsf{sum}\ \mathsf{xs}$$

bewezen. Formuleer een dergelijke wet voor een willekeurige lineaire functie in plaats van $(1+)$, dus

$$\mathsf{sum}\ (\mathsf{map}\ ((\mathsf{k}+)\ .\ (\mathsf{n}*))\ \mathsf{xs}) = \ldots$$

Bewijs de geformuleerde wet.

13.2 Bewijs de volgende

Wet *fold na concatenatie*
Als $(\oplus)$ een associatieve operator is, en e het neutrale element van $(\oplus)$, dan geldt:

$$\mathsf{foldr}\ (\oplus)\ \mathsf{e}\ .\ \mathsf{concat} = \mathsf{foldr}\ (\oplus)\ \mathsf{e}\ .\ \mathsf{map}\ (\mathsf{foldr}\ (\oplus)\ \mathsf{e})$$

13.3 Bepaal een functie g en een waarde e waarvoor geldt:

$$\mathsf{map}\ \mathsf{f} = \mathsf{foldr}\ \mathsf{g}\ \mathsf{e}$$

Bewijs de gelijkheid voor de gevonden g en e.

13.4 Bewijs de volgende wet:

$$\mathsf{length}\ .\ \mathsf{subs} = (2\hat{})\ .\ \mathsf{len}$$

13.5 Bewijs dat subs een combinatorische functie is.

13.6 Bewijs wet 12 en wet 13 uit paragraaf 13.8. p. 256

13.7 Bewijs dat de volgende eigenschap geldt: reverse . reverse = id, waarbij de functie id gedefiniëerd is als:

```
id x = x
```

Gebruik inductie, d.w.z. laat zien dat:

```
(reverse . reverse) []       = id []
(reverse . reverse) (x : xs) = id (x : xs)
```

14 QuickCheck

For a discussion of QuickCheck, consult Chapter 11, Testing And Quality Assurance of Real World Haskell [8], to be found at

`book.realworldhaskell.org/read/testing-and-quality-assurance.html`.

Exercises

For the exercises below you may want to consult the functions provided by the QuickCheck library, at

`hackage.haskell.org/package/QuickCheck-2.4.2/docs/Test-QuickCheck.html`,

in particular functions such as choose, sized, elements and frequency. We encourage experimenting with your code in a `ghci` session. To be able to experiment with QuickCheck, the first two exercises work better if you can show functions. For that you can add the following instance definition to your code:

```
instance (Enum a, Bounded a, Show a) => Show (a -> Bool) where
  show f = intercalate "\n" (map (\x -> "f " ++ show x ++ " = " ++
             show (f x))[minBound .. maxBound])
```

Also when you run your tests, you sometimes need to specialize the types a bit. For example, Fig. 14.1 shows the code that calls all kinds of test functions that the exercises below (except for 14.4) expect you to come up with.

14.1 Consider the ubiquitous filter function. There are many properties that you can formulate for the input-output behaviour of filter.

- Formulate the QuickCheck property that the result list cannot be longer than the input.
- Formulate the QuickCheck property that all elements in the result list satisfy the given property.
- Formulate the QuickCheck property that all elements in the result list are present in the input list.

```
runTests :: IO ()
runTests = do putStrLn "\nExercise 14.1"
  quickCheck (propFilterNoLonger :: (Bool -> Bool) -> [Bool] -> Bool)
  quickCheck (propFilterNoLongerWrong :: (Bool -> Bool) -> [Bool] -> Bool)
  quickCheck (propFilterAllSatisfy :: (Bool -> Bool) -> [Bool] -> Bool)
  quickCheck (propFilterAllElements :: (Bool -> Bool) -> [Bool] -> Bool)
  quickCheck (propFilterCorrect :: (Bool -> Bool) -> [Bool] -> Bool)
  putStrLn "\nExercise 14.2"
  quickCheck (propMapLength :: (Bool -> Bool) -> [Bool] -> Bool)
  putStrLn "\nExercise 14.3"
  quickCheck $ once (propPermsLength :: [Int] -> Bool)
  quickCheck $ once (propPermsArePerms :: [Int] -> Bool)
  quickCheck $ once (propPermsCorrect :: [Int] -> Bool)
  putStrLn "\nExercise 14.5"
  quickCheck (forAll genBSTI isSearchTree)   -- Use forAll to use custom generator
  quickCheck (forAll genBSTI propInsertIsTree)
  quickCheck (forAll genBSTI propInsertIsTreeWrong)
```

Fig. 14.1: Running all the properties, with type annotations

- Formulate a set of QuickCheck properties to completely characterize the filter function (you may choose also from among the three you have just implemented). Make sure to remove properties that are implied by (a subset of) the other properties.

14.2 Try to come up with a number of QuickCheck-verifiable properties for the map function, and implement these. Are there any properties of map that are awkward to verify?

14.3 Consider the function perms from section 7.1. We shall be writing QuickCheck tests to verify that this function p. 144

- Write a QuickCheck property that checks that the correct number of permutations is generated.
- Write a function isPerm :: $[a]$ -> $[a]$ -> *Bool* that verifies that the two argument lists are permutations of each other.
- Write the QuickCheck property that every list in the output of perms is a permutation of the input.
- Formulate a set of properties to completely characterize the perms function (you may choose also from among the ones you have just implemented). Make sure to remove properties that are implied by (a subset of) the other properties. Implement the properties that you still need as QuickCheck properties.

14.4 Do something similar for subs from section 7.1. p. 144

14.5 Consider the following datatype definition for binary trees that we shall want to use to implement binary search trees (translated from paragraph 6.7). p. 136

```
data Tree a = Branch a (Tree a) (Tree a)
            | Leaf
```

In order to test operations on binary search trees we need to randomly generate binary search trees. Write a generator `genBSTI :: Gen (Tree Int)` for binary search trees that contain integers. We suggest the following approach:

- generate the tree from the root, unfolding it as you go,
- randomly generate the content of the branch nodes, making sure that the randomly generated value does not break search tree property,
- you must ensure that unfolding the search tree eventually stops. You can do so by, as it were, randomly flipping a coin, and if it's head choose a *Tree* at that point in the tree, and a Branch otherwise. (note that you may want to tweak the chances of getting leaf a bit).

To test your generator write a function `isSearchTree :: Tree a -> Bool` that verifies that its argument is a binary search tree. Then use your test generator to test the property that given a binary search tree `t`, inserting a value into the tree results in yet another binary search tree. The code for inserting a new value into the tree is the following (translated from paragraph 6.7) p. 136

```
insertTree          :: Ord a => a -> Tree a -> Tree a
insertTree e Leaf = Branch e Leaf Leaf
insertTree e (Branch x li re) | e <= x = Branch x (insertTree e li) re
                              | e > x  = Branch x li (insertTree e re)
```

Experiment with mutating the implementation of `insertTree` to find out whether your generator and property can in fact discover that the mutated implementation no longer maps binary search trees to binary search trees.

15 Lazy evaluation

15.1 Introduction

In the preface we have mentioned that one of the distinguishing features of Haskell is the order in which expressions are evaluated. In imperative languages one normally does not pay attention to this, since usually the order in which commands are executed is quite obvious. Nevertheless one might wonder what values are precisely passed onto the function f in the program $x := 1; f\ (x\ ++, x\ ++)$; in some languages this is even explicitly left unspecified so the compiler can make its choice. Note also that this program is quite different from $x := 1, temp := x\ ++; f\ (temp, temp)$.

It should be clear by now that in a purely functional language, in which the evaluation of expressions does not have side-effects, we can take a much more liberal approach. As a second example consider an **if**-statement:

if < cond > **then** < expre1 > **else** < expr2 >

and now assume we abstract (not a very likely thing to do) from **if**-expressions :

myIf c e1 e2 = **if** c **then** e1 **else** e2

because we do not like to write keywords like **then** and **else**. Unfortunately this is not possible in most languages, since the expression

myIf *True* e1 e2

will diverge if the expression e2 diverges.[1] It is for this reason that one finds often, besides a proper and also a so-called conditional cand which only evaluates its second argument if the first argument evaluates to *True*. You may wonder whether this operator can be freely passed as a argument, and if so, what are the consequences of this? As we will see there is in Haskell no objection to define a function such as myIf yourself.

In the next section we will give a fairly precise description of lazy evaluation; afterwards, we provide some typical examples of where this evaluation strategy pays off in terms of clarity or efficiency of code.

[1] A ternary operator of this kind, written `cond ? thenexpr : elseexpr`, is part of the C-language and many of its successors. It only evaluates the thenexpr if the condition is true, and similarly for elseexpr. But this lazy behavior is only possible here since the ternary operator is hard-wired in the language.

15.2 The rules for lazy evaluation

In Haskell, evaluation takes place on a so-called *e*valuate-by-need basis, i.e.:

- an expression is only evaluated when its result is really needed
- an expression is no further evaluated than needed
- results of an evaluation are kept, so an expression is never completely re-evaluated.

To demonstrate how Haskell's evaluation strategy works we consider the following expression:

```
length (take 2 (repeat (2 + 3)))
```

We see that the top level of this expression is a call to the function length:

```
length (_ : xs) = 1 + length xs
length []       = 0
```

Actually the use of patterns is not really needed, and the function definition can be desugared into:

```
length l = case l of
             (_ : xs) -> 1 + length xs
             []       -> 0
```

So by substituting the right hand side of this function definition in the expression we get:

```
case (take 2 (repeat (2 + 3)) of
  (_ : xs) -> 1 + length xs
  []       -> 0
```

In order to choose which case branch to take, we are forced to evaluate the expression take 2 (repeat (2 + 3)) a bit, so we substitue the (also desugared) right hand side of the function take:

```
take n l = case n of
  0 -> []
  n -> case l of
    []       -> error "insufficient elements in the list"
    (x : xs) -> x : take (n − 1) xs
```

which results in:

```
case (case 2 of
  0 -> []
  n -> case repeat (2 + 3) of
    []       -> error "insufficient elements in the list"
    (x : xs) -> x : take (n − 1) xs
  ) of
  (_ : xs) -> 1 + length xs
  []       -> 0
```

Now we can make some progress since clearly $2/=0$ so we simplify the case expression and get:

```
case (case repeat (2 + 3) of
        []       -> error "insufficient elements in the list"
        (x : xs) -> x : take (2 − 1) xs
      ) of
  (_ : xs) -> 1 + length xs
  []       -> 0
```

Again the outer case expression cannot make any progress as long as the inner case expression has not returned at least some part of its result, so we proceed again by unfolding the definition of repeat:

```
case (case (2 + 3) : repeat (2 + 3) of
        []       -> error "insufficient elements in the list"
        (x : xs) -> x : take (2 − 1) xs
      ) of
  (_ : xs) -> 1 + length xs
  []       -> 0
```

Now we see a non-empty list so we can make the choice for the inner case, and get:

```
case (2 + 3) : take (2 − 1) (repeat (2 + 3))
      ) of
  (_ : xs) -> 1 + length xs
  []       -> 0
```

Now at last the outer case expression has sufficient information to make its choice, and the expression simplifies to:

```
1 + length (take (2 − 1) (repeat (2 + 3)))
```

At this stage the top-level function in the expression is +, for which its first argument is fine (it can be used in an addition), but the second argument is still an expression. Skipping some steps we finally get to the state where the expression to be evaluated is:

```
1 + (1 + 0)
```

which evaluates to $1 + 1$ and then finally the top level + can do its work, and the final result becomes 2.

Based on this example we conclude that evaluation in Haskell is driven by:

- the need to choose between branches of either an explicit **case** expressions or an implicit **case** expressions which results from the use patterns in function definitions
- or because some primitive function like + really needs its completely evaluated argument before it can compute its result. For such functions we say that they are *strict*, i.e. they will enforce the evaluation of their arguments when a call to the function shows up at the top level of the expression under evaluation

We may summarise Haskell's evaluation strategy by:

> "Evaluation is driven by pattern matching."

It is also important to note that the function repeat only returned that part of its result that was needed by the surrounding case-expression, and nothing more. Also note that the expression $2+3$ never got evaluated; its result was not needed to compute the length of the intermediate list.

In order to see how subtle this process is we slightly change the definition of the function take into:

```
take 0 l = []
take n l = head l : take (n − 1) (tail l)
```

and wonder what the result is of the expression length . take 3 $ undefined? The difference with the previous definition of take is that the point where the elements are taken from the argument list is delegated to the pattern matching performed by the functions head and tail. The value being passed to the function length is best described by:

```
head undefined : head (tail undefined) : head (tail (tail undefined)) : []
```

and we see that for the function length it contains sufficient information to conclude that the length is 3.

Since our new definition of take does not evaluate anything of its list argument we say that the function is *non-strict* in its second argument. Since it is guaranteed to evaluate its first argument we say it is strict in its first argument.

15.3 How lazy evaluation can make your life easier

Since languages such as Haskell, Java and C# perform automatic garbage collection *we do not have to worry about when the life of a value ends*! This takes away one of the most important sources of programming mistakes when using languages and systems where the programmer has to take care of this explicitly in the program text. With lazy evaluation we get something similar: *we do not have to worry about when the life of a value starts*! It suffices to inform the system of how the value is computed, might the need for it arise. If the need does not arise even better, since in that case no superfluous work was done.

When in an imperative program an assignment statement is executed we are likely to see this as a new value being assigned to a variable. The programmer however provides more information: he has made explicit in the program text that the old value of the variable is no longer needed. In this sense he has explicitly scheduled the computations and the way values are stored and kept more; this may lead to an increase in efficiency by avoiding garbage collections, but it also makes that a number of design decisions end up

hard coded all over the program, which is likely to make it less adaptable and probably even impossible to abstract from.

Unfortunately there are no such things as a free lunch, and programs written in lazy evaluated languages may exhibit unexpected memory requirements, which may not always be easy to find the cause of. In this case we may have to resort to techniques as heap-profiling to see what actually is going on when the program is evaluated. In general any formal reasoning about space and time behaviour gets more complicated since information is only available in a very implicit form.

The availability of lazy evaluation brings forward a whole series of new programming techniques, to some of which we will pay attention in this chapter. We mention a some (but definitely not all)

- describing process like structures
- describing recurrent relations
- decoupling the generation of search spaces and the actual searching process

15.3.1 Communicating processes

In Unix based systems we have the so-called *pipe* construct, where output from one process is used as the input for the next process. As an (not so realistic) example we might want to concatenate all the files in a directory that have an lhs (literate Haskell script) extension into a single file, run this file through a program called lhs2TeX, which inserts formatting commands to make sure the Haskell code in the TEX-files look nice, and finally run the resulting text through the LATEX program which will take the formatting commands and produce nicely formatted output. The Unix shell command for doing this might look like:

```
ls | grep ".lhs" | concat | lhs2TeX | pdflatex
```

When running this program the intermediate files never come completely into existence; generated parts are already consumed before the intermediate structures have been completely constructed. In Haskell we can achieve this effect by modelling the intermediate structures as list, and using functions which map lists onto lists.

As an example we model two simple processes process_p and process_q, mapping a function p respectively q over their input:

```
process_p = map p
process_q = map q
```

Each process is furthermore consuming the output produced by the other, and we start the whole computation by feeding a 1 to process_p. The result we are interested in is the output of process_q:

```
let pout = process_p pin
    qout = process − q qin
    pin  = 1 : qout
    qin  = pout
in qout
```

or shorter:

```
let loop f = let out = f out in out
in loop (map p . (1:) . map q)
```

Using this technique we can build arbitrarily complex process networks. It is of course important to make sure that some form of progress is always possible, i.e. there is always a process for which some input is available.

15.3.2 Eratosthenes' sieve

A famous algorithm, attributed to Eratosthenes, computes prime numbers as follows:

(i) take an increasing list of candidates containing all natural numbers except 1

(ii) we see that 2 is at the head of this list and conclude that it is the next prime number

(iii) remove all multiples of 2 from the list of candidates

(iv) the smallest remaining number in the list is 3, so conclude that 3 is a prime number and remove all multiples of 3 from the list of candidates

(v) the smallest remaining number is 5, so ...

We start out by defining a small helper function which removes all multiples of a number from a list:

```
removeMultiples n = filter ((/= 0) . (`mod` n))
```

The next step is to locate the numbers which show up at the beginning of the list, make them part of the result since these are the sought prime numbers, remove the multiples of these numbers from the rest of the list, and continue with the sifting process. This is concisely expressed by:

```
sift (p : xs) = p : sift (removeMultiples p xs)
```

By applying this function sift to the (infinite) list of initial candidates $[2..]$ we are done:

```
primes = sift [2..]
```

One way of looking at this code is to see it as initial process sift which creates an extra filtering process on its input for each encountered prime number. We conclude that the network of communicating processes is dynamic: it changes during the evaluation of the expression.

15.3.3 Hamming's problem: productivity

The so-called Hamming's problem asks you to generate a list of values, which have as only prime factors the numbers 2, 3 and 5, i.e. ($\{2^i 3^j 5^k | i >= 0, j >= 0, k >= 0\}$). The list should be generated in increasing order.

The typical way to approach such a problem is to start with an inductive definition of Hamming numbers:

(i) 1 is a Hamming number.

(ii) if n is a Hamming number then also $2 * n$, $3 * n$ en $5 * n$ are Hamming numbers.

(iii) purists add "And there are no other Hamming numbers", but for computer scientists this is natural, so we do not bother to mention it (did we?).

We now reason as follows:

(i) suppose that ham is the list sought, then the lists map (∗2) ham, map (∗3) ham, and map (∗5) ham also contain Hamming numbers; in other words, each Hamming number (except 1) has a predecessor in at least one of these lists (and most likely in all three)

(ii) if ham is *monotonically* increasing then this holds for these three lists too

The above observation immediately leads to the following code, in which we start the list of Hamming numbers with 1, generate new list of Hamming numbers from the computed hamming numbers, merge these lists so the resulting list is still in increasing order, and remove any duplicates values which come from different merged lists:

```
ham = 1 : remdup ((map (*2) ham)
                    `merge`
                  (map (*3) ham)
                    `merge`
                  (map (*5) ham)
        )
merge xxs@(x : xs) yys@(y : ys) | x <= y = x : merge xs  yys
   | otherwise                           = y : merge xxs ys
remdup (x : ys) = x : remdup (dropWhile (== x) ys)
```

When we replace the definition of remdup by:

```
remdup (x : y : zs) | x == y    =    remdup (y : zs)
                    | otherwise = x : remdup (y : zs)
```

then it looks at first sight that nothing has changed. It may come as a surprise however that something has changed: after producing a 1 the computation stops to produce any further results.Why?

After a couple of evaluation steps we get to the following situation:

```
ham = 1 : remdup (2 : (((2 * (head (tail ham)) : map (*2) (tail (tail ham))
                          `merge`
                         (3 : map (*3) (tail ham))
                        `merge`
                       (5 : map (*5) (tail ham))
                  )
```

Because of the pattern (x:y:zs) our new definition of remdup is not only interested in the value 2, but also in the value following that 2. But for this the function calls to merge have to produce a result first, for which the value of $2 * (\mathsf{head}\ (\mathsf{tail}\ \mathsf{ham}))$ is needed, in which we refer to tail ham, of which we know it will be the value 2. But we will not produce that value before we have seen a value depending on that 2. So the evaluation gets stuck.

When we compare the two definitions of remdup:

```
remdup (x : y : zs) | x == y    =     remdup (y : zs)
                    | otherwise = x : remdup (y : zs)
remdup' (x : ys)                = x : remdup' (dropWhile (== x) ys)
```

when applied to the sequence [1, <expr1>, <expr2>] then the first definition needs the result of <expr1>, before it yields the 1. The second definition yields the 1 directly.

We say that the second definition is *less strict* than the first one: if both definitions return something then these values will be the same, but the second definition will evaluate a smaller part of its argument and it will even produce part of an answer when applied to [1, undefined].

15.4 Memoisation

This section consists of a non-trivial case study in which we demonstrate various programming techniques. It deals with a technique called *memoisation* which can be used to make many recursive functions more efficient and lies at the heart of a technique called *dynamic programming*.

15.4.1 Timing behaviour of computing Fibonacci numbers

For the sake of demonstration we go back to the well known function which computes the so-called Fibonacci [2] numbers:

[2] Leonardo of Pisa (±1170 – ±1250):

```
fib :: Integer -> Integer
fib  0       = 0
fib  1       = 1
fib  n       = fib (n - 2) + fib (n - 1)
```

When we time this function for various inputs we get:

```
*Main> ghci TimeFib.hs
...
*Main> :set +s
*Main> fib 20
6765
(0.02 secs, 5491520 bytes)
*Main> fib 30
832040
(2.23 secs, 520944784 bytes)
*Main> fib 35
9227465
(24.79 secs, 5762515704 bytes)
*Main>
```

We see that for larger input values the time it takes to compute the result increases rapidly; the number of calls to fib are:

value of n	number of fib calls
5	15
10	177
15	1973
20	21891
25	242785
30	2692537

Why this is the case becomes clear when we take a look at the *c*all tree in figure 15.1 where we see that the function fib is evaluated three times with argument 2. The question we will answer in the rest of this chapter is how to avoid such duplicate calls.

15.4.2 Local memoisation

From figure 15.1 it becomes clear that we are well on our way to solving the problem, if for each call to fib we remember the argument and the corresponding result. This allows us to re-use the result the next time the same argument is passed in a call to fib. One solution is to use an *a*rray in which we store the results of all the needed calls, using the argument to fib as the index:

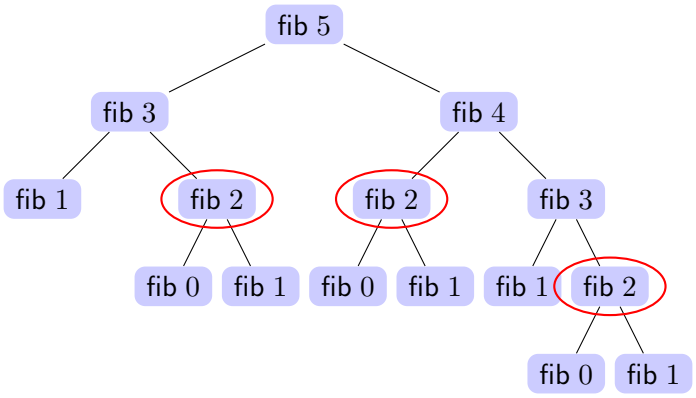


Fig. 15.1: The call tree for fib 5

```
import Data.Array

fib :: Integer      -> Integer
fib  n              =  fibs ! n
  where fibs = listArray (0, n) $
               0 : 1 : [fibs ! (k − 2) + fibs ! (k − 1) | k <- [2 .. n]]
```

So instead of computing the value of fib all over again, we use the right entry in the array (the operator ! retrieves the result associated with a particular argument). Each value in the array is defined in terms of (earlier) elements in the array; it is no problem to define elements in terms of other elements, since lazy evaluation will ensure a proper evaluation order. Note however that no element is defined in terms of itself; so we easily see that the definitions are productive and indeed will become available when the need arises; we again see lazy evaluation at work.

15.4.3 Global memoisation

A disadvantage of this approach is that when we have several calls to fib in our program we reconstruct such an array for each individual call; computed results are not shared between individual calls to fib. So the question arises whether we can cure this problem. And indeed we can, we call this *global memoisation*.

A global memo function

- also remembers the results of *previous calls* made elsewhere in the program,
- remembers the result for *all* arguments ever passed.

We will eventually define a function memo :: (*Integer* -> *a*) -> (*Integer* -> *a*), which takes a function of type *Integer* -> *a* and maps it onto its memoised counterpart.

Fixed-point combinator

An important step in our approach is to get access to all calls to fib and to relate them to each other, especially the recursive calls in the function body itself. From this we may already conclude that in order to get a memoising version of fib we will have to write this function fib in a somewhat special way, since there is no way in which we otherwise can locate these recursive calls; a program cannot inspect itself. For this we introduce the fixed-point combinator, which computes the fixed point of a function, i.e. a point for which it holds that f x equals x (note that functions can have more than one fixed point, e.g. id).

A *fixpoint combinator* is a higher-order function which "computes" the fixpoint of a function:

```
fix :: (a -> a) -> a
fix   f = f (fix f)
```

or by sharing calls to fix:

```
fix :: (a -> a) -> a
fix   f = let fixf = f (fixf) in fixf
```

Note that when f in the definition above has type (*Integer* -> *b*) -> (*Integer* -> *b*), which will be the case when we use fix, then the instance of fix is of type

```
fix :    ((Integer -> b) -> (Integer -> b) -> (Integer -> b) -> (Integer -> b))
   -> ((Integer -> b) -> (Integer -> b))
```

Using fix we can make the use of *recursion* explicit, which we will demonstrate using the factorial function:

```
fac :: Integer -> Integer
fac  0        = 1
fac  n        = n * fac (n - 1)
```

which we can, using fix, be rewritten as:

```
fac :: Integer -> Integer
fac = fix fac' where fac' f 0 = 1
                     fac' f n = n * f (n - 1)
```

So the idea is to introduce an extra parameter which is used in the recursive calls, and which actually is the recursive function we want to define. What we have succeeded in doing now is replacing all recursion by making use of a single recursive definition in the body of fix. Let us unroll the definition a couple of times to see why this definition works:

```
    fac 3
=     -- unfold fac
    fix fac′ 3
=     -- unfold fix
    fac′ (fix fac′) 3
=     -- unfold 2nd case arm
    3 ∗ fix fac′ (3 − 1)
=     -- evaluate 3 - 1
    3 ∗ fix fac′ 2
=     -- unfold fix
    3 ∗ fac′ (fix fac′) 2
=     -- unfold 2nd case arm
    3 ∗ (2 ∗ fix fac′ (2 − 1))
=     -- evaluate 2 - 1
    3 ∗ (2 ∗ fix fac′ 1)
```

The fix function creates a sufficient number of copies of the body of fac′ so we cannot see the difference with an infinite unfolding fac′ (fac′ (fac′ . . .)) which is implied by the original recursive definition.

15.4.4 Memoising fixpoint combinator

The function fix is the function which "constructs" the final fib function which we will call all over the program; the approach we take is that we *replace it with a memoising version*. So our plan of attack consists of the following steps:

- choose a *parameterised* datatype Memo for the memo tables
- define functions tabulate and apply,

  ```
  tabulate :: (Integer -> a) -> Memo a
  apply    :: Memo a -> Integer -> a
  ```

  such that:

  - tabulate f results in a *lazily constructed* memo table containing all results of possible calls to f
  - apply mem n retrieves the corresponding value for the parameter n from mem
- define a fixed point combinator memo using tabulate and apply

In our first approach we will represent memo tables as *infinite lists*:

```
type Memo a = [a]
```

For this the definitions of the functions tabulate and apply are almost trivial:

```
tabulate :: (Integer -> a) -> Memo a
tabulate   f                  =  map f [0..]

apply :: Memo a -> Integer -> a
apply   (x : _)  0 = x
apply   (_ : xs) n = apply xs (n − 1)
```

Now we can define our memoising fixpoint combinator as:

```
memo ::  ((Integer -> a) -> (Integer -> a)) -> (Integer -> a)
memo     f'                                  =  f
  where f = apply (tabulate (f' f))
```

- this combinator constructs a fixpoint f of f′
- the function f retrieves its result using apply from the memo table tabulate (f′ f)
- each element in the table is computed using f′
- recursive calls use the memoised function f
- thanks to lazy evaluation only those elements in the list are computed which are really used in constructing the resulting value
- the table does not depend on a parameter of f; calls to f share the table which is *persistent during the evaluation of the program*

15.4.5 Timing results

In order to see whether we really have achieved what we were looking for we look again at some timing results (counting the numer of reductions):

```
*Main>  fib 10
55
1450 reductions, 2316 cells
*Main> fib 20
6765
5060 reductions, 8178 cells
*Main> fib 25
75025
7690 reductions, 12463 cells
*Main> fib 30
832040
10870 reductions, 17649 cells
```

So we made a considerable progress. But it is even better, since when we have subsequent calls we get almost immediate results:

```
*Main> fib 30
832040
10870 reductions, 17649 cells
*Main> fib 30
832040
359 reductions, 583 cells
```

In the second call all we have to do is to look up the result in the table.

15.5 Memo trees

When we consider our first approach (using arrays) then we see that each lookup of a value takes constant time but that the maximum index is fixed, whereas in our memoising version we have linear lookup time but no upper bound on the possible index. So our next step is to give new definitions of Memo, tabulate and apply such that we get a logarithmic lookup time, while keeping the possibility for an unbounded number of different arguments.

For this we define two mutually recursive data structures, for which we refer to figure 15.2. What we see is an infinite binary tree with values in the root and each second child, i.e. when we have gone down to the right in the tree. Each of these nodes is indexed by an *Integer* value, and in the node we store the result of applying the memoised function to this index. The index of a node can be computed by adding the numbers on the path from the root to this node. We see that the contribution to this index value doubles at each level; when we go right we include the contribution of this level, and when we go left we do not take it into account. So the algorithm for finding where the function value for a specific parameter is located proceeds as follows:

(i) if the index value has become 0 (i.e. no 1 bits are present), we have reached the node we are looking for

(ii) if the right most bit of the representation of the index value is 1 go down to the right, otherwise left

(iii) shift the bit representation one position to the right and continue at step 1

So our new data type Memo, which holds a memoised value, comes with a sibling Memo′ which does not hold a value in its root node:

$$\textbf{data}\ \mathsf{Memo}\ \ a = \mathsf{Memo}\ \ (\mathsf{Memo}'\ a)\ a\ (\mathsf{Memo}\ a)$$
$$\textbf{data}\ \mathsf{Memo}'\ a = \mathsf{Memo}'\ (\mathsf{Memo}'\ a)\quad (\mathsf{Memo}\ a)$$

The function tabulate which constructs the memo tree uses two helper functions which do the actual work: one constructing a Memo value and the other one a Memo′ value.

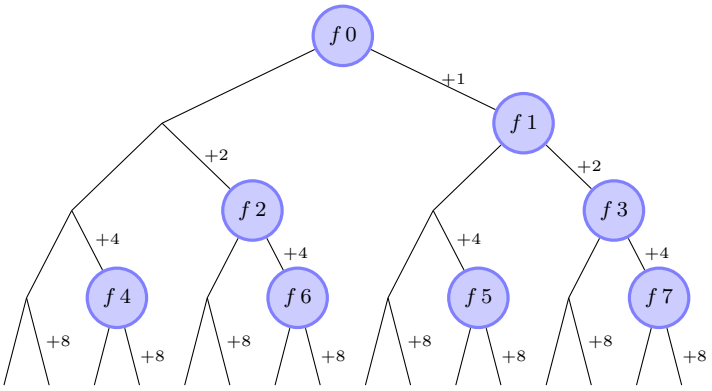


Fig. 15.2: An example memo tree

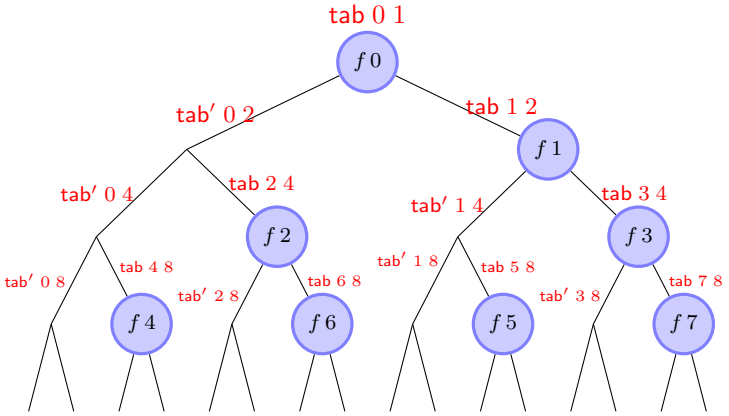


Fig. 15.3: The construction of the memo tree

The parameter k holds the sum of the contributions of the path to this node, and the i parameter holds the contribution made by this level in the tree:

```
tabulate :: (Integer -> a) -> Memo a
tabulate  f = tab 0 1
  where
    tab  k i = let j = 2 * i in Memo  (tab' k j) (f k) (tab (k + i) j)
    tab' k i = let j = 2 * i in Memo' (tab' k j)       (tab (k + i) j)
```

In figure 15.3 we see the calls which have generated the corresponding elements of the memo tree:

Locating the value for a specific index is the reverse process. By looking at the last bit of the binary representation we know whether to go down left or right; if no 1's are left we have reached the node we are looking for. At each recursive step the search key is halved and we descended one level in the tree:

```
apply :: Memo a -> Integer -> a
apply = app
  where
    app (Memo l x r) n | n == 0    = x
                       | even n    = app' l (n `div` 2)
                       | otherwise = app  r (n `div` 2)
    app' (Memo' l r) n | even n    = app' l (n `div` 2)
                       | otherwise = app  r (n `div` 2)
```

If the key reaches 0, we return the value in the current node.

If we now look at the timing result we get:

```
*Main> fib 5000
3878968454388325633701916308325905312082127714
0.37 secs, 26809216 bytes
*Main> fib 5000
3878968454388325633701916308325905312082127714
0.02 secs, 532752 bytes
```

We summarise the main results of this approach:

- the more efficient table structure requires some programming effort, but is a "one-time investment" (and maybe not even your own!)
- the choice of data structure is invisible to user of the library
- only a single action is required from the user: making the recursion explicit
- we can extend the memoisation for any kind of value that can be mapped onto an *Integer*
- functions with more than one parameter can be memoised by having memo tables containing memo tables and using successive lookups
- Haskell packages are available which use these techniques (e.g. `http://hackage.haskell.org/package/MemoTrie`

15.6 Reflection

Having come to the end of this chapter, in which we have seen many examples of where lazy evaluation can be put to good use, we should be honest and also mention some of the problems that come with lazy evaluation.

In the first place we have seen that we can write complicated recursive structures, but we are not liberated from the obligation to verify that all our definitions really make sense. And worse, if our definitions are not productive, we do not get a nice error message telling us where to look for the problem. It can be very difficult and time consuming to find out why some collection of mutually recursive definitions does not produce a result

at all!

A second disadvantage of the taken approach in the memoisation case is that it may be very space consuming. Even after the last call to the memoised function has been made this memo tree is kept in memory since in general it is not known whether any further calls will be made in the future. You, as a programmer, may know that this is not the case; but you cannot rely on the fact that the system will find this out for you. Sometimes a solution is to use a local **let** construct in which you introduce the memoised function. In that case the structure will be garbage collected as soon as the result from this **let** does not contain any unevaluated expressions referring to this function.

15.7 Seq, deepseq and being strict

What often happens is that as a side effect of computing some value you are interested in, you compute a second value (in the form of a yet unevaluated large expression) in which you are interested much later on. In some cases, it may be helpful to evaluate such an expression at an earlier stage. For example, when passing such an unevaluated expressions to a function which is known to always use the value of the expression, we can safely evaluate the expression *before* calling the function, since they will be needed anyway.

To put this on a more precise footing, we call a function f *strict* in its argument, if f $\perp$ equals $\perp$. Here $\perp$ signals abnormal termination, or crash. Haskell provides a value for $\perp$ as well: undefined :: a, which is universally polymorphic value: the type system will allow to write it anywhere. But if during evaluation, you need to know the *value* of undefined the program will crash. A variant of undefined is the function error :: *String* -> a that behaves like undefined, but that has an argument of type *String* that provides some additional information about the crash:

```
fac :: Int -> Int
fac 0 = 1
fac n | n < 0 = error "Don't ``fac'' with a negative argument, moron"
      | otherwise = n * fac (n - 1)
```

```
*Main> fac (-2)
*** Exception: Don't ``fac'' with a negative argument, moron
```

The motivation for strictness is that if you can expect a function call to crash when passed a crashing argument, it will not do any harm to evaluate the argument to the function *before* passing it in instead of letting the function decide what part of the argument it needs (which would normally be done in the case of lazy evaluation).

The reason for wanting to evaluate the argument sooner rather than later is that the volume of memory taken up by your program (space times the time that space is al-

located) can sometimes dramatically decrease if you evaluate some expressions sooner. But since we do not want to run the risk of accidentally turning a normally terminating into an abnormally terminating one, we would like to be sure that this change does not affect the outcome.

The GHC performs strictness analysis for you, but sometimes it pays to help the compiler a bit by adding your own strictness annotations: you can do so by putting strictness annotations in the form of an exclamation mark in the parameter list of a function. Similarly, you can turn fields in a datatype into strict fields. Indeed, a few ! marks can do wonders to memory use and used processor time of your program. But if you do not know what you are doing, this may also work against you: by putting annotations where they do not belong expressions may be evaluated that are in fact not needed, potentially consuming lots of time and space. (On a personal note, many of the performance problems I have seen and solved in Haskell applications arise due to a lack of space, resulting in the garbage collector being run very often with only little effect, leading to a kind of "thrashing" behaviour reminiscent of file systems of some decades ago.)

15.7.1 Strictness annotations

So how does this all work out in your Haskell code? Consider the following piece of Haskell code, taken from the Real World Haskell book (Chapter 25):

```
data Pair = Pair !Int !Double

mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    Pair n s = foldl' k (Pair 0 0) xs
    k (Pair n s) x = Pair (n + 1) (s + x)
```

It defines a pair of strict *Int* and Double, by virtue of the exclamation marks, that directly precede the types *Int* and Double. This means that whenever such a pair is constructed, the expressions are evaluated into Weak Head Normal Form (WHNF). This means that it will be evaluated until the outer constructor becomes known. In the case of booleans and integers this means that the integer and boolean values are completely known, but for lists this means that we only know that the outer constructor is either a (:) or a [], but the arguments to (:) may not be evaluated.

You may have encountered the explicit application operator ($) :: $(a \to b) \to a \to b$ before. Ordinary function application has the highest priority of all. Nested calls are then written: sum (map f xs), to signal that f and xs are arguments of map, not of sum. Some people dislike the use of parentheses and instead prefer to write: sum $ map f xs. The effect is that ($) breaks the expression in two: everything to the left and to the right of it. It then applies the result of evaluating the former to the latter. As usual in

Haskell, a thunk representing the complete right expression, map f xs, is passed to sum, and it may decide to evaluate it or not.

For ($) there is also a strict version, the ($!) operator. It operates as you may expect: before passing the argument expression map f xs into sum, for sum $! map f xs, map f xs will be evaluated to WHNF.

Another way to to influence the strictness of evaluation is by putting seqs in Haskell programs. The function seq has the same type as the const function: $a \to b \to b$, both deliver the value of their right argument and "forget" the first. But there is a significant difference as well: seq will in fact evaluate its first argument (of type a) to WHNF, before returning its right argument.

This allows you to define the following identity function that, as a side effect, evaluates its argument to WHNF:

```
whnf :: a -> a
whnf x = x `seq` x
```

Consider now the following expression which uses foldl to compute the sum of values in a *long* list:

```
Prelude> import Data.List
Prelude Data.List> :set +s
Prelude Data.List> foldl (+) 0 [1..10000000]
50000005000000
(7.24 secs, 1694522336 bytes)
```

So, to sum one million integers, we need more than seven seconds and a 1.6 GB of memory. That is quite a lot. Why may that be?

Consider the definition of foldl:

```
foldl (⊕) e []       = e
foldl (⊕) e (x : xs) = foldl (⊕) (e ⊕ x) xs
```

Although we know that we will need all the values from the list, we are building up a huge expression (thunk) in the "accumulating parameter" e. Only when we deliver e at the end will the interpreter force evaluation of the thunk to its integer value in order to print it on screen. One way to improve this is to pass not $e \oplus x$ to foldl, but whnf $(e \oplus x)$ instead. Since the result is of type integer, the consequence is that the accumulating parameter will in fact only store integer values, not expressions that can evaluate to the integers.

And in fact the module *Data*.List provides a foldl′ function that does exactly that. And this is what we get:

```
Prelude Data.List> foldl' (+) 0 [1..10000000]
50000005000000
(0.37 secs, 962205176 bytes)
```

Memory consumption is substantially reduced, and running time more than that.

15.7.2 Seq versus deepseq

Strictness annotations only lead to values being evaluated to WHNF. If the constructor is in fact a constant (like *True*, 2 and []), then everything there is to know about this value is known. But sometimes you want more. For example, when you read in a file as a list of characters, you often know that you will be needing it all. It typically pays to read in the entire file immediately. But a list is not a primitive value: if you seq the reading of the file, you will know immediately that the file has ended or not, but that is all: you will not have read in the complete file. To achieve that you need to use deepseq. What does deepseq do? It simply takes a value of some type, and starts by seq'ing the top-level constructor, and then it will (normally) recursively deepseq the arguments to the top-level constructor (although what is evaluated precisely is in fact up to the programmer). As you can imagine it is not wise to deepseq an infinite computation, as we do in the following example where we use the function force that is like whnf, but use deepseq instead of seq:

```
Prelude> head (whnf [1..])
1
Prelude> :m Control.DeepSeq
Prelude Control.DeepSeq> head (force [1..])
...expect a long wait...computer may hang...
```

The documentation of the module *Control*.DeepSeq illustrates by the following example:

```
import System.IO
import Control.DeepSeq

main = do
  h <- openFile "f" ReadMode
  s <- hGetContents h
  s `deepseq` hClose h
  return s
```

The function deepseq is used just like seq: the left argument is forced (but now deeply, as opposed to superficially with seq), and the value of the right argument is returned. There is also a deep analogue of ($!), called ($!!).

Although seq can be used immediately on all datatypes, deepseq has to be told how it should "descend down" the values, forcing computations at every level. You do that by making a datatype an instance of the NFData type class, by defining the function rnf. Fortunately, many instances of NFData are already defined in *Control*.DeepSeq, including those for lists, arrays, tuples, and the like.

One other way to employ deepseq is to write a foldl″ variant that applies deepseq instead of seq. A problem to achieve big gains over the provided foldl and foldl′ is that the latter were written with all kinds of compiler optimizations present in GHC in mind. But normally, if you want to fold over a list to compute some non-primitive type, then deepseq could very well help you. This is how you could write such a foldl″ (thanks, Jeroen Bransen):

```
foldld f e xs = foldl″ e xs
where
  foldl″ e [] = e
  foldl″ e (x : xs) = let acc = f e x
    in acc `deepseq` foldl″ acc xs
```

The only difference with a hand-written version of foldl′ is the use of deepseq over seq. And indeed, this foldld will typically beat a hand-written foldl′.

To conclude: although, we believe it is important for you to understand about strictness and laziness, deepseq, seq and the like, it is also important to realize that you should only optimize your program when you find that performance is a problem. Often you will find that making your program stricter is not going to help, because the compiler will typically perform a strictness analysis for you and use that to optimise your code. But it is also true that a compiler may not be able to spot all such opportunities and you may need to help it. Although we do not discuss this in this reader, the Real World Haskell book has a chapter that explains how to find out where the performance bottlenecks in your Haskell application are, and what to do about them. Also remember that a badly placed strictness annotation can also lead to worse performance, sometimes even non-termination.

Exercises

15.1 In an older version of the Prelude the function intersperse, which places an element between all elements of a list, was defined as:

```
intersperse e []           = []
intersperse e [x]          = [x]
intersperse e (x : y : xs) = x : e : intersperse e (y : ys)
```

What would you expect the result of the expression `intersperse 'a' ('b':undefined)` to be? Can you give a definition of intersperse that is less strict?

15.2 Given the data type

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

we define the function tja:

```
tja t = let tja' (Leaf a)   n ls = (0, if n == 0 then a : ls else ls)
            tja' (Node l r) n ls = let (lm, ll)  = tja' l (n - 1) rl
                                       (rm, rl) = tja' r (n - 1) ls
                                   in ((lm `min` rm) + 1, ll)
            (m, r) = tja' t m []
        in r
```

If this code computes something explain what it computes (small example?); if it does not compute anything explain why this is the case.

16 Combinator libraries and EDSLs

16.1 Introduction

Over the last few years, many computer scientists and professionals have shifted their attention from general purpose languages (GPL) to *domain specific languages (DSLs)*, relatively small languages designed for solving problems in a particular domain [7, 5, 4]. They have realized it can be much more productive than a general purpose language. Moreover, a DSL is often easier to grasp by non-professionals and therefore can expect to have more wide-spread use. A well-known, early example is the SQL language for database querying, another is the language for writing Excel formulas. In the O.O. world programming in a domain-specific language is very similar to what they call model driven engineering: from a typically visual representation the code of the application is generated. Maintenance of the application can then take place on the model (which is more intuitive and higher-level), followed by pressing a button to regenerate the code, instead of maintaining the source code. In the context of this course, we consider textual domain-specific languages, as they are embedded in Haskell.

According to Walid Taha [9], a DSL has four essential characteristics: the domain is well-defined and central, the notation is clear, the informal meaning is clear, and the formal meaning is clear and implemented. It is the latter characteristic that sets a DSL aside from a jargon [9]. To these characteristics, I personally like to add:

- an implementation of the DSL can communicate with the programmer about the program in *terms of the domain*.

The rationale is that programmers make mistakes, and communicating the diagnosis of a mistake in terms other than that of the domain completely defeats the purpose of working with a DSL!

Although providing domain level feedback can demand a substantial engineering effort, the problem is not particularly difficult if for every DSL we essentially implement a new compiler. But this is not always the best approach. In [6], Paul Hudak argues that DSLs are the "ultimate abstraction", and introduces the idea of *embedded* DSLs (EDSLs) (Fowler calls these *internal* DSLs, non-embedded DSLs are called *external* [3]). EDSLs typically inherit the style, syntax, type system, infrastructure, and tooling of a chosen host language. There are significant advantages including:

- The complete infrastructure of the host language can be reused, e.g., libraries, code generation, debuggers, implementation of floating point numbers, that are costly

to implement and maintain for separate DSLs.

- EDSLs (for the same host language) can typically be combined relatively easily. Combining multiple external DSLs, each with their own tool chain, is a daunting task by itself and at worst for every useful combination a separate tool chain must be maintained.

Arguably, a further benefit is that when the EDSL is not expressive enough to solve a particular problem, the programmer can always fall back on the host language, which is usually a general purpose programming language. Moreover, the host language provides a form of basic syntax, leading to EDSLs that are similarly styled and therefore may be easier to learn. On the other hand, how likely is it that the style of programming of the host language is the same or similar to that of the domain? An essential aspect of a domain-specific language is that programs written in it should exhibit a certain amount of *fluency*: the syntax that is employed feels like the syntax that may have been in use for decades or more in that domain.[1]

EDSLs also have two important disadvantages: since the EDSL is embedded inside (encoded into) some general purpose language, the compiler for the latter has no awareness of concepts in the EDSL, which leads to the inabilities

- to report type error messages in terms of the domain (domain specific error diagnosis),
- to exploit knowledge of the domain to generate better code (domain specific optimisations).

Research is currently being undertaken (also at this university) on solving these two problems.

16.2 EDLSs and combinators

To be able to embed languages well into a host language, certain language features turn out to be very useful. For Haskell, type classes, higher-order functions and parametric polymorphism provide much of what is necessary (GADTs are another such feature, but we do not consider GADTs in this course).

On the other hand, it is striking how creative library designers can be in embedding languages inside a given host language. For example, fluent interfaces were "invented" to provide a generic way of embedding domain-specific code while following the syntax implied by the O.O. paradigm (as Wikipedia explains the idea of fluent interfaces

[1] Ever notice that modern programming languages do their utmost to ensure that arithmetic expressions can be written in the way that you learned to in primary and secondary school? This is no accident. Since arithmetic is so generally useful, it is typically (but not always) built into the language to ensure its original fluency. But we cannot expect host language designers to do this for every possible domain.

actually goes back to Smalltalk in the 70s). The idea is to have methods implement domain-aspects, and to chain various operations together by means of *method-chaining*. To give you an idea, the following example for "SQL embedded in Java", was taken from Wikipedia:

```
Author a = AUTHOR.as("a");
create.selectFrom(a)
      .where(exists(selectOne()
                    .from(BOOK)
                    .where(BOOK.STATUS.eq(BOOK_STATUS.SOLD_OUT))
                    .and(BOOK.AUTHOR_ID.eq(a.ID))));
```

Even if you do not understood much of SQL and Java, it may seem intuitively clear what is expressed by this query.

It should be noted that although Haskell has sufficient flexibility for implementing the semantics (meaning) of EDSLs, it is much harder to achieve a syntactic likeness of an EDSL if the EDLS happens to have a syntax much different from Haskell. Some general purpose languages have in fact specialized in such extensibility. Scheme and its relative Racket, for example, are functional languages that allow for syntactic extensiblity by means of syntax macro's (but you are stuck with the parentheses), and the dynamically typed language Ruby was devised in the way that caters for the easy embedding of DSLs. However, all these languages are dynamically typed: syntactic extensibility is easier to implement if you do not have to worry about type correctness.

In a sense, Haskell is then a middle road: since programmers can define new operators, and overload existing ones, a certain amount of flexibility is provided, and, of course, Haskell remains statically typed.

The higher-orderness of Haskell allows the definition of what we call *combinator libraries*. To illustrate, let's consider a particular domain, that of 2D diagrams (boxes, circles, etc.). We then expect an EDLS to at the very least provide:

- Data types that represent boxes, circles etc., for example $\mathsf{Circle}\ (\mathsf{x}, \mathsf{y})\ \mathsf{r}$, in which (x, y) is a 2D point, and r is the radius.
- Operations that output graphical renderings of values of such type.

But how do you then actually build such diagrams? Primitive diagrams such as circles and boxes tend to be defined by means of primitives. But how can we compose large and complicated diagrams, and reuse parts of one diagram in another, much as we are used to when we construct software?

Consider the analogy of assembling a car: a car is to the consumer maybe a single object, but it is assembled by putting various components (the wheels, the motor, the chassis, etc) together in some way. In other words there is a domain-specific program at work here that given a number of primitives (e.g., products delivered to the car company like

wheels, screws, paint and glue), that tells us how to put the components together. Such a specification is typically unique for the car, or maybe you have a single specification that abstracts away from the particular wheels.

To actually construct the car, we also need operations for assembling components together, like gluing and welding. Such operators are called *combinators* in the world of programming languages. They represent operations that take values (car components) of your domain, to construct new, larger (car components) in that domain, and eventually the car as a whole. And each such component may itself be constructed in exactly such a fashion, until we finally arrive at the primitive components.

In the examples that follow we shall see many examples of this phenomenon. In the area of diagram construction typical operations that you can expect to see are:

- given two diagrams, put them side by side,
- scale, rotate or translate a diagram,

Essential here is that we have operators that take a domain value as an argument (or more than one), to construct other elements of the domain although there may be arguments of other kinds as well. For example, to scale a diagram we typically alos pass in the scale factor. If elements of the domain are best represented as functions, it is essential that the host language supports higher-order functions to give types to these combinators. Although some of these combinators are defined as part of the EDSL, there is nothing that prevents you from defining your own. For example, if the library supports scaling and translation, but not reflection (mirroring), you can define your own reflection by a combination of translations and scaling by a factor of -1. [2]

16.2.1 Shallow and deep embedding

When you embed an EDSL in a language, you have to choose whether you want this embedding to be *shallow* or *deep*. What does that mean? In both cases, the Haskell code that you write (in that domain) is plain Haskell, but in a *deep* embedding, the domain-level expression serves to build an abstract syntax tree (AST) of the domain program, and that tree is a Haskell value. An advantage of such a representation is that

- you can write Haskell functions to analyze, validate and optimize such domain programs (because the AST ultimately represents such a domain program)
- you can have various different semantics for the domain program. For example, for our 2D diagrams, one "semantics" might be to spit out a bitmap with a rendering of the picture, another might be the same picture but then in SVG (Scalable

[2] A popular example is the domain of context free languages, represented in Haskell by parser combinators. The basic combinators are those defined in (Extended) Backus Naur Form for defining context-free grammars, in terms of which other useful combinators can be defined (see the course on Languages and Compilers for more details).

Vector Graphics) format, and another might the (optimized and analyzed) domain program in textual form.

In a *shallow* embedding, the domain-level expression that directly represents the intended semantics in Haskell. The only thing we can do with such an expression, is to evaluate it using standard Haskell semantics. This provides less flexibility, but may suffice for certain simpler domains.

16.3 Example 1: Chris Done's formatting package

The language C introduced a procedure called printf for printing formatted text. Consider the following example:

```
printf ("His name was %s, and he was aged %d", "John", 69);
```

The output is the string in which `John` is interpolated in the position of `%s`, and `69` replaces `%d`. The letter following the percentage sign is an indicator what type of value should be interpolated. For example, the `d` stands for signed decimal, and `s` for string. Since C is a rather lax language when it comes to types, there is no guarantee that what you write matches what the formatting string demands. Can we do better in Haskell? Maybe.

One option is to use *Text*.Printf. It allows you to write

```
Prelude> :m Text.Printf
Prelude Text.Printf> printf "%d plus %d makes %s\n" 2 3 "five"
2 plus 3 makes five
```

But unfortunately, types are checked at run-time, not compile time.

```
Prelude Text.Printf> printf "%d plus %d makes %s\n" 2 3
2 plus 3 makes *** Exception: Printf.printf: argument list ended prematurely
Prelude Text.Printf> printf "%d plus %d makes %s\n" 2 3 5
2 plus 3 makes *** Exception: Printf.printf: bad argument
```

Again, the question arises: can we do better? Chris Done constructed a library called formatting (to be found on Hackage), in which these errors become type errors (not very nice ones, but that is not his fault) [3]. The library is set-up as a small combinator library, and can be understood as a small EDSL for writing C-style formatting code. Another advantage of the combinator library is that it can be extended with programmer-defined

[3] Much of the material in this section is taken from Chris' blog: http://chrisdone.com/posts/formatting

formatters for whatever you fancy. Formatters for such things as time, dates and the like are already part of the library. [4]

In this chapter, we shall be looking at the facilities the library provides, and how it can be used. We try to avoid discussing how it was implemented (although you are free to go and have a look).

A central function is format, which we illustrate by some examples: essentially it builds up a structure with holes in it (as many as specified), and after you have plugged the holes, returns a *Text* (which is simply a more efficient version of a *String*). To say hello to the world, we can write:

```
hello = format (now (fromString "Hello, World!"))
```

and executing hello in the terminal then gives the very boring

```
FormatEx> hello
"Hello, World!"
```

Before we go on, consider this: the Formatting package only works on *Text*s and not on *String*s, but literals like `"Hello, World!"` are *String*s not *Text*s. Clearly, we often want to use literal strings in our formattings, and since programmers have found writing all those explicit fromStrings cumbersome, a compiler flag was added to overload strings. Essentially, it allows the compiler to "correct" type errors by adding a conversion function fromString to literals, and in our examples, these will silently convert *String*s to *Text*s. This is not *too* dangerous since a *Text* is simply a more efficient representation of the same concept.

So if we include `{-# LANGUAGE OverloadedStrings #-}` in the source file we can write

```
hello = format (now "Hello, World!")
```

The conversion from the literal string to the *Text* value that now expects is performed for us. Now, the now function converts a *Text* to some data type that format understands (the type Holey ... but let us not get into that). The point is that there is also a fromString to directly convert *String*s to that type, so with `OverloadedStrings` we can even write

```
hello = format "Hello, World!"
```

Is more concise better? Since the EDSL is meant to state whether you have the value "now", or "later", omitting the now may make the program less clear. So yes, we certainly prefer not to write the fromString to transform a *String* to a *Text*, but do prefer to keep the now. As a side note, if you want to be fully explicit in your use of fromString since

[4] The library is inspired by the HoleyMonoid library developed by Martijn van Steenbergen, a former master student at UU. With it you can describe expressions with holes in them, which can be filled at a later time. It should not come as a big surprise that the holes represent the formatting directives, like `%d`.

you do not want your code to depend on something as arbitrary as a compiler flag, you may find that writing fromString in your code leads to another problem: it may not be able to find fromString. This is because if you use fromString explicitly, you also need to import *Data.String* explicitly. If you use the compiler flag, it also performs the import for you.

To summarise, if you turn on `OverloadedStrings` and import *Data.String*, then the following definitions are all equivalent:

```
hello   = format (now "Hello, World!")
hello'  = format (now (fromString "Hello, World!"))
hello'' = format (fromString "Hello, World!")
hello''' = format "Hello, World!"
```

There is one snag. What about the following:

```
helloS :: String -> Text
helloS s = format s
```

This will not compile: fromString is only silently applied to *literals*, not to any expression of type *String* (which is good, because otherwise we lose a lot of the strong typing guarantees which Haskell provides). In other words, now you will have to import *Data.String* and write

```
helloS :: String -> Text
helloS s = format (fromString s)
```

and

```
FormatEx> helloS "Hello, World!"
"Hello, World!"
```

Now that we have that covered, let us continue with the formatting package proper. The idea of formatting is that some part of what we want to print/format is known up front, but some things are passed in later. These things generate "holes" that can be plugged at a later time, and when all holes are plugged we can in fact display the result. For that we use the later function. Its use is pretty intuitive:

```
flexible2 :: (Show a, Show b) => a -> b -> Text
flexible2 x y = format (now "Value "
                        % later (fromString . show)
                        % now " is "
                        % later (fromString . show)
                        % now ".")
                       x y   -- Pass in the first and second thing to show
```

As you can see, the values we have already (the literals) are passed to now, and we leave two holes as indicated by later, one hole of type a and one of type b. Note that flexible2 has a straightforward Haskell type: nothing intermediate is being constructed here. This implies that formatting is a shallowly embedded DSL. These holes are also plugged here by passing in the arguments x and y. The function works for all showable types:

```
*FormatEx> flexible2 45 "my age"
"Value 45 is \"my age\"."
*FormatEx> flexible2 45 [45]
"Value 45 is [45]."
*FormatEx> flexible2 id "identity function"
...
   No instance for (Show (a0 -> a0)) arising from a use of ‘flexible2’
    Possible fix: add an instance declaration for (Show (a0 -> a0))
    In the expression: flexible2 id "identity function"
    In an equation for ‘it’: it = flexible2 id "identity function"
```

In the latter case, there is no *Show* instance for functions, so the expression is type incorrect.

Note that another important combinator was introduced above: we often want to sequence various parts of the message, and we just saw how the % operator can be used for the purpose. Since the effect of using later is to introduce a new hole (represented as the argument to a function), this sequencing operator has much in common with function composition[5].

The type of flexible2 depends on the show function to visualize information. This has some disadvantages: there is no single best way to present, e.g., an integer or float. In particular, for the case of floating point numbers, we sometimes want to present 1 / 3 as 0.33 and sometimes as 0.333333333, depending on the situation. Also, we sometimes want to present integers in either decimal, binary, octal or hexadecimal notation. For this reason various additional formatters are available in a module *Data.Text*.Format. What the Formatting package adds to that is the possibility of building expressions with holes in which we already say how the hole should be formatted once the information becomes available (passed as an argument), but not what the values are.

A disadvantage of our formulation of flexible2 is that we can pass anything into flexible2 that has a *Show* instance. Sometimes we want, or have to be, more precise. Looking back to the start of this section, we wanted ensure that in

```
printf "%d plus %d makes %d\n" 2 3 "five"
```

the types of the eventual “plugs” match with those specified by the hole, in a way that promises to catch these inconsistencies at compile-time, not at run-time. The formatting

[5] You can in fact see that from its type (%) :: Monoid n => Holey n b c -> Holey n b1 b -> Holey n b1 c, if you squint to omit the n, and references to Holey.

functions themselves are responsible for this. For example, a formatter that correctly detects the mistake in the above code would be

```
*FormatEx> let sumit = format (int  % now " plus " % int
                                    % now " makes " % int % now "  n")
*FormatEx> sumit 2 3 (2+3)
"2 plus 3 makes 5\n"
*FormatEx> sumit 2 3 "five"
  ...
    Couldn't match expected type `Integer' with actual type `[Char]'
    In the third argument of `sumit', namely `"five"'
    In the expression: sumit 2 3 "five"
    In an equation for `it': it = sumit 2 3 "five"
```

If we consider the "domain" to be of "formatting directives", then in EDSL terms the later and now functions are the primitives, and (%) and the associated but undiscussed (.%) may be considered the combinators. The latter two combine multiple "expressions with holes" into a new "expression with holes".

For the rest, the formatting package consists of functions that create a hole with some formatting function governing how the information in the hole will eventually be formatted. Examples of this kind can be found in Formatting.Formatters and includes functions like hex (that formats an integer in hexadecimal notation), and fixed i (that displays real numbers with a certain, fixed amount of precision).

To give you an idea:

```
*FormatEx> :set -XOverloadedStrings
*FormatEx> format (fixed 2 % now ", " % fixed 4) (1/3) (1/3)
"0.33, 0.3333"
```

The first line is to make sure the literal string `", "` typed into `ghci` is also liberally interpreted as, in this case, *Text*s when needed. Otherwise, we'd have to write

```
*FormatEx> format (fixed 2 % now (fromString ", ") % fixed 4) (1/3) (1/3)
"0.33, 0.3333"
```

It should be clear by now that we can format all kinds of data types and in many different ways. For example, we can format the temperature in Celsius as follows, which is essentially a floating point number with one decimal behind the period, and the special ° sign:

```
celsius2Text :: Float -> Text
celsius2Text = format (fixed 1 % now "\x00b0" % now "C")
```

Although the function can format text, the type also shows that it is itself not a formatter, i.e., of type Format a. In order to add the formatting to our repertoire of formatting directives, we need a function celsius, so that it can be used in the following fashion:

```
warmwhen :: Real a => a -> Text -> Text
warmwhen = format (now "It was " % celsius % now " on " % text)
```

This celsius formatter can be written as follows:

```
celsius :: Real a => Format a
celsius = fixed 1 % now "\x00b0" % now "C"
```

which works for any type of real number. As you can see, the only difference is to not apply the format function, since that is the function that takes a formatter and turns it into a Haskell function that maps some inputs to a *Text*. Once we apply format, we leave the domain of formatter specifications.

```
*FormatEx> warmwhen 22.9 "Oct 8"
"It was 22.9\176C on Oct 8"
```

If you apply putStrLn to this string you get `It was 22.9°C on Oct 8`.

After this simple form of extensibility, let us look at a more generic example.

```
maybe :: b -> (a -> b) -> Maybe a -> b   -- as a reminder

mint = later (maybe "Don't give me Nothing" (bprint int))
mint = later (maybe "Don't give me Nothing" (format int))
```

The type of mint is Holey Builder r (*Maybe Integer* -> r) suggesting that it needs a *Maybe Integer* to be able to return a result. This is not so strange, since we have already supplied the first and second argument. So we can write

```
*FormatEx> format mint (Just 23)
"23"
*FormatEx> format mint Nothing
"Don't give me Nothing"
```

There is no reason to stick to *Maybe Integer* in this case, if we then pass in the default to print for *Nothing* and the formatter for the contents of the *Just* as arguments:

```
mfmt x f = later (maybe x (bprint f))
```

and then write

```
*FormatEx> (format (mfmt "Crash!" int)) (Just 23)
"23"
*FormatEx> (format (mfmt "Crash!" int)) Nothing
"Crash!"
```

Without going further into details, the type error messages that you get when you pass in something of a non-matching type, or when you forget to pass something typically demand that you add some type class instance you never had any intention of writing. This is pretty typical of type incorrect code in the presence of type classes. If you pass too many arguments, the error messages tend to reveal something about the underlying implementation, in this case the use of Builders and HoleyT datatype.

16.4 Example 2: Brent Yorgey's diagrams package

A much more extensive EDSL is Brent Yorgey's *diagrams* package. It can be used for constructing pictures in a compositional way, and offers many combinators for constructing complex pictures out of simpler ones. The place to start is `http://projects.haskell.org/diagrams/` which features an on-line version to try it out, and also has many examples. We shall restrict ourselves to the mature 2D picture facilities of *diagrams*. In particular, we shall largely be following the introductory manual located at `http://projects.haskell.org/diagrams/doc/manual.html`.

The motivations for Yorgey to devise *diagrams* in the first place was to have a specification of diagrams that is declarative (only what you want, not how to draw it), compositional (building complex diagrams out of simpler ones), embedded (in Haskell, so that Haskell's powerful features including its type system can help you build your diagrams), extensible (with new combinators for building diagrams in your own way more effectively), and flexible (supporting not only 2D, but, based on similar mechanisms, also 3D pictures and animations).

Some of these demands, particularly the type safety and the flexibility have led to what is at first glance a pretty complicated library: in its construction it uses type classes a lot, and some of these can seem pretty abstract.

The examples we give below are often taken from the *diagrams* website, sometimes of our own making. The apparent vagueness of some of these pictures is due to our way of displaying them here, and are not due to the library. To be able to run them you will typically put them in some module (called DiaEx here) that you then start off with the following preamble:

```
{-# LANGUAGE NoMonomorphismRestriction #-}
module DiaEx where

import Diagrams.Backend.SVG
import Diagrams.Backend.SVG.CmdLine
import Diagrams.Prelude
import Data.Maybe
import Data.Tree
import Diagrams.TwoD.Layout.Tree
```

Not all examples below need all of it, but it does not hurt to import it all during experimentation. If you load DiaEx into `ghci` (having first installed *diagrams* from Hackage as indicated on the website), you can get a pretty good idea how extensive the library really is by typing `Diagrams.` in the interpreter and pressing the Tab key:

```
*DiaEx> Diagrams.
Display all 1201 possibilities? (y or n)
```

The *diagrams* package uses plenty of types throughout, like Diagram and Colour, but also type classes such as Color and Monoid. Inside `ghci`, you can quickly get more information by typing

```
*DiaEx> :info Color
class Color c where
  toAlphaColour :: c -> AlphaColour Double
  fromAlphaColour :: AlphaColour Double -> c
          -- Defined in ‘Diagrams.Attributes’
instance Color SomeColor -- Defined in ‘Diagrams.Attributes’
instance (Floating a, Real a) => Color (Colour a)
  -- Defined in ‘Diagrams.Attributes’
instance (Floating a, Real a) => Color (AlphaColour a)
  -- Defined in ‘Diagrams.Attributes’
```

giving you the type class members, the instances of the type class, and the locations of all of these.

An easy way to use *diagrams* is to write code into the module, and use the function main to draw the pictures of interest. In this case, the list of pictures to draw (we'll see their code later), are displayed vertically, with some space inserted in between (that's the vspace). As you can see, the list of examples contains four elements, the first of which is in fact itself composed of multiple diagrams.

```
main = mainWith examples

vspace :: CatOpts R2
vspace = with & sep .~ 0.2

examples :: Diagram B R2
examples = vcat' vspace
  [circleAndTheSquare # center
    ===
   scaledCosCircles # center
  , nestedCirc
  , scale 0.25 (sierpinski 6)
  , kierpinski 4
  ]
```

Then inside `ghci` you do

```
*DiaEx> :main -o DiaEx.svg -w 500
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package bytestring-0.10.0.2 ... linking ... done.
Loading package zlib-0.5.4.1 ... linking ... done.
.......
.......
```

and afterwards there will be a file DiaEx.svg that you can visualize by loading it into, say, the Firefox browser. Everytime you change the code, you should do a :r, rerun the :main, and reload the page in Firefox.

16.4.1 The basics (but are they?)

Maybe typically for EDSLs, the basics of the library are not the basis of the library: what is used by the EDSL author as the generic building blocks of the library tends to be hidden behind lots of syntactic sugar.

This is a good thing: it is much nicer if someone can draw a few pictures without first having to learn everything there is to know about all the type classes that are being used throughout the library for what may at first be obscure reasons. Of course, at some point, you will be wanting to know more to get more out of the library. Or as it happens, your code will contain a type error and then you will be confronted by what lies underneath. And maybe to make sense of what you did wrong, you will need to know a bit more about how the library is organised. But for now let's just try a few things and not make any mistakes.

The code example in Figure 16.2 illustrates a number of often used primitives and

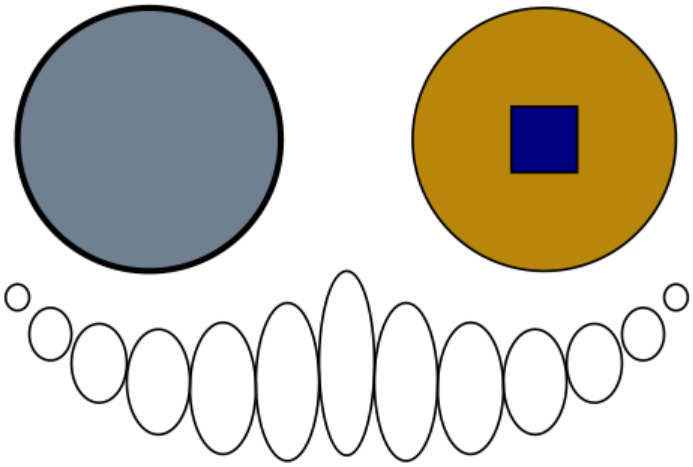


Fig. 16.1: An ugly face (with monocle)

combinators. The identifier uglyFace renders to the picture in Fig. 16.1.

The identifier uglyFace represents the complete diagram, and is itself composed of two other diagrams, circleAndTheSquare and scaledCosCircles. The first draws the upper half, the second the lower half. The two are positioned one above the other by means of the (===) combinator. Note how the notation === between the two diagrams also suggests in the code how they will be layed out. This is no accident! It becomes much easier to understand what the diagram will look like in this way, making the code more intuitive and self documenting. This is something that EDSL designers will tend to strive for.

Note that each of these identifiers has type Diagram B R2. What does that mean? Well, that each is a diagram (duh), but also that the diagrams are two-dimensional (hence the R2), and B is an alias for the backend that is used for rendering. Except for a single 3D example later on, all examples use 2D graphics, and SVG (Scalable Vector Graphics) as the backend. The latter comes from our choice to import Diagrams.Backend.SVG.

Let us continue our discussion of the ugly face code with circleAndTheSquare. This diagram consists of two circles, spaced apart by some distance. To put the two circles side by side we use the (|||) combinator. But what is that strut unitX in the middle? Without it, the two circles would be put side by side with no space in between. The strut primitive ensures that the circles are spaced unitX apart. Essentially, a strut v is a diagram that produces no output but does take up space, as governed by v. The first circle, theCircle is based on the circle drawing primitive; you pass it a size, in this case 1 for a unit circle. The diagram theSquare illustrates yet another important combinator: atop. It takes two diagrams and places the first over the second. It is typically used in infix style, as we do here.

Every diagram has certain attributes, e.g., background color, foreground color, line

```
uglyFace :: Diagram B R2
uglyFace = circleAndTheSquare # center
             ===
           scaledCosCircles # center

circleAndTheSquare :: Diagram B R2
circleAndTheSquare = theCircle ||| strut unitX ||| theSquare
  where
    theCircle = circle 1 # lw veryThick # fc slategray
    theSquare = square 0.5 # fc navy `atop` circle 1 # fc darkgoldenrod

scaledCosCircles :: Diagram B R2
scaledCosCircles =
  foldr c mempty ([0.1, 0.2 .. 0.6] ++ [0.7, 0.6 .. 0.1])
    where
      c rad res = circle rad # scaleX (1 − rad)
                             # translateY (0 − sin (pi ∗ rad)) ||| res
```

Fig. 16.2: Ugly face code

width, and origin. These can all be changed with the (#) operator. Consider

```
circle 1 # lw veryThick # fc slategray
```

This is a circle of unit size, followed by an expression that modifies its line width to veryThick and its foreground color to slategray. The identifiers veryThick and slategray are, in addition to many other values for line width and colours, defined by the *diagrams* library.

Omitting a somewhat scary detail, the type of fc is

```
fc :: (HasStyle a, ...) => Colour Double -> a -> a
```

The HasStyle type class ensures that fc can be applied to anything that "has a style" to modify, and not to anything that does not have it. It then expects a Colour Double to indicate the colour to change to and applies it to the second argument (which is the diagram). So why do we not pass the diagram in then? That is because of (#) :: a -> (a -> b) -> b, which as the type suggests is essentially reverse application, takes care of that:

```
circle 1 # lw veryThick
```

is in fact the same as

```
lw veryThick (circle 1)
```

The EDSL designer purposefully defined it in this way: when you construct diagrams you typically start by defining the circle, and then consider changing some of its attributes. So it makes sense to have the setting of attributes follow the construction of the diagram. To the diagram designer the notion of lw indicating an attribute or property will also typically be more notationally intuitive than that of lw as a function, but note that since lw *is* a function we can map it over a list of diagrams if we want to. For example,

```
map (fc slategray) [circle 1, circle 2, circle 3]
```

Note that the priority of (#) has purposefully been chosen as very high so that you typically do not have to write many parentheses to indicate which properties belong to which diagram. For example, in

```
square 0.5 # fc navy `atop` circle 1 # fc darkgoldenrod
```

the first fc can only apply to the square, but the final fc could apply to either the circle 1 or to the superposition of the square on top of the circle. Since (#) binds very strongly, stronger than atop (and the other combinators), it modifies the properties of circle 1 only. Essentially, the way it has now been set up, we get a syntactically lighter version of the fluent interfaces in Java for setting properties.

There are many other attributes besides color and line width that can be changed. Consider for example the center property that is used (twice) in uglyFace. Its effect is to set the origin of a diagram to its center. In Fig. 16.3 you can see the result of omitting the center properties (ignoring the red dots for the moment). So what is happening here? For the library to decide know how to put diagrams together, every diagram has a chosen point of reference, the origin. And each combinator has certain rules that it applies how the new origin is computed from the origins of the constituents. Since it is not very easy to remember all those, you can make the origin explicit in the visualization, by applying the showOrigin property which is visually represented by a red dot.

Now, what (===) does, is to line up its two diagram arguments in a way that the origin of the first argument lies directly above the other. If that gives a diagram that is badly aligned, then we just have to explicitly change the origins of the arguments. In the example, we used center to do exactly that. Beside center there are many variants like alignT for moving the origin to the top, and alignBL for moving it to the bottom left corner.

Finally, we turn to scaledCosCircles (note that we have purposefully implemented this function in a somewhat verbose style; other, more elegant solutions exist), in which we employ linear transformations and translation to achieve an effect (other well known transformations such as rotation, shearing, and reflection are also supported).

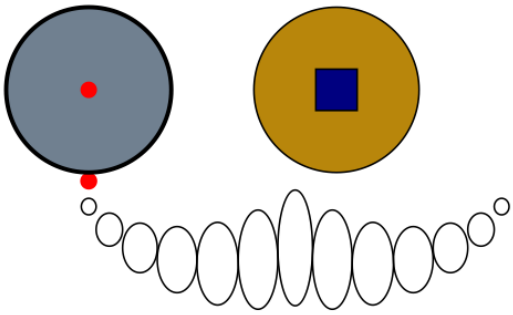


Fig. 16.3: More ugliness

```
scaledCosCircles :: Diagram B R2
scaledCosCircles =
  foldr c mempty ([0.1, 0.2 .. 0.6] ++ [0.7, 0.6 .. 0.1])
    where
      c rad res = circle rad # scaleX (1 − rad)
                            # translateY (0 − sin (pi ∗ rad)) ||| res
```

In the definition we exploit the possibility to combine "ordinary Haskell" with *diagrams*, something that many consider a benefit of embedding a DSL inside a general purpose language. In this case, we build a compound picture by putting a number of transformed ellipses side by side (using the binary (|||)) (there are easier ways for doing this, but please bear with us) using foldr.

The foldr generates an ellipse for each of the radii in the argument list. It also passes mempty which represents an empty diagram, and is a zero element for both (|||) and (===). Each ellipse is constructed by the helper function c, which takes a radius, and the result of putting all the following ellipses side by side. We then construct the new ellipse based on its radius, and put it side by side with the others. The scaleX attribute scales the circle (of radius rad) by an amount inversely proportional to its radius, so the larger the circle is, the more ellipse like it becomes. We also translate the origin of the ellipse by an amount that follows a sine wave. Since it depends only on the radius of the original circle, we get two (partial) sine waves.

Some more diagrams

Since we have recursion at our disposal, many examples on the *diagrams* webpage use it for visual effects. Here we reproduce the code to produce a shaken up version of the Sierpinski casket, as rendered in Fig 16.4:

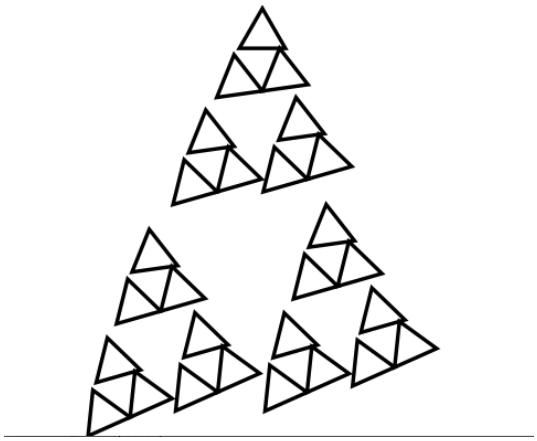


Fig. 16.4: "Kierpinksi" casket of order 4

```
kierpinski 1 = eqTriangle 1
kierpinski n = s
                  ===
               centerX (s ||| s)
                       # rotate (8 @@ deg) # centerX
  where s = kierpinski (n − 1)
```

An actual Sierpinski casket of order 6 (for reference) is given in Fig. 16.5. To obtain the code to draw the Sierpinski casket, all you have to do is delete the line #rotate (8 @@ deg) # centerX from the kierpinski function.

The rotate (8 @@ deg) is responsible for the somewhat shaky looks of the "Kierpinsky" casket. The basis of a Sierpinsky casket consists of putting two triangles side by side, and a triangle on top. You get a Sierpinski casket of order n by applying this to Sierpinski casket to order n − 1.

To draw a bullseye you can hand code it as follows (see Fig. 16.6):

```
nestedCirc :: Diagram B R2
nestedCirc = nest 1 blue red
  where
    nest s col1 col2 =
        if s < delta then this
        else nest (s − delta) col2 col1 `atop` this where
          this = circle s # lw veryThin # fc col1
    delta = 0.1
```

This pattern can also be expressed by using mconcat (which is essentially atop extended to lists), and using zipWith in combination with cycle to cycle through the colours (see

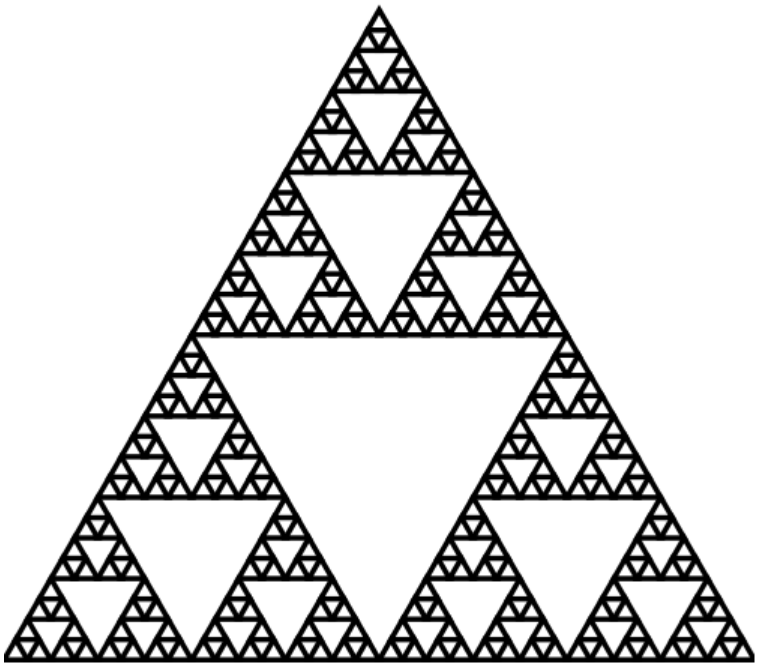


Fig. 16.5: Sierpinksi casket of order 6

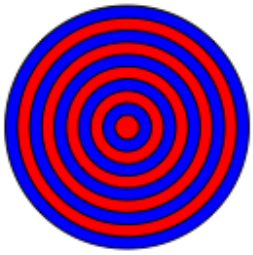


Fig. 16.6: A bullseye handcrafted

Fig. 16.7: A bullseye with mconcat

Fig. 16.7):

```
bullseye :: Diagram B R2
bullseye = mconcat $ zipWith (\ s c -> circle s # fc c # lw veryThin)
                             [0.1, 0.2 .. 1.0] (cycle [red, white])
```

16.4.2 Somewhat deeper

An important aspect of the library is to use type classes to impose restrictions on how certain values can be employed. Let us now look at some of the type classes that play an important role within *diagrams*. Consider the function vcat. Its type is

```
vcat :: (Juxtaposable a, HasOrigin a, Monoid' a, V a ~R2) => [a] -> a
```

So what does that mean? The qualified type demands that if we apply vcat to a list of type $[a]$, then the element type had better have a number of properties: it has to be Juxtaposable (i.e., be something that you can set side by side (in any direction)), HasOrigin unsurprisingly guarantees vcat that the argument in fact has an origin (so that vcat can use it to align the elements of the argument list based on their respective origins), and finally, a has to be an instance of Monoid′. Before we go on to explain a bit more about the Monoid′ we remark, without wanting to go into any details, that V a ~R2 implies that we are dealing with a 2D diagram.

So what about the Monoid′ restriction. In mathematics, a set A and binary operation $\oplus$ of type A -> A -> A form a *monoid* if $\oplus$ is associative, so $x \oplus (y \oplus z)$ equals $(x \oplus y) \oplus z$ for all x, y and z in A. In other words: parentheses can be omitted. In addition, A must contain a (unique) identity element 0, so that $0 \oplus x = x = x \oplus 0$ for all $x \in A$. The type class Monoid has two basic members: mempty to represent the identity element, and mappend to represent the binary operator (there is also mconcat which generalizes mappend to a list of arguments). The reason why Monoid′ is used, and not Monoid is because Yorgey wants the structure to also be a semigroup (there mappend is called <>). To a mathematician this would be strange, since a semigroup is a monoid without the

demand for an identity element. But in the Haskell libraries, Semigroup was not defined as a superclass of Monoid; this may (and maybe should) change in the future, and then Monoid suffices.

So what is achieved by turning diagrams into a monoid (a similar reasoning applies to transformations, envelopes, traces, trails, paths, styles, and colors)? The intuition is simple: when you combine multiple diagrams into one diagram you do not want the result to depend on the order in which the diagrams are put together. Additionally, the ability to have an explicit "empty diagram" is also typically useful, when for example you want to visualize a list of diagrams and need something for the case of the empty list.

But there is more: a diagram Diagram b v is a synonym for QDiagram b v Any. The Q in QDiagram stands for query. A diagram is a query in the sense that it defines where the diagram is defined: in the case of QDiagram b v Any, this means that the associated query returns *False* for points "outside" the diagram and *True* for points "inside". Combining two diagrams then becomes a matter of taking the logical *or* of the two diagrams, and *False* is its identity. This is exactly the behavior of the Any monoid. The use of mconcat is an example where we used this to put a number of circles on top of each other (see Fig.16.7). Clearly, Diagram is a particularly important example of a QDiagram, but sometimes it is useful to use another monoid for combining diagrams. The documentation of *diagrams* gives a few examples, but we shall not further pursue that here. The use of the Monoid type class in this case, is part of a general recipe: to reuse existing type classes in order to exploit both their behavior, but also their notation. For example, if you use monads in a DSL, then you also inherit the popular do-notation. In fact, it often happens that some type that does not follow the monad laws is implemented in terms of monads, simply to be able to use the associated notation for monads.

Another typical aspect of embedded DSLs is that there is never any shortage of functions, simply because they are so easy to add. In the main program we used a function called vcat′ that like vcat arranges diagrams vertically. But vcat′ is a bit more general in that you can also pass in an argument that governs how much spacing is inserted in between the diagrams. And as you may expect: vcat is implemented by means of a call to vcat′. But that is not where it ends, since vcat′ is again implemented in terms of cat′, by providing a fixed direction to stringing the diagrams together. This is a phenomenon typical for EDSLs: since it is easily possible to define new useful functions and combinators in terms of others, many such will exist. This has disadvantages too: when you start out to use an EDSL the number of functions may seem daunting, and it takes a while to find out what are the ones best suited for a clean implementation. For example, when you have seen an example using vcat′ you may in fact use it for lining diagrams up vertically without any spacing in between, only to find out later that there is also a vcat function to which you need only pass the diagrams and which gives the same behavior. By that time, you may even have defined a vcat of your own.

As you can easily see from the on-line manual page, we have omitted many, many interesting aspects of the library: dealing with text, images, envelopes, advanced colour

management and textures, trails and paths, all of them things you would expect in a tutorial. But as we explained earlier, our purpose here is not to provide such a tutorial, but to illustrate how a number of primitives (and fewer than you maybe expected), combined with a number of powerful combinators can result in a rich domain specific language for a given domain, in this case that of diagrams.

Error diagnosis

With this more complicated library/EDSL, type error diagnosis also suffers a bit more. For example, somebody wrote some on-line code resulting in

```
Couldn't match type 'Diagrams.Core.Types.QDiagram
                       b Diagrams.TwoD.Types.R2 Data.Monoid.Any'
              with 'a0 -> a0'
Expected type: (a0 -> a0)
               -> Diagrams.Core.Types.QDiagram
                    b Diagrams.TwoD.Types.R2 Data.Monoid.Any
  Actual type: Diagrams.Core.Types.QDiagram
                 b Diagrams.TwoD.Types.R2 Data.Monoid.Any
               -> Diagrams.Core.Types.QDiagram
                    b Diagrams.TwoD.Types.R2 Data.Monoid.Any
Relevant bindings include
  example :: Diagrams.Core.Types.Diagram b Diagrams.TwoD.Types.R2
    (bound at /tmp/1814470824.hs:9:1)
```

Although the message tells you where to look, it does reveal the underlying type structure of the *diagrams* package. And in many cases errors show up as a "missing instance" error message. For example, if you write circle 1 # lw, omitting the line width, you get

```
    Could not deduce (TrailLike (Measure R2))
      arising from a use of 'circle'
    from the context (HasStyle a, V a ~ R2)
      bound by the inferred type of
               it :: (HasStyle a, V a ~ R2) => a -> a
      at Top level
    Possible fix:
      add an instance declaration for (TrailLike (Measure R2))
    In the first argument of '(#)', namely 'circle 1'
    In the expression: circle 1 # lw
```

in which the message suggests to add an instance declaration, which is not a likely solution for this problem.

Beyond 2D

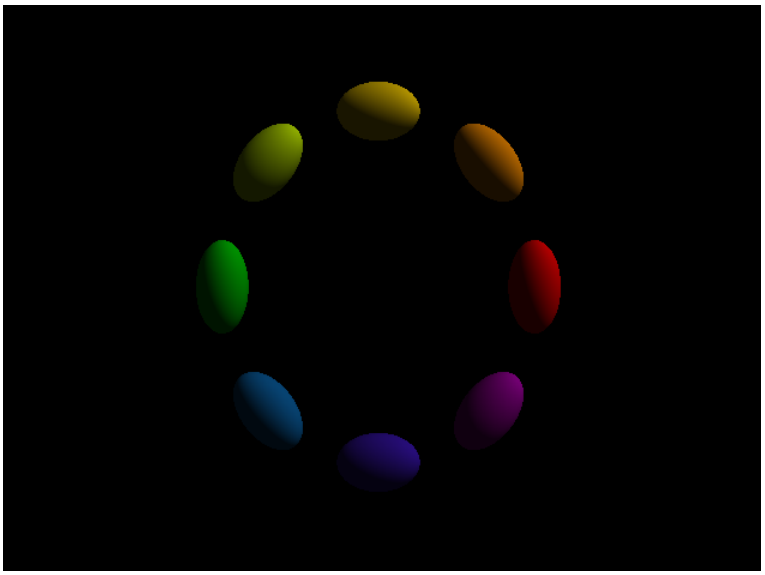


Fig. 16.8: Warped spheres

Thus far we have only seen 2D pictures, and that is for a very good reason: this chapter is not to explain how *diagrams* works but what makes it an EDSL. There is support however for creating 3D images, although a bit less mature and less well-documented than for 2D. We will restrict ourselves here to just giving an example (again taken from the website), of a number of warped spheres (see Fig. 16.8 for the result). Note the use of R3, and the POVRay backend in the diagram type signatures.

```
import Diagrams.Prelude.ThreeD
import Diagrams.Backend.POVRay
import Data.Colour.Palette.ColorSet

cam = mm50Camera # translateZ 40

_xy :: Spherical
_xy = direction . r3 $ (-1, -1, -0.5)

light = parallelLight _xy white

s = sphere # scaleY 1.6 # translateX 6

color theta = fc $ rybColor (floor $ theta * 24)
```

```
example :: Diagram POVRay R3
example = mconcat
  [transform (aboutZ (t @@ turn)) (s # color t) | t <- [0, 1 / 8 .. 7 / 8]]

main :: IO ()
main = putStrLn $ renderDia POVRay POVRayOptions $ mconcat [example, cam, light]
```

There is also the possibility of animating things, but again we do not provide any details.

16.5 Example 3: Lennart Augustsson's BASIC embedding

To conclude this chapter, and without any further explanation, we would like to illustrate just how far you can go reproducing the fluency of a domain (in this case, another general purpose language) in Haskell. BASIC was an early programming language (developed in 1964). A typical aspect of BASIC is the use of line numbers for statement, and the fact that jumping around in the code is based on these line numbers.

Lennart Augustsson embedded it in Haskell, and the code in Fig. 16.9 is in fact legal Haskell. Moreover, running main actually interprets the BASIC program following BASIC semantics. For more details visit Lennart's blog (`http://augustss.blogspot.nl/`)

```
{-# LANGUAGE ExtendedDefaultRules, OverloadedStrings #-}
import BASIC

main = runBASIC $ do
  10 GOSUB 1000
  20 PRINT "* Welcome to HiLo *"
  30 GOSUB 1000

  100 LET I := INT (100 * RND (0))
  200 PRINT "Guess my number:"
  210 INPUT X
  220 LET S := SGN (I - X)
  230 IF S <> 0 THEN 300

  240 FOR X := 1 TO 5
  250 PRINT X * X;" You won!"
  260 NEXT X
  270 STOP

  300 IF S <> 1 THEN 400
  310 PRINT "Your guess ";X;" is too low."
  320 GOTO 200

  400 PRINT "Your guess ";X;" is too high."
  410 GOTO 200

  1000 PRINT "********************"
  1010 RETURN

  9999 END
```

Fig. 16.9: A BASIC program embedded in Haskell

Bibliography

[1] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[3] M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2011.

[4] J. Hage and M. Odersky. Private communication, September 2011.

[5] F. Henglein et al. Hiperfit. Research project, `http://hiperfit.dk/`.

[6] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996.

[7] J. Hughes, M. Sheeran, K. Claessen, and P. Jansson. Raw fp: Productivity and performance through resource aware functional programming. `http://wiki.portal.chalmers.se/cse/pmwiki.php/RAWFP/RAWFP`.

[8] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[9] W. Taha. Plenary talk iii domain-specific languages. In *Computer Engineering Systems, 2008. ICCES 2008. International Conference on*, pages xxiii –xxviii, nov. 2008.