Functioneel Programmeren
2010-2011

J.D. Fokker
S. Holdermans
A. Löh
S.D. Swierstra

27 september 2010, 22:39

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

Inhoudsopgave

Voorwoord 2005-20010

De 2005-2006 versie van het dictaat verschilt op een aantal plaatsen van de vorige versie.

De belangrijkste overgang is het gevolg van het beschikbaar komen van een (weliswaar nog wat beperkt) klasse mechanisme in Helium, de in Utrecht gebouwde Haskell interpreter voor beginnend onderwijs in functioneel programmeren. Als gevolg hiervan is besloten om bij het behandelen van de taal Haskell, evenals vroeger het geval was, maar direct in het diepe te springen; het overloading mechanisme van Haskell is direct zichtbaar in het type van polymorfe functies. Deze overgang heeft een groot aantal (veelal zeer kleine) wijzigingen in het dictaat met zich mee gebracht, en er zijn er ongetwijfeld ook nog een flink aantal over het hoofd gezien. Om mogelijke verwarring te voorkomen zullen we op de website bij het vak de noodzakelijke correcties vermelden. U kunt ook zelf bijdragen aan deze verfijningsslag door deze bij docent of werkcollegeleiders te melden. Ook de studentassistenten kunnen uw meldingen verwerken. Zo hopen we dat we met vereende krachten z.s.m. weer een consistente versie van dit dictaat zullen hebben.

Omdat gebleken is dat velen in het begin moeite hebben zich de Haskell syntax eigen te maken is een appendix met zogenaamde syntaxdiagrammen opgenomen. Mocht je vastlopen in een foutmelding van een compiler (anders dan een melding over type fouten) dan kan je m.b.v. deze diagrammen relatief eenvoudig nagaan waar je programma van de Haskell syntax afwijkt. Wellicht is een goed idee om bij het bestuderen van het dictaat, en bij het maken van de opgaven, in deze diagrammen aan te tekenen welke taalconstructies geen problemen meer geven.

Nog een verandering is het naar voren halen van het hoofdstuk over input en output, zodat iedereen zo snel mogelijk in staat is een echt programma te schrijven.

Tenslotte zijn in een appendix een aantal tentamens uit het jaar 2004-2006 opgenomen, zodat extra oefenmateriaal beschikbaar is gekomen.

In de versie 2006-2007 zijn slechts een groot aantal kleine tekstuele aanpassingen gedaan. Verder zijn er aan de bij dit vak behorende website nogal wat extra links toegevoegd. De website www.haskell.org bevat in toenemende mate uitstekende tutorials over een keur van onderwerpen, zoals onder andere IO en het gebruik van monads. De behoefte deze ook in het dictaat op te nemen doet zich in steeds mindere mate gelden.

In de versie van 2007-2008 is een appendix toegevoegd met een introductiepracticum. Verder zijn een aantal zeer kleine correcties aangebracht.

In de versie 2008-2009 is het hoofdstuk over de parser combinators in een uitgebreidere Engelse versie toegevoegd.

In de versie 2009-2010 zijn hoofdstukken toegevoegd over Huffman-codering en een hoofdstuk over de interpretatie van Prolog in Haskell. Ook dit hoofdstuk is weer in het Engels, waarmee een stapje verder is gedaan in de richting van een volledig Engels dictaat.

In de versie van 2009-2010 zijn een aantal foutjes verholpen, en is er een hoofdstuk over het gebruik van wxHaskell toegevoegd. De huidge versie van het dictaat is er een die in transitie naar een volledig Engelse versie. Omdat dat nogal wat voeten in aarde heeft voorzien we vooreerst in een gemengde versie. Het spreekt voor zich dat we ons aanbevolen houden voor zowel globaal commentaar als gedetailleerd, en voor zowel inhoudelijk als textueel.

Hoofdstuk 1

Preface and Reading Guide

Why learning functional programming

Since the beginning of Computer Science literally thousands of programming languages have been designed, proposed, used, and discarded again. Before asking you to learn yet another programming language you may want to know why this new language is a real improvement.

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"[3]) had a very simple structure, and their mapping to machine code was straightforward and compilers hence 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.

Over the years however the so-called *semantic gap* between the expressivity of the programming language and the eventually generated machine code has widened, and most programmers nowadays have no idea how the eventually generated machine code looks like. Hence the process of programming has changed over the years too, by shifting the focus from pure code production to the process of expressing one's thought as effectively as possible. Besides the efficiency of the final code, other aspects have gained importance: *how long does it take to get a reasonably efficient program*, *how much can be re-used from library code*, *how well can the program be statically checked for inconsistencies* and *how well is the programming language and its compiler supported by the operating system and a programming environment.*

As a consequence of programs getting larger and larger, and hence the increasing emphasis on being able to re-use code, the question how well re-usability is supported by the programming language at hand has become of utmost importance: *can we easily re-use existing code*, *how can we write code such that it can be easily re-used*, *how can we enforce that the way we compose programs 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?*.

It in answering such questions where modern functional programming languages shine. But before being able to understand why, we have to explain a bit more about the essence of functional programming, which lies in the absence of state. All mainstream programming languages, such as Fortran, Algol-60 [**?**], Cobol[**?**], Simula-67[**?**], Smalltalk[**?**], C, C++ , Objective C, Java, C#, Perl, PHP, Python, or whatever you have – all being so-called so-called 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. The primitive command underlying all other commands is the *assigment statement* which changes the state of a variable by assigning a new value to it by overwriting the old value. The use of assigments in looping constructs, such as *for*- and *while*

statements, makes it posible for programs to run for a longer time. Unfortunately it is precisely this assigment 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 program (in Haskell-like syntax):

> **let** $x = getChar$
> **in if** $x \equiv x$ **then** *print* `"True"` **else** *print* `"False"`

Now everyone would expect this program to print `True`, and even we do so. Let us now take the definition of x and expand is literally at the sites of its use:

> **if** $getChar \equiv getChar$ **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 retuning the next character form the input, as a side-effect that the global state changes. We thus observe that imperative languages lack so-called *referential transparency*, i.e. the possibility to safely replace a name by its definition in any context. Or stated the other way around, to give a name to equal sub-structures 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 emphasize 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 assigments, how can we then have loops, and how can we then keep our machines busy for more than a fraction of a second?". 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** $\{ sum := sum + i := i - 1 \};$
> *return sum*
> $\}$

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

> *function sum_upto* $(n) =$
> **let** *function loop* $(i, sum) =$ **if** $i \equiv 0$ **then** *return sum* **else** *return* $(loop\ (i - 1, sum + i))$
> **in** *return* $(loop\ (n, 0))$

We can summarise the above with: *"In Purely functional programming we use recursion instead of loop constructs"*.

This being said there is however a lot which weakens our just reached conclusion, since many functional languages, such as LISP[**?**], Scheme [1] and ML [4], take a hybrid approach: they are heavily based on functions, but nevertheless allow assignments in programs too. It are only the purely functional languages such as Miranda [6], Clean [**?**] and Haskell [**?**] (in order of historical appearance), from which the concept of an assignment is completely banned, and where the promises of referential transparency are thus fully fulfilled.

Because modern functional languages have this concept of *referential transparency* they have provided playground for a lot of research on programming languages per se; this has lead to interesting

type systems, and many new ways of abstracting from all kind of language constructs. These developments are now finding their way back to more commonly used programming languages, such as Java, C# [**?**], F# [**?**] and Scala [5].

By studying functional languages we will develop a deeper insight in the fundamental structures underlying all programming languages, and the relationships between the many concepts which together make up for a full and expressive language.

The developments 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 honour 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 the expressive power of these formalisms is 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.

Unfortunately the fact that a formalism has universal expressive power does not make it a good programming language per se.

Why learning Haskell

In these lecture notes we have chosen Haskell as the main programming language, since it is by now the most widely used pure functional language, with a very 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.

Besides being a very nice programming language for a wide variety of applications, Haskell is also used by more theoretically oriented 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. More specifically it has become the main language at the ACM conference *International Conference on Functional Programming* (ICFP), and currently even has its own lively ACM *Haskell Symposium* (HS). Hence by learning Haskell you will implicitly become aware of many important programming language concepts, and their further development.

For the impatient we now provide a quick overview of the main characteristics of Haskell.

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 is name to the Backus-Naur Form (BNF), delivered his ACM Turing award lecture [2] 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 of this the style of programming is completely different from what you may be used to; you may even have to "unlearn" things you have learned thus far and once you become accustomed to the fact that you can freely abstract from almost everything, you will also want to do this when writing imperative programs. In your everyday way of writing imperative programs you will be much more aware of the fact that every now and then state changes.

One of the most popular programming paradigms is the so-called "object-oriented" approach, where the program manipulates statefull 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. We will see how we can get an equivalent effect when programming functionally, and will also see what the functional approach can add to this, such as easy recoverability, the ability to replay parts of a computation, etc.

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. One tends however to forget the other important aspect, i.e. *that 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 easy, 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. An incredible amount of programmer's time is wasted in debugging imperative programs in which this explicit scheduling is not sufficiently well thought out, and most security holes found stem from such things like the ability to follow pointers which point to data which no longer exists.

As we will see programming in Haskell takes away a large class of programming errors; the price paid is of course that data which is no longer accessible has to be collected by some general mechanism. Fortunately modern compilers are vey good at discovering when data is no longer available, and the speed of modern machines makes that we are more than in the past prepared to pay for the extra security and programming convenience.

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 (sometimes called 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 hared. This sharing, 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 provides very elegant formulations for otherwise complicated algorithmic problems; of course the new evaluation strategy comes with its own problems, which we have to become aware of. In addition lazy evaluation comes with its own overhead, since whenever a parameter is accessed it has to be checked whether it has been evaluated already or whether we are touching it for the first time. Also here progress in program analysis research has made a large contribution to making such dynamic tests largely superfluous.

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

$$ones = 1 : ones$$

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 :. 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.

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 an string value somewhere, and then read it back and use this bit-pattern as a pointer. As a consequence no pointer arithmetic is possible. 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 can check 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) performs these verifications while executing the program, which yields a considerable overhead and makes it very hard to locate such mistakes.

An advantage of such dynamically typed languages is that one does not have to explicitly assign types to variables, since it are the values themselves which 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 the types 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 lines of 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:

$$\begin{array}{l} procedure\ swap\ (var\ i, j : Integer); \\ var\ h : Integer; \\ begin\ h := i; i := j; j := h \\ end \end{array}$$

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 a multitude of 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:

$$swap :: \forall\ a \circ (a, a) \rightarrow (a, a)$$
$$swap\ (x, y) = (y, x)$$

here we have first specified the type, and next have given the function. One may now wonder how many functions one can write which have this type. A bit of reflection shows that there are four:

$$swap\ (x, y) = (y, x)$$
$$swap\ (x, y) = (x, x)$$
$$swap\ (x, y) = (y, y)$$
$$swap\ (x, y) = (x, y)$$

We see that if we take 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 s 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 *parametric polymorphic* [**?**]. Because we know that the function has to work for any type a this also implies that we cannot write functions like:

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

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

So 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)
```

We see to our surprise that the automatically inferred type is even more general than we might have thought! We can even swap two values of which the types differ; swapping a value of type $(Integer, Char)$ returns a value of type $(Char, Integer)$; the function swap is even the only function with this type and thus the type provides a precise specification of what the function does.

You will see that once you start mastering the art of writing your programs in an as general way as possible, you will be rewarded by the type checker by reporting inconsistencies in your program.

Class System

Besides having automatic type inference which supports parametric polymorphism, Haskell also has a way of *overloading* function names: i.e. to have several different functions 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 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 only goal to keep 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 been extended and extended, and currently the original design is just a minuscule fraction of what is currently available. New versions of the Haskell standard are expected to incorporate more and more of these new features, through which a whole new class of programming techniques becomes available; the class system has become a completely new way of exploiting the type system in telling the compiler how to generate type specific code. This takes away the necessity to write a lot of code explicitly. The area of *generic programming* (not to be confused with the "generics" in Java, which refers to parametric polymorphism) uses the class system to derive code which we know how to write once we know what type of values it has to deal with. Examples of generic functions are e.g. checking for equality, or serialising and deserialising values.

What does it entail to learn a new programming language

Learning a new programming language involves a number of quite different activities. We mention them here explicitly so that you, once you run into problems, can probably classify your problem, and know where to look for an answer.

Grammar

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 (if this were true most people would remain silent for their entire life). Haskell is defined in the Haskell98 report, but although quite readable, this is not the place to start from. It is however useful to keep in mind that once you run into a problem, you can try to look up the precise definition in the report. Currently a newer version of the report is in the works, so you should not be surprised if it suddenly shows up. On the other hand for you, as a beginner, there is no need to run for it.

Many courses teach Haskell in a stepwise fashion, gradually introducing new concepts. Keep in mind that Haskell is a large language, with many subtle corners. A dilemma for many courses is the question: "Should I 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 great 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 more clear later.

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.

Idioms

Every programming language has special ways of "how to say thing". 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 way, but am I right in pursuing this path, or is there a completely different way of doing this". Usually the answer to these questions is an affirmative "yes". It is only by having an open mind that you can appreciate the different ways of attacking things, and learning how to compare and judge them. In hte end you will build up a toolbox for yourself, containing a collection of such idioms, and you will apply them without even remembering that once you found this very intricate.

Style

Just as with other languages there are different styles of programming. On the one extreme we have people who 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. As an example look at the following versions of the same function *both*, 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; the function *map* takes two arguments, the function to be applied an the list containing the elements:

$$
\begin{array}{l}
mapBoth :: \forall\ a\ b\ c \circ (b \to c) \to (a \to b) \to [a] \to [c] \\
mapBoth\ g\ f\ xs = \textbf{let}\ ys = map\ f\ xs \\
\qquad\qquad\qquad\quad\ \ zs = map\ g\ ys \\
\quad \textbf{in} \qquad\qquad\quad\ \ zs
\end{array}
$$

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

$$mapBoth\ f\ g\ xs = map\ g\ (map\ f\ xs)$$

Some others may even like to use function composition denoted by a $\circ$, as in:

$$mapBoth\ f\ g\ xs = (map\ g \circ map\ f)\ xs$$

or even shorter:

$$mapBoth\ f\ g = map\ g \circ map\ f$$

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 $map\ g \circ map\ f \equiv map\ (g \circ f)$, so we get:

$$mapBoth\ f\ g = map\ (g \circ f)$$

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

$$
\begin{array}{lll}
mapBoth\ g\ f = & & map\quad((\circ)\ \ g\quad f) \\
mapBoth\ g\ f = (& & map \circ ((\circ)\ \ g))\ f \\
mapBoth\ g = & & map \circ ((\circ)\ \ g) \\
mapBoth\ g = & (\ (\circ)\ map & \quad((\circ)\ \ g) \\
mapBoth\ \ \ = & (((\circ)\ map) \circ (\circ)) & g \\
mapBoth\ \ \ = & (\ (\circ)\ map) \circ (\circ) &
\end{array}
$$

Somewhat surprisingly 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 piont in a space; here we could also have said "viariable-free"). We leave it up to you to develop your own style, and hope that you will become famous as a Haskell author.

Hoofdstuk 2

Functioneel Programmeren

Functionele talen

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 (assigment) zoals $x{:=}y{+}1$ 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 programmeerstijl*. Een 'programma' bestaat in hoofdzaak uit de definitie van een aantal functies. Bij het 'uitvoeren' van een programma wordt een functie van parameters voorzien, en moet het resultaat berekend worden. Bij die berekening is nog een zekere mate van vrijheid aanwezig. 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?

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, mar 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 wel een manier 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://shootout.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 een aanpassen van een klein gedeelte van een programma volstaan worden teneinde een in executietijd vergelijkbare versie te krijgen. In sommige gevallen zijn functionele formuleringen essentiëel sneller, en dienen de imperatieve varianten sterk herschreven te worden.

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 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, en daarmee in de hele Scandinavische industrie, werd de taal Erlang veel gebruikt. 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 inmiddels bekend staat als Haskell98 (inmiddels wordt er gewerkt aan een nieuwe versie hiervan bekend als Haskell' (prime)).

Er bestaan inmiddels een aantal verschillende compilers van deze taal (de 'Glasgow' (GHC), de 'York' (NHC)en de 'Chalmers' compiler) en een aantal interpreters. Naast de interpreter 'Hugs' bestaat er inmiddels ook een interpreteren versie van de GHC, 'ghci' genaamd, die zowel kan interpreteren als reeds vertaalde modules aanroepen.

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 "standaard zaken" nu via bibliotheken beschikbaar kunnen worden

gesteld.

De Haskell-interpreter

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 waarden 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.

Met het commando :? krijg je een lijst van mogelijke commando's, die je ook veelal via menu's kunt selecteren (figuur 2.1). 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. 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 functie-aanroep echter veel vaker voor dan vermenigvuldigen. Daarom wordt 'naast elkaar zetten' in Haskell geïnterpreteerd als functie-aanroep, 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
```

```
loeki:~ doaitse$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :h
 Commands available from the prompt:

   <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 a command :<cmd>
   :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
   :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
   :type <expr>                show the type of <expr>
   :undef <cmd>                undefine user-defined command :<cmd>
   :!<command>                 run the shell command <command>

 -- Commands for debugging:

   :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)
```

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*. We zullen overigens zien dat een lijst slechts één van de manieren is om gegevens samen te voegen, zodat functies op grote hoeveelheden gegevens tegelijk kunnen worden toegepast.

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 standaarduncties 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.

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; zo heeft de veel gebruikte editor Emacs een speciale Haskell-mode[1]. Ook bestaan er voor de ontwikkelomgeving Eclipse plug-ins die het ontwikkelen van Haskell programma's vergemakkelijkt.

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

[1] `http://www.haskell.org/haskell-mode/`

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. De file-extensie `.hs` wordt meestal gebruikt om aan te geven dat het om een *Haskell script* gaat. Er wordt nu een tekstverwerker opgestart.

In de file '`MijnEersteHaskellModule.hs`' kan nu bijvoorbeeld de definitie worden gezet van de faculteit-functie. 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 functie-definities 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
```

Opdrachten aan de interpreter

Naast `:?` en `:l` bstaan er nog andere opdrachten die direct voor de interpreter zijn bedoeld (en die dus niet als een uit te rekenen expressie worden beschouwd). Al deze opdrachten beginnen met een dubbele punt.

De volgende opdrachten zijn mogelijk:

:? Dit is een opdracht om een lijstje te maken van de andere mogelijke opdrachten. Handig om te weten te komen hoe een opdracht ook al weer heet ('je hoeft niet alles te weten, als je maar weet hoe je het kan vinden').

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

:l *file* (load) Na deze opdracht kent Haskell de functies die in de gespecificeerde file zijn gedefinieerd.

:t *expressie* (type) Door deze opdracht wordt het type (zie sectie 2) van de gegeven expressie bepaald. blz. 28

Standaardfuncties

Ingebouwd/voorgedefinieerd

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

$$pi = 3.1415927$$

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 had misschien beter als operator gedefinieerd kunnen worden, en bijvoorbeeld als `!^!` genoteerd kan worden: blz. 15

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

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 een handige manier om te weten te komen wat een standaardfunctie doet. Voor *sum* luidt de definitie bijvoorbeeld

$$sum = foldl'\ (+)\ 0$$

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.

Dit soort functies wordt *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.

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 apostrof-teken (') en het onderstrepings-teken (_). Kleine letters en hoofdletters worden als verschillende letters beschouwd. Een paar voorbeelden van mogelijke functie- of parameter-namen zijn:

$$f\ sum\ x3\ g'\ tot\_de\_macht\ langeNaam$$

Het onderstrepings-teken 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. Ook in andere programmeertalen wordt dit vaak gedaan.

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
blz. 115 zogenaamde *constructor-functies*. De definitie daarvan wordt beschreven in paragraaf 8.

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 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:

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

In dit diktaat zullen we in de geformatteerde programma's hiervoor soms wiskundige symbolen gebruiken, en zien de operatoren `&&`, `||`, `<=` en `=/` er uit als $\wedge$, $\vee$, $\leqslant$ en $\not\equiv$.

De operatoren op de eerste van deze twee regels worden in de prelude gedefinieerd. Op de tweede regel staan operatoren die je zelf gedefinieerd zouden kunnen hebben. Operatoren die met een dubbele punt (:) beginnen kunnen uitsluitend gebruikt worden voor *constructor-operatoren* (net zoals namen die met een hoofdletter beginnen dat zijn voor constructor-functies).

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

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

Gelukkig blijven er nog genoeg over om je creativiteit op bot te vieren...

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` 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 $0.5\text{e} - 2$ 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 tussen *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 zijn de wordt geselecteerd. Vooreerst volstaat de interpretatie: "$+$ werkt tussen alles wat je kan optellen". Een tipje van de sluier wordt opgelicht als je in de interpretator GHCI vraagt naar het type van de operator $+$:

```
$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
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 voolopig 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 vierkanstswortel-functie
sin	de sinus-functie
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

Gehele getallen moeten kleiner blijven dan 2^{31}, anders treedt er *overflow* op:

```
Prelude> 3 * 100000000
300000000
Prelude> 3*1000000000000000 :: Int
 -296517632
Prelude> 3*1000000000000000 :: Integer
3000000000000000
```

In het middelste geval hebben we explciet 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 onzin antwoord. In het laatste geval hebben we aangegeven dat het resultaat met een voldoende grote precisie berekend dient te worden, en krijgen we wel het juiste resultaat.

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.

Je kunt in Haskell ook met gehele getallen groter dan 2 miljard rekenen. Je moet dan echter aangeven dat het type van de getallen *Integer* is, in plaats van het gebruikelijke *Int*. In dat geval kunnen de getallen zo groot worden als maar nodig is:

```
? 123456789 * 123456789 :: Integer
15241578750190521
```

Boolese 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* of *Boolean values* (genoemd naar de Engelse wiskundige George Boole). Functies (en operatoren) die zo'n waarde opleveren heten *Boolean functions* of *Boolese functies*.

Behalve $<$ bestaat er ook een operator $>$ (groter-dan), een operator $\leqslant$ (in programmatekst `<=`) (kleiner-of-gelijk), en een operator $\geqslant$ (groter-of-gelijk, in programmatekst `>=`). Daarnaast bestaan er operatoren $\equiv$ (gelijk-aan, `==`) en operator $\not\equiv$ (ongelijk-aan, `/=`). Voorbeelden:

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


? sqrt 2.0 == 1.5
False
```

Uitkomsten van Boolese functies kunnen gecombineerd worden met de operatoren $\wedge$ ('en', `&&`) en || ('of', `||`). De operator $\wedge$ 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 $\neg$ 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
```

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. De operator ++ plakt twee lijsten aan elkaar. Bijvoorbeeld:

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

De functie *null* is een Boolese functie op lijsten. Deze functie kijkt of een lijst leeg is (geen elementen bevat). De functie *and* werkt op een lijst waarvan de elementen Boolese 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.

Functies op functies

In de functies die tot nu toe besproken zijn zijn de parameters getallen, Boolese 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 de 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
blz. 37 'functionele' taal!). In hoofdstuk 3 worden meer van dit soort functies besproken.

Functie-definities

Definitie door combinatie

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

$$
\begin{array}{lll}
\mathit{fac}\ n & & = \mathit{product}\ [1\,..\,n] \\
\mathit{oneven} & x & = \neg\ (\mathit{even}\ x) \\
\mathit{kwadraat} & x & = x * x \\
\mathit{som\_van\_kwadraten}\ \mathit{lijst} & & = \mathit{sum}\ (\mathit{map}\ \mathit{kwadraat}\ \mathit{lijst})
\end{array}
$$

Functies kunnen ook meer dan één parameter krijgen:

$$
\begin{array}{lll}
boven & n\ k & = fac\ n\ /\ (fac\ k * fac\ (n-k)) \\
abcFormule & a\ b\ c & = [(-b + sqrt\ (b*b - 4.0*a*c))\ /\ (2.0*a) \\
 & & \ ,(-b - sqrt\ (b*b - 4.0*a*c))\ /\ (2.0*a) \\
 & & \ ]
\end{array}
$$

Functies met nul parameters worden meestal 'constanten' genoemd:

$$
\begin{array}{l}
pi = 3.1415926535 \\
e\ \ = exp\ 1.0
\end{array}
$$

Elke functie-definitie heeft dus de volgende vorm:

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

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

$$
\begin{array}{l}
negatief\ x = x < 0 \\
positief\ x\ \ = x > 0 \\
isnul\ x\ \ \ \ = x \equiv 0
\end{array}
$$

Let in de laatste definitie op het verschil tussen de = en de ≡. Een enkel is-teken (=) scheidt in functiedefinities de linkerkant van de rechterkant. Een dubbel is-teken (≡, in de programmatekst geschreven als `==`) 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 deel-expressies worden tweemaal uitgerekend. Om dat te voorkomen, is het in Haskell mogelijk om deel-expressies een naam te geven. De verbeterde definitie wordt dan als volgt:

$$
\begin{array}{lll}
abcFormule' & a\ b\ c & = [(-b + d)\ /\ n \\
 & & \ ,(-b - d)\ /\ n \\
 & & \ ] \\
 & & \ \textbf{where}\ d = sqrt\ (b*b - 4.0*a*c) \\
 & & \ \ \ \ \ \ \ \ \ \ n = 2.0*a
\end{array}
$$

Het woord **where** is niet de naam van een functie: het is één van de 'gereserveerde woorden' die in paragraaf 2 opgesomd zijn. Achter '**where**' staan definities. In dit geval definities van blz. 18
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. Gedurende het berekenen van één aanroep van *abcFormule'*, bij een gegeven a, b en c, zijn ze echter constant.

Definitie door gevalsonderscheid

Soms is het nodig om in de definitie van een functie meerdere gevallen te onderscheiden. De absolute-waarde functie *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:

$$\begin{array}{l} abs\ x \mid x < 0 = -x \\ \qquad\ \mid x \geqslant 0 = \ \ x \end{array}$$

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

$$\begin{array}{l} signum\ x \mid x > 0 = \ \ 1 \\ \qquad\qquad\ \mid x \equiv 0 = \ \ 0 \\ \qquad\qquad\ \mid x < 0 = -1 \end{array}$$

De definities voor de verschillende gevallen worden 'bewaakt' door Boolese 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 Boolese expressie, een =-teken, en een expressie[2]. Deze beschrijving is echter ook nog niet volledig...

Definitie door patroonherkenning

De parameters van een functie in een functie-definitie, zoals x en y in

$$f\ x\ y = x * y$$

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

$$f\ 17\ (1 + g\ 6)$$

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

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 functie-definitie, waarin een patroon wordt gebruikt als formele parameter is:

$$f\ [1, x, y] = x + y$$

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

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

is heel normaal dat functies niet voor alle mogelijke actuele parameters gedefinieerd zijn. Zo is bijvoorbeeld de functie *sqrt* niet gedefinieerd voor negatieve parameters, en de operator / niet voor 0 als rechter parameter.)

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

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

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 $som\ [3, 4]$ 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 variable 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 $[1, x, y]$);
- De operator : met patronen links en rechts (bijvoorbeeld $a : b$);

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

$$\begin{aligned} &False \wedge False = False \\ &False \wedge True = False \\ &True \wedge False = False \\ &True \wedge True = True \end{aligned}$$

Met de operator : kunnen lijsten worden opgebouwd. De expressie $x : y$ 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:

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

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'.

Definitie door recursie of inductie

In de definitie van een functie mogen standaardfuncties en zelf-gedefinieerde 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:

$$f\ x = f\ x$$

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:

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

Een recursieve definitie van de faculteit-functie is de volgende:

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

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

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

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

Ook in dit geval is de parameter van de recursieve aanroep (n) kleiner dan de parameter van de te definiëren functie $(n+1)$.

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:

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

Een recursieve definitie waarin voor het gevalsonderscheid patronen worden gebruikt (in plaats van Boolese 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 Boolese expressies) luidt als volgt:

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

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

$$\begin{aligned} &som\ [\,] = 0 \\ &som\ (kop : staart) = kop + som\ staart \end{aligned}$$

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:

$$\begin{aligned} &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (\mathit{kop} : \mathit{staart}) &&= 1 + \mathit{length}\ \mathit{staart} \end{aligned}$$

Daarbij is de waarde van het *kop*-element niet van belang (alleen het feit dàt er een kop-element is).

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

$$\begin{aligned} &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (\_ : \mathit{staart}) &&= 1 + \mathit{length}\ \mathit{staart} \end{aligned}$$

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 functie-definitie 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:

$$\begin{array}{ll} \textbf{where} & \textbf{where} \\ \quad a = f\ x\ y & \quad a = f\ x \\ \quad b = g\ z & \quad y\ b = g\ z \end{array}$$

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

$$\begin{array}{lll} f\ x\ y = g\ (x + w) & & \\ \qquad \textbf{where}\ g\ u = u + v & & \\ \qquad\qquad\qquad\quad \textbf{where}\ v = u * u & & \\ \qquad\qquad\quad w = & 2 + y & \end{array}$$

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.

Het is 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 evenver 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
blz. 18 $<--->$. Een losse `--` kan echter geen operator zijn: deze combinatie werd in paragraaf 2 gereserveerd.

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.

Typering

Soorten fouten

Vergissen is menselijk, ook bij het schrijven of intikken van een functie. Gelukkig kan de interpreter waarschuwen voor sommige fouten. Als een functie-definitie niet aan de vorm-eisen 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 Helium interpreter:

```
Prelude> let isNul x = x = 0

<interactive>:1:16: parse error on input `='
```

De Ghci geeft een wat cryptischer foutmelding.

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 syntax-fouten 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 tik-fout. Bij het analyseren van de definitie

$$fac\ x = produkt\ [1\,..\,x]$$

meldt de Helium interpreter:

```
(1,9): Undefined variable "produkt"
  HINT - Did you mean "product" ?
```

en de GHCI:

```
Reading script file "nieuw.hs":
ERROR "nieuw.hs" (line 19): Undefined variable "produkt"
```

Deze fouten worden pas opgespoord tijdens de tweede fase: de afhankelijkheids-analyse (*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 Boolese 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:

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

De deel-expressie (*term*) *True* heeft het *type Bool* (een afkorting van 'Boolese waarde'). Zo'n boolean-waarde kan niet worden opgeteld bij 1, wat van het type *Int* is (een afkorting van *integer, oftewel geheel getal*). Sterker nog, boolean-waardes kunnen helemaal niet opgeteld worden.

De GHCI geeft een nog cryptischer melding als we vragen om de waarde van $1 +$ *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 Haskell definitie. De operator + kan veel verschillende betekenissen hebben, maar kennelijkniet die van een functie die twee Boolean waarden 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.

Andere typerings-fouten treden bijvoorbeeld op bij het toepassen van de functie *length* op iets anders dan een lijst, zoals in *length* 3:

```
(1,20): Type error in application
*** Expression     : length 3
*** Term           : 3
*** Type           : Int
*** Does not match : [a]
```

Pas als er geen typerings-fouten meer in een programma zitten, kan de vierde analyse-fase (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 typerings-fouten meer.

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 typerings-fout 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.

Typering van expressies

Het type van een expressie kan bepaald worden met de interpreter-opdracht `: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.

Er zijn aantal basis-types:

- *Int*: het type van de gehele getallen (*integer numbers* of *integers*), tot een maximum van ruim 2 miljard;
- *Integer*: het type van gehele getallen, zonder praktische begrenzing[3]
- *Float*: het type van de floating-point getallen;
- *Bool*: het type van de Boolese waardes *True* en *False*;
- *Char*: het type van letters, cijfers en symbolen op het toetsenbord (*characters*), dat in paragraaf 6 zal worden besproken.

blz. 76

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 bools, 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 typerings-fout:

```
(1,20): Type error in element of list
*** Expression     : [1,True]
*** Term           : True
*** Type           : Bool
*** Does not match : Int
```

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:

[3] niet in Helium

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

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

Andere voorbeelden van types van functies zijn:

$$\begin{aligned} &\mathit{sqrt} &&:: \mathit{Float} \rightarrow \mathit{Float} \\ &\mathit{even} &&:: \mathit{Int} \rightarrow \mathit{Bool} \\ &\mathit{reverse} &&:: [\mathit{Int}] \rightarrow [\mathit{Int}] \end{aligned}$$

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, Boolese 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:

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

Hoewel zo'n *type-declaratie* 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.

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:

$$\mathit{length} :: [a] \rightarrow \mathit{Int}$$

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:

$head :: [a] \to a$

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 type-variabele gebruikt als voor het type van de elementen van de lijst.

Een type waar type-variabelen in voorkomen heet een *polymorf type* (letterlijk: 'veelvormig type'). Functies met een polymorf type heten polymorfe functies. Het verschijnsel zelf heet *polymorfie* of *polymorfisme*.

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 identiteits-functie (de functie die zijn parameter onveranderd oplevert):

$$\begin{aligned} id &:: a \to a \\ id\ x &= x \end{aligned}$$

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 Boolese 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 $a \to a$ hebben, zodat de functie *id* ook op zichzelf kan worden toegepast: $id\ id$.

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
blz. 16 paragraaf 2 heeft twee integer parameters en een integer resultaat. Het type is daarom:

$boven :: Int \to Int \to Int$

blz. 22 De functie *abcFormule* uit paragraaf 2 heeft drie floating-point getallen als parameter en een lijst met floating-point getallen als resultaat. De type-declaratie luidt daarom:

$abcFormule :: Float \to Float \to Float \to [Float]$

blz. 22 In paragraaf 2 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:

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

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 functie-parameter (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 functie-parameter.

In de type-declaratie van *map* moeten er haakjes staan om het type van de eerste parameter $(a \to b)$. 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 $(a \to b)$ 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:

$$(\wedge) :: Bool \to Bool \to Bool$$

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 $Int \to Int \to Int$ als $Float \to Float \to Float$ zijn. Toch is + niet echt een polymorfe operator. Als het type $a \to a \to a$ zou zijn, zou de operator namelijk ook op bijvoorbeeld *Bool* parameters moeten werken, hetgeen niet mogelijk is. Zo'n functie die 'een beetje' polymorf is, wordt een *overloaded* functie genoemd.

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 $a \to a \to a$ mits a een type is in de klasse *Num*'.

Let op het gebruik van het pijltje met dubbele stok ($\Rightarrow$ of desgewenst $\Rightarrow$). 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:

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

Zelf-gedefinieerde functies kunnen ook overloaded zijn. Bijvoorbeeld de functie

$$kwadraat\ x = x * x$$

heeft het type

$$kwadraat :: Num\ a \Rightarrow a \to a$$

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]
```

blz. 128 Het gebruik van klassen en de definitie ervan wordt uitgebreid besproken in hoofdstuk 9. Klassen werden hier alleen kort genoemd om de overloaded operatoren te kunnen typeren.

Opgaven

2.1 De taal Haskell is genoemd naar Haskell B. Curry. Wie was dat? (Zoek op op internet).

2.2 Als de onderstaande dingen in een programma staan, zijn ze dan:
- iets met een vaststaande betekenis (gereserveerd woord of symbool);
- naam van een functie of parameter;
- een operator;
- niets van dit alles?

Als het een functie of operator is, betreft het dan een 'constructor'-functie respectievelijk -operator?

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

2.3 Hoeveel is:

$4.0e3 + 2.0e\text{-}2$
$4.0e3 * 2.0e\text{-}2$
$4.0e3 \;/\; 2.0e\text{-}2$

2.4 Wat is het verschil in betekenis tussen $x = 3$ en $x \equiv 3$?

2.5 Schrijf een functie *aantalOpl* die, gegeven a, b en c, het aantal oplossingen van de vergelijking $ax^2 + bx + c$ oplevert, in twee versies:
- met gevalsonderscheid
- door combinatie van standaardfuncties

blz. 28 **2.6** Wat is het voordeel van 'genest' commentaar (zie paragraaf 2)?

2.7 Wat is het type van de volgende functies: *tail*, *sqrt*, *pi*, *exp*, $(\uparrow)$, $(\not\equiv)$ en *aantalOpl*? Hoe kan je aan de interpreter vragen om dat type te bepalen, en hoe kun je de types zelf specificeren in een programma?

2.8 Stel dat x de waarde 5 heeft. Wat is de waarde van de expressies $x \equiv 3$ en $x \not\equiv 3$? (Voor wie de programmertaal C kent: Wat is de waarde van deze expressies in de taal C?)

2.9 Wat betekent '*syntax error*'? Wat is het verschil tussen een *syntax error* en een *type error*?

2.10 Bepaal de types van 3, *even* en *even* 3. Hoe doe je dat laatste? Bepaal nu ook het type van *head*, $[1, 2, 3]$ en *head* $[1, 2, 3]$. Wat gebeurt er bij het toepassen van een polymorfe functie op een actuele parameter?

2.11 Wat is het type van de volgende expressies:

- *until even*
- *until or*
- *foldr* $(\wedge)$ *True*
- *foldr* $(\wedge)$
- *foldr until*
- *map sqrt*
- *map filter*
- *map map*

2.12 Als voorwaarde voor het zinvol-zijn van een recursieve definitie staat in paragraaf 2 de voorwaarde genoemd dat de parameter bij de recursieve aanroep eenvoudiger moet zijn, en dat er een niet-recursief basis-geval moet zijn. Beschouw nu de volgende definitie van de faculteitfunctie: blz. 25

$$\begin{array}{l} fac\ n \mid n \equiv 0 \quad\ = 1 \\ \qquad\ \mid otherwise = n * fac\ (n-1) \end{array}$$

a. Wat gebeurt er bij de aanroep $fac\ (-3)$?

b. Hoe kan je de voorwaarde waaronder een recursieve definitie zinvol is nauwkeuriger formuleren?

2.13 Wat is het verschil tussen een *lijst* en het wiskundige begrip *verzameling*?

2.14 In paragraaf 2 wordt een recursieve definitie voor machtsverheffen gegeven. blz. 25

a. Geef nu een alternatieve definitie voor machtsverheffen, waarbij je de gevallen dat n even en oneven is apart behandelt. Je kunt daarbij gebruik maken van het feit dat $x^n = (x^{n/2})^2$.

b. Welke tussenresultaten worden er berekend bij het uitrekenen van 2^{10} bij de oude definitie en bij de nieuwe definitie?

2.15 Stel dat is gedefinieerd:

$$\begin{array}{ll} driekopie\ x & = [x, x, x] \\ sums & (x : y : xs) = x : (x + y : ys) \\ sums & xs = xs \end{array}$$

Wat is dan de waarde van de expressie

$$map\ driekopie\ (sums\ [1\,..\,4])$$

Hoofdstuk 3

Getallen en functies

Operatoren

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 voor de precieze regels voor naamgeving). blz. 18

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, en de apostrof links van de 'return'-toets.)

Het is dus toegestaan (+) 1 2 te schrijven in plaats van $1 + 2$. Deze notatie is in paragraaf 9 ook gebruikt om het type van + te kunnen declareren: blz. 128

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

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 meerder 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 ‘*boven*‘ 3 leest nu eenmaal makkelijker dan *boven* 5 3. Dit kan natuurlijk alleen als de functie twee parameters heeft.

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

[1] Onlangs 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.

[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. Op mijn school prefereerde men

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 prioriteits-nivo's. De vergelijkings-operatoren, zoals $<$ en $\equiv$, 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 typerings-fout 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	$\circ$ (`.`) en `!!`
nivo 8	`^`
nivo 7	$*$, /, '*div*', '*rem*' en '*mod*'
nivo 6	$+$ en $-$
nivo 5	:, $+\!\!+$ en `\\`
nivo 4	$\equiv$, $\not\equiv$,$<$, $\leqslant$, $>$, $\geqslant$, $\in$ (`` `elem` ``) en $\notin$ (`` `notElem` ``)
nivo 3	$\wedge$
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 deel-expressies 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 *kwadraat* $3 + 4$ berekent dus het kwadraat van 3, en telt daar 4 bij op. Zelfs als je schrijft *kwadraat* $3 + 4$ wordt eerst de functie aangeroepen, en dan pas de optelling uitgevoerd. Om het kwadraat van 7 te bepalen zijn haakjes nodig om de hoge prioriteit van functie-aanroep te doorbreken: *kwadraat* $(3 + 4)$.

blz. 24

Ook bij het definiëren van functies met gebruik van patronen (zie paragraaf 2) is het van belang te bedenken dat functie-aanroep altijd voor gaat. In de definitie

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

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

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 in welke volgorde hij berekend moet worden. 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 $(a \oplus b) \oplus c$;
- 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;

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'.

- 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 rechts-associatief. (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 *functie-applicatie*, dus $f\ x\ y$ moet gelezen worden als $(f\ x)\ y$ (de reden hiervoor wordt besproken in sectie 3); blz. 40
- de operator !! (zie paragraaf 6); blz. 66
- 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 6.16); blz. 91
- de vergelijkings-operatoren $\equiv$, $<$ enzovoort: het heeft meestal toch geen zin om $a \equiv b \equiv c$ te schrijven. Probeer je dat toch dan krijg je de volgende melding:

  $$\begin{array}{l} \mathit{Fail} : \mathit{ambiguous\ use}\ \textbf{of}\ \mathit{non-associative\ operator} \equiv \mathit{at}\ (1,13) \\ \quad \mathit{with\ non-associative\ operator} \equiv \mathit{at}\ (1,16)\,\} \end{array}$$

  (*ambiguous* betekent: 'dubbelzinnig', 'voor meerdere uitleg vatbaar');
  Wil je testen of x tussen 2 en 8 ligt, schrijf dan niet $2 < x < 8$ maar $2 < x \wedge x < 8$.

De volgende operatoren zijn associatief:

- de operatoren $*$ en $+$ (deze operatoren worden overeenkomstig wiskundige traditie links-associërend uitgerekend);
- de operatoren $+\!\!+$, $\wedge$ en || (deze operatoren worden rechts-associërend uitgerekend omdat dat efficiënter is);
- de operator voor functiecompositie $\circ$ (zie paragraaf 3). blz. 45

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 $8\ \uparrow$

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

infixl $6\ +,\ -$
infix $4\ \equiv, \not\equiv, \in$

Door een slimme keuze voor de prioriteit te maken, kunnen haakjes in expressies zo veel mogelijk worden vermeden. Bekijk nog eens de operator 'n boven k' uit paragraaf 2: blz. 16

$$n\ `\mathit{boven}`\ k = \mathit{fac}\ n\ /\ (\mathit{fac}\ k * \mathit{fac}\ (n-k))$$

of met een zelfbedacht symbool:

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

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

Voor de prioriteit van '*boven*' 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 '*boven*' b '*boven*' c uit te rekenen, kan de operator het beste non-associatief gemaakt worden. De prioriteits-definitie luidt al met al:

infix 5 !^!, '*boven*'

Currying

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 argumenten 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:

$$opvolger :: Int \to Int$$
$$opvolger = plus\ 1$$

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 functie-parameter 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 *opvolger*, het type $Int \to Int$ heeft, dan is *plus* zelf blijkbaar een functie van *Int* (het type van 1) naar dat type:

$$plus :: Int \to (Int \to Int)$$

Door af te spreken dat $\to$ naar rechts associeert, zijn de haakjes hierin overbodig:

$$plus :: Int \to Int \to Int$$

blz. 32

Dit is precies de notatie voor het type van een functie met twee parameters, die in paragraaf 2 werd besproken.

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 een 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).

Haakjes

De 'onzichtbare operator' functie-toepassing 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 functie-toepassing 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:

$$f :: A \to B \to C \to D \to E$$

of, als je alle haakjes zou schrijven:

$$f :: A \to (B \to (C \to (D \to E)))$$

Zonder haakjes is dit alles natuurlijk veel overzichtelijker. De associatie van $\to$ en functie-applicatie is daarom zó gekozen, dat Currying 'geruisloos' verloopt: functie-applicatie associeert naar links, en $\to$ associeert naar rechts. In een makkelijk te onthouden slagzin:

als er geen haakjes staan,
staan ze zó, dat Currying werkt.

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

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

  De haakjes in $(a \to b)$ 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:

  $$kwadraat\ (sin\ pi)$$

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

Operator-secties

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 *operator-secties*.

Met operator-secties kunnen een aantal functies gedefinieerd worden:

$$\begin{array}{ll} opvolger & = (+1) \\ verdubbel & = (2*) \\ halveer & = (/2.0) \\ omgekeerde & = (1.0/) \\ kwadraat & = (\uparrow 2) \\ tweeTotDe & = (2\uparrow) \\ eencijferig & = (\leqslant 9) \\ isNul & = (\equiv 0) \end{array}$$

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

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

Functies als parameter

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. Het zoeken van overeenkomsten en het uitfactoriseren hiervan is een soort tweede natuur geworden voor veel functionele programmeurs, en het geeft vaak een goed inzicht in de essentie van een algorithme. Door hoofd- en bijzaken goed te scheiden kunnen zeer algemeen toepasbare bibilotheken worden geschreven.

Functies met functieparametersr worden soms *hogere-orde functies* genoemd, om ze te onderscheiden van 'laag-bij-de-grondse' numerieke 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:

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

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.

Omdat er voor ook voor veel andere data types een map-achtige functie bestaat is er in de prelude ook een functie *fmap* (functorial map) gedefinieerd: Dat merk je als je het type van *fmap* opvraagt:

```
? :t fmap
fmap :: Functor f => (a->b) -> f a -> f b
```

De tweede parameter van *fmap* is blijkbaar niet een lijst-van-a's, maar een f-van-a's, waarbij f een 'Functor' moet zijn. Omdat ook de lijstconstructor $[\ldots]$ zo'n *Functor* is, waarbij je bij *Functor* aan zoiets kan denken als een "functie die een type op een type afbeeldt" kun je dus overal waar je *map* schrijft ook *fmap* schrijven. Om historische redenen is voor lijsten *map* ingeburgerd geraakt, en we zullen deze conventie hier verder volgen. Kom je in een foutmelding het woord *Functor* tegen, interpreteer dat dan voorlopig als 'lijst-achtig'.

Een andere veel gebruikte hogere-orde functie op lijsten is *filter*. Deze functie levert díe elementen uit een lijst[3] , 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 functie-parameter van *filter* het type $a \to Bool$. Ook de definitie van *filter* is recursief:

$$
\begin{array}{lll}
filter & & :: (a \to Bool) \to [a] \to [a] \\
filter\ p\ [\,] & & = [\,] \\
filter\ p\ (x : xs) & \mid p\ x & = x : filter\ p\ xs \\
& \mid otherwise & = filter\ p\ xs
\end{array}
$$

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 Boolese waarden allemaal *True* zijn):

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs \\
and & [\,] & = True \\
and & (x : xs) & = x \wedge and\ xs
\end{array}
$$

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 $\wedge$).

[3] Nou ja, eigenlijk een *MonadZero*, wat weer een bijzonder soort *Functor* is, maar we gaan hier niet moeilijk zitten doen...

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

$$\begin{aligned} &\mathit{foldr}\ \mathit{op}\ e\ [\,] &&= e \\ &\mathit{foldr}\ \mathit{op}\ e\ (x : xs) &&= x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ xs \end{aligned}$$

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

$$\begin{aligned} \mathit{sum} &= \mathit{foldr}\ (+)\ 0 \\ \mathit{product} &= \mathit{foldr}\ (*)\ 1 \\ \mathit{and} &= \mathit{foldr}\ (\wedge)\ \mathit{True} \end{aligned}$$

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

$$\mathit{foldr}\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

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

De functie *foldr* 'vouwt' de lijst ineen tot één waarde, door ale "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.

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:

$$\mathit{until} :: (a \to \mathit{Bool}) \to (a \to a) \to a \to a$$

De functie heeft drie parameters: de eigenschap waar het eindresultaat aan moet voldoen (een functie $a \to \mathit{Bool}$), de functie die steeds wordt toegepast (een functie $a \to a$), 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 recursief. Het recursieve en het niet-recursieve geval worden ditmaal niet onderscheiden door patronen, maar door gevalsonderscheid met 'verticale streep/Boolese expressie':

$$\begin{aligned} \mathit{until}\ p\ f\ x \mid p\ x &= x \\ \mid \mathit{otherwise} &= \mathit{until}\ p\ f\ (f\ x) \end{aligned}$$

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* $(\equiv 0)$ $(+1)$ 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!
?
```

Samenstelling

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 '*na*' is, dan is het bijvoorbeeld mogelijk om te definiëren:

$$\begin{aligned} &\mathit{oneven} &&= \neg\ `\mathit{na}`\ \mathit{even} \\ &\mathit{dichtbijNul} &&= (<10)\ `\mathit{na}`\ \mathit{abs} \end{aligned}$$

De operator '*na*' kan als hogere-orde operator worden gedefinieerd:

$$\begin{aligned} &\textbf{infixr}\ 8\ `\mathit{na}` \\ &f\ `\mathit{na}`\ g = h \\ &\qquad\qquad \textbf{where}\ h\ x = f\ (g\ x) \end{aligned}$$

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 *na*:

$$\mathit{na} :: (b \to c) \to (a \to b) \to (a \to c)$$

Omdat $\to$ naar rechts associeert, is het derde paar haakjes overbodig. Het type van *na* kan dus ook geschreven worden als

$$\mathit{na} :: (b \to c) \to (a \to b) \to a \to c$$

De functie *na* 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 *na* worden gedefinieerd als functie met drie parameters:

$$\mathit{na}\ f\ g\ x = f\ (g\ x)$$

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 *oneven* hierboven wordt *na* dus in feite partieel geparametriseerd met $\neg$ en *even*. De derde parameter is nog niet gegeven: deze wordt pas ingevuld als *oneven* wordt aangeroepen.

Het nut van de operator *na* lijkt misschien beperkt, omdat functies als *oneven* ook gedefinieerd kunnen worden door

$$oneven\ x = \neg\ (even\ x)$$

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 `na` even) [1..100]
```

In de prelude wordt de functiesamenstellings-operator 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 functie-nivo plaatsvinden (zie 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 $f = g \circ h \circ i \circ j \circ k$ te kunnen schrijven in plaats van $f\ x = g\ (h\ (i\ (j\ (k\ x))))$?

De lambda-notatie

blz. 40

In paragraaf 3 werd opgemerkt dat de functie die je als parameter meegeeft aan een andere functie vaak ontstaat door partiële parametrisatie:

$$map\ (plus\ 5)\ [1\,..\,10]$$
$$map\ (*2)\ [1\,..\,10]$$

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

$$filter\ (\neg \circ even)\ [1\,..\,10]$$

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 in een lijst. Het is dan altijd mogelijk om de functie apart te definiëren in een **where**-clausule:

$$ys = map\ f\ [1\,..\,100]$$
$$\quad \textbf{where}\ f\ x = x*x+3*x+1$$

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:

$$ys = map\ (\lambda x \rightarrow x*x+3*x+1)\ [1\,..\,100]$$

Numerieke functies

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 $(x \mathbin{/} 10)$ ‘*rem*‘ 10. Het op twee na laatste $(x \mathbin{/} 100)$ ‘*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 priemgetallen

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

$$\begin{aligned} &\mathit{deelbaar} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool} \\ &\mathit{deelbaar}\ t\ n && = t\ \text{‘}\mathit{rem}\text{‘}\ n \equiv 0 \end{aligned}$$

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

$$\begin{aligned} &\mathit{delers} && :: \mathit{Int} \to [\mathit{Int}] \\ &\mathit{delers}\ x && = \mathit{filter}\ (\mathit{deelbaar}\ x)\ [1 \mathinner{..} x] \end{aligned}$$

De functie *deelbaar* wordt hierin partieel geparametriseerd met x; door de aanroep van *filter* worden díe elementen uit $[1 \mathinner{..} x]$ ge‘filter’d, waardoor x deelbaar is.

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

$$\begin{aligned} &\mathit{priem} && :: \mathit{Int} \to \mathit{Bool} \\ &\mathit{priem}\ x && = (\mathit{delers}\ x) \equiv [1, x] \end{aligned}$$

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

```
priemgetallen    :: Int → [Int]
priemgetallen x = filter priem [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 deze 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 = weekdag (dagnummer d m j)
weekdag 0 = "zondag"
weekdag 1 = "maandag"
weekdag 2 = "dinsdag"
weekdag 3 = "woensdag"
weekdag 4 = "donderdag"
weekdag 5 = "vrijdag"
weekdag 6 = "zaterdag"
```

De functie *weekdag* gebruikt zeven patronen om de juiste tekst te selecteren (een woord tussen
blz. 75 aanhalingstekens (*λkwoot*) is een tekst; voor de details zie paragraaf 6).

De functie *dagnummer* 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 *dagnummer* is dan namelijk simpeler: de 1e januari van het jaar 0 zou op een zondag zijn gevallen.

```
dagnummer d m j = (   (j − 1) ∗ 365
                    + (j − 1) / 4
                    − (j − 1) / 100
                    + (j − 1) / 400
                    + sum (take (m − 1) (maanden 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:

$$\begin{aligned} &take\ 0\ xs &&= [\,] \\ &take\ (n+1)\ (x : xs) &&= x : take\ n\ xs \end{aligned}$$

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

$$\begin{aligned} &maanden\ j = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] \\ &\quad \textbf{where} \quad feb \mid schrikkel\ j = 29 \\ &\qquad\qquad\quad \mid otherwise\ = 28 \end{aligned}$$

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

$$schrikkel\ j = deelbaar\ j\ 4 \wedge (\neg\ (deelbaar\ j\ 100) \vee deelbaar\ j\ 400)$$

Een andere manier om dit te definiëren is:

$$\begin{aligned} &schrikkel\ j \mid deelbaar\ j\ 100 = deelbaar\ j\ 400 \\ &\qquad\qquad\quad \mid otherwise\ = \quad deelbaar\ j\ 4 \end{aligned}$$

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:

$$dag\ d\ m\ j \mid j > 1752 = weekdag\ (dagnummer\ d\ m\ j)$$

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 *weekdag* en *dagnummer* nodig. Voor *dagnummer* was een functie *maanden* nodig, en voor *maanden* 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 *deelbaar* geschreven, met behulp daarvan een functie *delers*, daarmee een functie *priem*, en tenslotte de gevraagde *priemgetallen*.

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).

Opgaven

3.1 Verklaar de plaatsing van de haakjes in de expressie $(f\ (x+h) - f\ x)\ /\ h$.

3.2 Welke haakjes zijn overbodig in de volgende expressies?

- $(plus\ 3)\ (plus\ 4\ 5)$
- $sqrt\ (3.0) + (sqrt\ 4.0)$
- $(+)\ (3)\ (4)$
- $(2 * 3) + (4 * 5)$
- $(2 + 3) * (4 + 5)$
- $(a \to b) \to (c \to d)$

3.3 Waarom is het in de wiskunde gebruikelijk om machtsverheffen naar rechts te laten associëren?

3.4 Is de operator *na* ($\circ$) associatief?

3.5 In de taal Pascal heeft $\wedge$ dezelfde prioriteit als $*$, en $||$ dezelfde prioriteit als $+$. Waarom is dat niet handig?

3.6 Geef een voorbeeld van een functie met de volgende types:

- $(Float \to Float) \to Float$
- $Float \to (Float \to Float)$
- $(Float \to Float) \to (Float \to Float)$

blz. 45

3.7 In paragraaf 3 wordt beweerd dat *na* ook beschouwd kan worden als functie met één parameter. Hoe kun je dat zien aan het type? Geef een definitie van *na* in de vorm *na* $y = \ldots$

3.8 Bij functiecompositie werd eerst het tweede argument van $\circ$ op het argument losgelaten en daarna het eerste. Schrijf een functie *endan* zodat je i.p.v. f ‘*na*‘ g ‘*na*‘ h kan schrijven h ‘*endan*‘ g ‘*endan*‘ f. Hoe denk je dat het met de associativiteit van $\circ$ en ‘*endan*‘ zit? Een mooie notatie voor ‘*endan*‘ zou ; zijn, maar helaas is dat een gereserveerd symbool in Haskell

3.9 Schrijf een functie die bepaalt hoeveel jaar een bedrag op de bank moet staan om, bij een gegeven rente, een gegeven eindkapitaal te kunnen incasseren.

3.10 Waarom kunnen de functies *wortel* en *derdemachtswortel* wel worden geschreven zonder gebruik te maken van de functie *diff*, maar is het niet mogelijk om zo een algemene functie *inverse* te schrijven?

3.11 Schrijf een functie *integraal*, die de integraal van een functie tussen twee grenzen berekent. De functie werkt door het integratiegebied in een (te specificeren) aantal gebiedjes te verdelen, en op elk gebiedje de functie te benaderen door een lineaire functie. In welke volgorde kunnen de parameters het beste worden meegegeven, om de functie handig partieel te kunnen parameteriseren?

3.12 We hebben gezien dat $[\ldots]$ een constructie is die types op types afbeeldt. Zo is $\to$ een type constructor die twee types op een type afbeeldt, en dus $c \to$ een type functie die een type a op het type $c \to a$ afbeeldt. Hoe ziet het type van *map* voor deze type constructor er uit, en kun je –louter uitgaand van het type van deze *map*– beredeneren hoe map nu gedefinieerd is?

Hoofdstuk 4

Case Study: Some Numeric Functions

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
1.6666667
```

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 $\rightarrow$ 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:

$$\begin{aligned} &\mathit{diff}\ f\ x = (f\ (x+h) - f\ x)\ /\ h \\ &\quad \textbf{where}\ \ h = 0.0001 \end{aligned}$$

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 functie-transformatie.

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

$$\mathit{afgeleide\_van\_sinus\_kwadraat} = \mathit{diff}\ (\mathit{kwadraat} \circ \mathit{sin})$$

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:

$$\mathit{flexDiff}\ h\ f\ x = (f\ (x+h) - f\ x)\ /\ h$$

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

$$\begin{aligned} \mathit{grofDiff} &= \mathit{flexDiff}\ 0.01 \\ \mathit{fijnDiff} &= \mathit{flexDiff}\ 0.0001 \\ \mathit{superDiff} &= \mathit{flexDiff}\ 0.000001 \end{aligned}$$

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 wortel-functie kunt maken, als deze niet ingebouwd zou zijn. Het demonstreert een techniek die veel gebruikt wordt bij het
blz. 54
rekenen met *Float* getallen. De functie wordt in paragraaf 4 gegeneraliseerd naar inverses van andere functies dan de kwadraat-functie. Daar 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 gebruikt worden: blz. 44

```
wortel x = until goedGenoeg verbeter 1.0
  where verbeter    y = 0.5 * (y + x / y)
        goedGenoeg  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
```

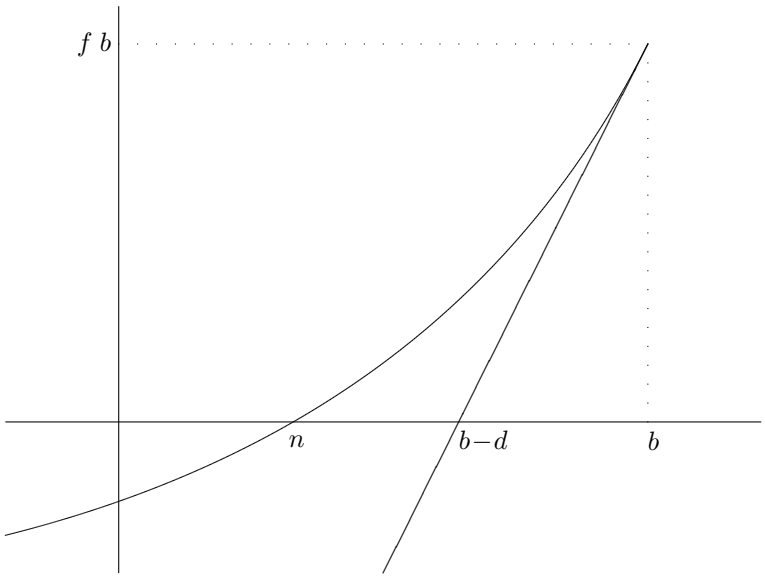
De hogere-orde functie *until* werkt op de *functies verbeter* en *goedGenoeg* en op de startwaarde 1.0. Hoewel *verbeter* naast 1.0 staat, wordt de functie *verbeter* dus niet onmiddellijk op 1.0 toegepast; in plaats daarvan worden beiden aan *until* meegegeven. Door het Currying-mechanisme is het immers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

Iets anders dat opvalt aan de definitie van *verbeter* en *goedGenoeg* 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). blz. 22

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 verbeter-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 *wortel* kan de Newton's verbeter-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.

```
nulpunt f y0 = until goedGenoeg verbeter y0
  where        verbeter     b = b - f b / diff f b
               goedGenoeg b = f b ~= 0.0
```

De eerste benadering die gebruikt kan worden, is als extra parameter aan de functie meegegeven.
blz. 51 De differentieer-functie uit paragraaf 4 komt ook goed van pas.

Inverse van een functie

Het nulpunt van de functie f met $f\ x = x^2 - a$ is $\sqrt{a}$. De wortel van a kan dus bepaald worden door het nulpunt van f te zoeken. Nu de functie *nulpunt* beschikbaar is, kan *wortel* dus ook zo geschreven worden:

```
wortel a = nulpunt f 1.0
  where  f x = x * x - a
```

Ook de derdemachtswortel kan op deze manier berekend worden:

```
derdemachtswortel a = nulpunt f 1.0
  where               f x = x * x * x - a
```

In feite kan de inverse van elke gewenste functie bepaald worden, door deze functie te gebruiken in de definitie van f, bijvoorbeeld:

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

Er begint zich een patroon af te tekenen in al deze definities. Dat is altijd een signaal om een
blz. 44 hogere-orde functie te definiëren, die de generalisatie ervan is (zie paragraaf 3, waar *foldr* werd gedefinieerd als generalisatie van *sum*, *product* en *and*). De hogere-orde functie is in dit geval *inverse*, die als extra parameter een functie g meekrijgt waarvan de inverse berekend moet worden:

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

Als je het patroon eenmaal ziet, is zo'n hogere-orde functie niet moeilijker meer dan de andere definities. Die andere definities zijn speciale gevallen van de hogere-orde functie, en kunnen nu ook geschreven worden als partiële parametrisatie:

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

De functie *inverse* kan naar believen gebruikt worden als functie met twee parameters (een functie en een *Float*) en *Float* resultaat, of als functie met één parameter (een functie) en een functie als resultaat. Het type van *inverse* is namelijk

$$inverse :: (Float \to Float) \to Float \to Float$$

wat ook geschreven kan worden als

$$inverse :: (Float \to Float) \to (Float \to Float)$$

omdat $\to$ naar rechts associeert.

De wortel-functie uit paragraaf 4 maakt in feite ook gebruik van de Newton-methode. Dit blijkt door in de definitie van *wortel* hierboven: blz. 52

$$\begin{array}{l} wortel\ a = nulpunt\ f\ 1.0 \\ \quad \textbf{where}\quad f\ x = x * x - a \end{array}$$

de aanroep $nulpunt\ f\ 1.0$ te vervangen door de definitie daarvan:

$$\begin{array}{lll} wortel\ a = until\ goedGenoeg\ verbeter\ 1.0 & & \\ \quad \textbf{where} & verbeter & b = b - f\ b\ /\ diff\ f\ b \\ & goedGenoeg & b = f\ b \mathrel{\tilde{=}} 0.0 \\ & f & x = x * x - a \end{array}$$

In dit specifieke geval hoef je *diff f* niet numeriek uit te rekenen: de afgeleide van de hier gebruikte f is immers de functie $(2*)$. De formule voor *verbeter b* is dus te vereenvoudigen:

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

Dit is precies de verbeter-formule die in paragraaf 4 werd gebruikt. blz. 52

Opgaven

4.1 Geef een definitie van *derdemachtswortel* die geen gebruik maakt van numeriek differentiëren (ook niet indirect via *nulpunt*).

4.2 Definiëer de functie *inverse* uit paragraaf 4 met gebruikmaking van de lambda-notatie. blz. 54

Hoofdstuk 5

IO in Haskell

Inleiding

Alle programma's die we tot dusverre hebben gezien bestaan uit een aantal functie definities; in de interpreter kan je dan die functies toepassen op geschikte argumenten. Het resultaat is een waarde die door de interpreter wordt afgedrukt. De functies zelf doen geen in- en uitvoer. We kunnen dus halverwege het uitrekenen van een waarde dus niet even tussendoor aan de gebruiker vragen nog een waarde in te tikken, of alvast een tussenresultaat laten zien. In dit hoofdstuk laten we zien hoe je in Haskell functies kunt schrijven die zelf uitvoer produceren en om invoer kunnen vragen. In een later hoofdstuk komen we terug op het bouwen van grafische user interfaces.

Op het eerste gezicht lijken Haskell en het plegen van in- en uitvoeracties niet goed samen te gaan. Haskell is een zogenaamde *puur* functionele taal, dat wil zeggen dat het resultaat van functies alleen maar afhangt van de waarden van de argumenten. Het resultaat kan dus niet afhangen van een eventuele object-toestand (zoals in Java) of globale variabelen, of van waardes in de externe wereld, zoals de inhoud van files. Dit betekent onder andere dat de volgorde van berekening niet uitmaakt voor het uiteindelijke resultaat[1] Het is niet moeilijk om je te realiseren dat bij in- en uitvoer de volgorde van berekenen er wel degelijk toe doet.

Een voorbeeld van de problemen die voorkomen worden door deze benadering laat zich aan de hand van het volgende voorbeeld uiteenzetten.

Omdat we altijd graag gebruik willen kunnen maken van abstracties is er bij het ontwerp van Haskell van een aantal strenge eisen uitgegaan. Als we een let-gebonden variable x in een expressie zoals

> **let** $x = <expr>$
> **in** $\ldots x \ldots x \ldots$

vervangen door zijn definitie dan krijgen we

> $\ldots <expr> \ldots <expr> \ldots$

en deze expressie dient precies dezelfde waarde op te leveren. Zo moet ook:

> $(\lambda x \rightarrow <expr1>) <expr2>$

altijd gelijk zijn aan:

> **let** $x = <expr2>$ **in** $<expr1>$

[1] het kan wel vesrchil uitmaken voor het al dan niet termineren van een aanroep

Stel nu dat we in Haskell een functie *random* zouden hebben die telkens een andere waarde oplevert –en dus onder water gebruik maakt van iets van toestand, om er voor te zorgen dat niet telkens dezelfde waarde wordt opgeleverd– dan zouden dus:

$$\textbf{let } x = \mathit{random} \textbf{ in } x \equiv x$$

en

$$\mathit{random} \equiv \mathit{random}$$

dezelfde waarde op moeten leveren. Dit leidt tot een tegenspraak. Van de eerste expressie zullen we i.h.a. verwachten dat hij altijd *True* oplevert, terwijl we van de tweede verwachten dat hij i..h.a. *False* zal opleveren. Het *principle of substitution*, d.w.z. dat je een variabele altijd door zijn definitie kan vervangen, gaat hier dus niet langer op.

Acties

Bij het ontwerp van Haskell is ervoor gekozen om expressies in twee groepen te verdelen: acties die normale waarden opleveren waarden en expressies die *acties* opleveren. De eerste groep bevat alle normale expressies die geen effect hebben op de buitenwereld en dus in willekeurige volgorde uitgerekend kunnen worden. De tweede groep, de acties, bevat expressies die wèl een effect hebben op de buitenwereld (zoals in- en uitvoer). Op het niveau van types kun je het verschil zien tussen de twee groepen. Een expressie met het type $IO\ a$ is een actie die in- en uitvoer kan doen (IO staat voor input en output), en uiteindelijk een waarde van type a beschikbaar stelt aan de rest van de berekening. Expressies van een willekeurige ander type – dus zonderIO aan het begin – zijn gewone waarden. Op top niveau is een Haskell programma meestal een functie die iets van type $IO \ldots$ oplevert. De betekenis hiervan is dat de opgeleverde rij acties uitgevoerd wordt. Essentieel hierbij is dat we spreken van een rij acties, en niet van een verzameling. De volgorde waarin uit kleine acties samengestelde acties worden opgebouwd, is in het programma nauwkeurig gespecificeerd (middels een handig gekozen syntax).

Lezen en schrijven van een enkel karakter

Een voorbeeld van een actie is het inlezen van een letter van het toetsenbord:

$$\mathit{getChar} :: IO\ Char$$

Aan het type zie je dat *getChar* een actie is die als resultaat een waarde van type *Char*, ofwel een letter, beschikbaar stelt aan de rest van de berekening. Merk op dat het type $IO\ Char$ iets anders is dan het type *Char*. We kunnen op een plaats waar we een waarde van type *Char* willen hebben dus niet zomaar *getChar* schrijven. Immers, daarmee verliezen we de mogelijkheid de totale verzameling acties die door ons programma wordt berekend, zorgvuldig te ordenen.

Het omgekeerde, een letter naar het scherm schrijven, gebeurt met de functie *putChar*. Deze functie krijgt als parameter een letter mee en levert dan een actie op die de letter op het scherm zal tonen. Dan is nog de vraag wat het resultaat is van die actie: iedere actie heeft als type iets van de vorm $IO\ a$, en wat is nu de a in dit geval? De actie levert echter niets op, we zijn alleen maar geïnteresseerd in het *effect* van de actie. In Java hebben we het in zo'n geval over `void` methoden. In Haskell is er het *unit* type dat voor dit doel gebruikt kan worden. De notatie is () en het type heeft één waarde die ook als () wordt genoteerd. Je zou het als een "nul-tupel" kunnen beschouwen. We hebben nu alle ingrediënten voor het type van *putChar*:

$$\mathit{putChar} :: \mathit{Char} \rightarrow IO\ ()$$

Deze functie kunnen we proberen in de interpreter:

```
? putChar 'K'
K?
```

Merk op dat de actie uitgevoerd wordt en dat het resultaat van de actie () niet getoond wordt. Hoe we iets met resultaten van acties kunnen doen zien we later.

Combineren van acties

Eén keer een letter op het scherm zetten is natuurlijk niet zo interessant. Het wordt iets leuker als we meerdere letters kunnen afdrukken; daarvoor is het nodig om acties te combineren. Zoals gezegd is de volgorde van uitvoeren belang bij acties en dit komt tot uiting in een speciale notatie:

```
tweeLetters :: IO ()
tweeLetters = do putChar 'H'
                 putChar 'i'
```

De zogenaamde **do**-notatie bestaat uit het gereserveerde woord **do**, gevolgd door één of meerdere acties. Merk op dat de layout van de acties er toe doet (net als bij **where**, zie 2): even ver inspringen betekent een nieuwe actie, verder inspringen betekent dat de regel nog bij de vorige hoort en minder ver inspringen beeindigt de notatie[2]. Het type van de hele **do**-expressie is het type van de laatste actie (hier *putChar* `'i'`) en is dus in dit geval *IO* ().

Recursieve acties

Een functie met een *IO* type mag, net als iedere andere functie, recursief zijn. Dit maakt het mogelijk om niet alleen maar twee, maar een hele lijst van letters af te drukken:

```
putStr :: String → IO ()
putStr (c : cs) = do putChar c
                     putStr  cs
```

Deze functie *putStr* drukt de eerste letter van de lijst af en gaat dan recursief verder met de rest. De oplettende lezer ziet dat deze functie-definitie nog niet compleet is omdat het basisgeval, de lege lijst, niet behandeld wordt. In het geval van een lege lijst willen we niets doen, maar dat 'niets' moet wel van het juiste type zijn: *IO* (). Er is gelukkig een voorgedefinieerde functie die we hiervoor kunnen gebruiken:

```
return :: a → IO a
```

Gegeven een waarde construeert *return* een actie die niets doet (geen zij-effecten heeft) en de gegeven parameter doorgeeft aan de rest van de berekening. In *putStr* willen we een unit waarde opleveren dus dat is de parameter van *return*. De complete definitie luidt nu:

```
putStr :: String → IO ()
putStr (c : cs) = do putChar c
                     putStr  cs
putStr []       = return ()
```

De naam *return* kan verwarring opleveren bij mensen die gewend zijn aan Java of C(++) omdat in die talen dat gereserveerde woord betekent dat niet alleen de parameter opgeleverd wordt maar ook dat de functie/methode beëindigd wordt. In Haskell is dat niet het geval; *return x* is een actie die *x*doorgeeft en verder geen zij-effect heeft. Hier is een vreemd gebruik van *return*:

[2] De preciese layout regels in Haskell zijn noal gecompliceerd; kijk eens in de taaldefinitie om uit te vinden hoe dit precies zit

```
tweeLetters' :: IO ()
tweeLetters' = do putChar 'H'
                  return   "doet helemaal niets"
                  putChar 'i'
```

Deze functie heeft hetzelfde effect als *tweeLetters* omdat de *return* nu eenmaal met de lege actie overeenkomt, en de waarde die doorgegevn wordt aan de rest van de berekening wordt genegeerd. Vrijwel altijd zie je *return* optreden als laatste actie in de **do**-notatie, en dan is dus het berekenen van de rij acties voltooid.

De functie *putStr* hoef je niet zelf te definiëren, want ook deze is al opgenomen in de prelude. Hetzelfde geldt voor *putStrLn* die ook een *String* afdrukt maar aan het eind ook nog een regelovergang toevoegt.

Acties met resultaten

We zagen eerder de functie *getChar*, een actie die een letter inleest. Zo bestaat er ook een functie die een hele regel inleest van het toetsenbord:

```
getLine :: IO String
```

Als we deze acties uitvoeren is er ook een resultaat waarin we geïnteresseerd zijn. Binnen de **do**-notatie kunnen we aan dat resultaat komen door het te binden aan een variabele. Dat ziet er zo uit:

```
x ← getLine
```

De actie *getLine* zal uitgevoerd worden en het resultaat (een regel tekst) zal aan de variable x gebonden worden. In verdere acties kunnen we die x dan gebruiken. Hier is een eenvoudig programmaatje dat hier gebruik van maakt:

```
groet :: IO ()
groet = do putStr "Wat is je naam? "
           naam ← getLine
           putStrLn ("Hallo, " ++ naam)
```

Het ziet er dus uit als een *toekenningsopdracht* in Java, maar dan met een pijl die naar links wijst in plaats van een =-teken.

Het hoofdprogramma

De voorbeelden van IO tot nu toe hebben nog steeds de vorm van functiedefinities welke je in de interpreter kunt aanroepen. Dit is over het algemeen niet wat we onder een *programma* verstaan. Een programma kun je starten en begint dan op een specifieke plek met de uitvoering. In een Java applicatie is dat de *main* methode. In Haskell bestaat dat ook maar in dan gaat het natuurlijk om een functie en niet om een methode. De naam is wel hetzelfde, *main*. Van de *groet*-functie kunnen we nu een programma maken:

```
main :: IO ()
main = do putStr "Wat is je naam? "
          naam ← getLine
          putStrLn ("Hallo, " ++ naam)
```

Ook dit programma kunnen we inlezen in de interpreter en dan *main* evalueren. We kunnen er ook een echt programma van maken door het te compileren. Dat geeft ons een bestand met als extensie `lvm` en dat kun je uitvoeren met het programma `lvmrun`. Stel dat het programmaatje in het bestand `Groet.hs` staat:

```
$ helium Groet.hs
Compiling Groet.hs
Compilation successful
$ lvmrun Groet.lvm
Wat is je naam? Arjan
Hallo, Arjan
```

Dit is vergelijkbaar met de situatie in Java waar de compiler `javac` class-files genereert die je met het programma `java` kunt uitvoeren.

Een groter voorbeeld: getal raden

Het is tijd voor een wat groter programma. De computer moet een getal raden dat de gebruiker in gedachten neemt door herhaaldelijk vragen te stellen. Het getal moet tussen 1 en 100 liggen en de eerste gok zal in het midden liggen (50). Als de gebruiker aangeeft dat zijn/haar getal kleiner is dan zal 25 gegokt worden enzovoort. De functie *raad* is indirect recursief via *verwerkAntwoord*.

```
main :: IO ()
main = do
    putStrLn "Neem een getal tussen 1 en 100 (inclusief) in gedachten"
    raad 1 100

raad :: Int → Int → IO ()
raad onder boven = do
  putStrLn ("Is het " ++ show midden ++ "? (g = groter, k = kleiner, j = ja)")
  antwoord ← getLine
  verwerkAntwoord antwoord
  where
    verwerkAntwoord       :: String → IO ()
    verwerkAntwoord "g" = raad (midden + 1) boven
    verwerkAntwoord "k" = raad onder (midden − 1)
    verwerkAntwoord "j" = putStrLn "Geraden!"
    verwerkAntwoord _   = raad onder boven

    midden                :: Int
    midden                = (onder + boven) `div` 2
```

Imperatief programmeren voorbij

In- en uitvoer in een imperatieve taal als Java gaat op vergelijkbare wijze als met de **do**-notatie. De berekeningsvolgorde ligt al vast dus daarvoor hoef je geen speciale voorzieningen te treffen. Het lijkt erop dat we dit model hebben nagebouwd in Haskell, maar als we even verder kijken dan zien we dat het model met acties veel krachtiger is. Acties zijn namelijk ook waarden in Haskell en kunnen dus net zo behandeld worden als bijvoorbeeld getallen: ze kunnen opgeslagen worden in lijsten, als argument meegegeven worden en opgeleverd worden als functieresultaat: ze hebben alle burgerrechten (*first-class citizens* en zijn aan geen enkele beperking onderworpen.

Laten we nog eens kijken naar de functie *putStr*. Deze functie drukt iedere letter van een *String* af met *putChar*. Dat roept herinneringen op aan "doe iets voor ieder element van een lijst", *map* dus. Als we *map* loslaten op *putChar* en de *String* krijgen we... een lijst van acties! Deze acties willen vervolgens na elkaar uitvoeren. Hiervoor gebruiken we de standaardfunctie:

```
sequence_ :: [IO a] → IO ()
```

Deze functie combineert alle acties in de lijst tot één actie. Nu kunnen we *putStr* opnieuw schrijven maar dan zonder dat we zelf de recursie uitprogrammeren:

$$putStr\ cs = sequence\_\ (map\ putChar\ cs)$$

Of nog korter:

$$putStr = sequence\_ \circ map\ putChar$$

Kortom, we kunnen abstraheren over de 'lus' die langs alle letters loopt. Probeer dat maar eens in Java met een *while*!

Andere acties

We hebben alleen maar invoer van toetsenbord en uitvoer naar het scherm gezien, maar het mechanisme van acties is natuurlijk veel algemener dan dat. Ook het inlezen en wegschrijven van bestanden kan gebeuren als actie:

$$readFile :: String \to IO\ String$$
$$writeFile :: String \to String \to IO\ ()$$

Van beide functies is de eerste parameter een bestandsnaam: *readFile* leest dat bestand in en levert de inhoud op in de vorm van één grote *String*. De functie *writeFile* schrijft de tekst in de tweede argument in het bestand en heeft net als *putStr* geen resultaat.

Grafische user-interfaces kunnen op vergelijkbare wijze gemaakt worden. In plaats van over bestanden spreek je dan over windows, knoppen en events.

Opgaven

Opgaven

5.1 Schrijf de functie *getLine* in termen van *getChar*.

5.2 Breid het raad-spelletje uit met een test voor vals spelen. Als de gebruiker zegt dat het getal kleiner is dan 50 en daarna probeert om door steeds 'groter' in te voeren toch weer op 50 uit te komen dan speelt hij/zij vals.

5.3 Schrijf de functie *sequence_*.

5.4 Schijf en functie die om een filenaam vraagt, deze file inleest en opsplitst in een rij regels. Het karakter `'\n'` geeft hierbij een regelovergang aan. Druk als resultaat het aantal regels af.

5.5 Verander het vorige programma zo dat elke regel weer gesplitst wordt een een rij woorden, waarbij een aantal spaties (`' '`) de woorden scheidt. Druk nu behalve het aantal regels ook het aantal woorden af. Maak je niet druk om leestekens.

5.6 schrijf de functie *getLine*

Hoofdstuk 6

Lijsten

Lijsten

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 $[Int]$, $[Float]$, $[Int \to Float]$, $[[Int]]$ 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 lijst-opsommingen met hun type zijn:

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

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:

$$\begin{array}{ll} [1+2, 3*x, length\ [1,2]] & :: [Int] \\ [3<4, a \equiv 5, p \wedge q] & :: [Bool] \\ [diff\ \ sin, inverse\ cos] & :: [Float \to Float] \end{array}$$

De gebruikte functies moeten dan wel als resultaat het gewenste type hebben.

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

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

Een lijst met één element wordt ook wel een *singleton-lijst* genoemd. De lijst $[[1,2,3]]$ is ook een singleton-lijst: 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 singleton-lijst, 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 type-variabelen (zie blz. 31 paragraaf 2), dus het type van de lege lijst is $[\,a\,]$:

$[\,] :: [\,a\,]$

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:

sum $[\,]$	$[\,]$ is een lege lijst getallen
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
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.

$(:) :: a \to [\,a\,] \to [\,a\,]$

Als bijvoorbeeld xs de lijst $[3,4,5]$ is, dan is $1 : xs$ 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 (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 interval-notatie: 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 name-blz. 18 lijk één van symbolen-combinaties die in paragraaf 2 werden gereserveerd 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:

$$\begin{array}{l} \mathit{enumFromTo}\ x\ y \mid y < x = [\,] \\ \qquad\qquad\qquad\quad \mid \mathit{otherwise} = x : \mathit{enumFromTo}\ (x+1)\ y \end{array}$$

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 niet meer dan een aardigheidje die het gebruik van de taal iets makkelijker maakt; het zou geen groot gemis zijn als deze constructie niet mogelijk was, want dan kan altijd nog de functie *enumFromTo* gebruikt worden.

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, blz. 24 *som* en *length* in paragraaf 2, en *map*, *filter* en *foldr* in paragraaf 3. Hoewel dit allemaal stan- blz. 25 daardfuncties zijn die in de prelude worden gedefinieerd, en ze dus niet zelf gedefinieerd hoeven te blz. 44 worden, is het toch belangrijk om hun definitie te bekijken. Ten eerste omdat het goede voorbeelden zijn van definities van functies op lijsten, ten tweede omdat de definitie vaak de duidelijkste beschrijving geeft wat een standaardfunctie doet.

In deze paragraaf volgen nog meer definities van functies op lijsten. Veel van deze functies zijn recursief, dat wil zeggen dat ze, voor het patroon $x : xs$, zichzelf aanroepen met de (kleinere) parameter xs.

Lijsten samenvoegen

Twee lijsten van hetzelfde type kunnen worden samengevoegd tot één lijst met de operator $+\!\!+$. Deze operatie wordt ook wel *concatenatie* ('samen-ketening') 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 definitie luidt:

$$\begin{array}{l} (+\!\!+) \qquad\quad :: [a] \to [a] \to [a] \\ [\,] +\!\!+ \quad ys = ys \\ (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \end{array}$$

In de definitie wordt de linker parameter onderworpen aan patroon-analyse. 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 bijvoorbeeld

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

De definitie van *concat* is als volgt:

$$\begin{array}{l} \mathit{concat} \qquad\quad :: [[a]] \to [a] \\ \mathit{concat}\ [\,] \qquad = [\,] \\ \mathit{concat}\ (xs : xss) = xs +\!\!+ \mathit{concat}\ xss \end{array}$$

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 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 in de wandeling 'concatenatie' genoemd. (Vergelijk de situatie met de operator $\wedge$, 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 een aantal 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:

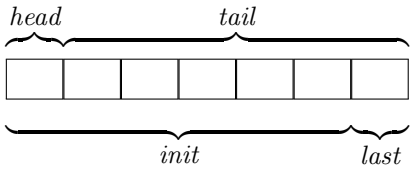
$$\begin{aligned} &\mathit{head} && :: [a] \to a \\ &\mathit{head}\ (x : xs) && = x \\ &\mathit{tail} && :: [a] \to [a] \\ &\mathit{tail}\ (x : xs) && = xs \end{aligned}$$

Deze functies doen patroon-analyse 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 foutmelding.

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

$$\begin{aligned} &\mathit{last} && :: [a] \to a \\ &\mathit{last}\ (x : [\,]) && = x \\ &\mathit{last}\ (x : xs) && = \mathit{last}\ xs \end{aligned}$$

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:



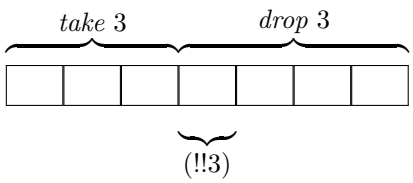
De functie *init* selecteert alles *behalve* het laatste element. Ook daarvoor is weer recursie nodig:

$$\begin{aligned} &\mathit{init} && :: [a] \to [a] \\ &\mathit{init}\ (x : [\,]) && = [\,] \\ &\mathit{init}\ (x : xs) && = x : \mathit{init}\ xs \end{aligned}$$

Het patroon $x : [\,]$ kan worden (en wordt meestal) geschreven als $[x]$.

blz. 49

In paragraaf 3 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:



Deze functies zijn als volgt gedefinieerd:

$$
\begin{array}{ll}
take, drop & :: Int \to [a] \to [a] \\
take\ 0\ xs & = [\,] \\
take\ n\ [\,] & = [\,] \\
take\ n\ (x : xs) & = x : take\ (n-1)\ xs \\
drop\ 0\ xs & = xs \\
drop\ n\ [\,] & = [\,] \\
drop\ n\ (x : xs) & = drop\ (n-1)\ xs
\end{array}
$$

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:

$$
\begin{array}{ll}
\textbf{infixl}\ 9\ !! & \\
(!!) & :: [a] \to Int \to a \\
(x : xs)\ !!\ 0 & = x \\
(x : xs)\ !!\ n & = xs\ !!\ (n-1)
\end{array}
$$

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 had bijvoorbeeld zo blz. 48 gedefinieerd kunnen worden:

$$
\begin{array}{l}
weekdag\ d = [\texttt{"zondag"}, \texttt{"maandag"}, \texttt{"dinsdag"}, \texttt{"woensdag"}, \\
\qquad\qquad \texttt{"donderdag"}, \texttt{"vrijdag"}, \texttt{"zaterdag"}]\ !!\ d
\end{array}
$$

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:

$$
\begin{array}{ll}
reverse\ [\,] & = [\,] \\
reverse\ (x : xs) & = reverse\ xs +\!\!+ [x]
\end{array}
$$

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, die we gebruiken om het uitendelijke resultaat in op te bouwen:

$$
\begin{array}{l}
reverse\ x = reverseaccum\ x\ [\,] \\
\quad \textbf{where}\ reverseaccum\ (x : xs)\ result = reverseaccum\ xs\ (x : result) \\
\qquad\qquad reverseaccum\ [\,]\ \ \ result = result
\end{array}
$$

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:

$$\begin{array}{l} length :: \quad [a] \to Int \\ length\ [\,] \quad = 0 \\ length\ (x : xs) = 1 + length\ xs \end{array}$$

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:

$$\begin{array}{l} elem :: \quad Eq\ a \Rightarrow a \to [a] \to Bool \\ elem\ e\ xs = or\ (map\ (\equiv e)\ xs) \end{array}$$

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

$$elem\ e = or \circ (map\ (\equiv e))$$

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

$$notElem\ e\ xs = \neg\ (elem\ e\ xs)$$

Deze functie kan ook gedefinieerd worden met

$$notElem\ e = and \circ (map\ (\not\equiv e))$$

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 functie-parameter, iets met alle elementen van een lijst. De functie *map* past zijn functie-parameter toe op alle elementen van de lijst:

$$\begin{array}{rcl} xs & = & [\ 1\ ,\ 2\ ,\ 3\ ,\ 4\ ,\ 5\ ] \\ & & \quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow \\ map\ kwadraat\ xs & = & [\ 1\ ,\ 4\ ,\ 9\ ,\ 16\ ,\ 25\ ] \end{array}$$

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

$$\begin{array}{rcl} xs & = & [\ 1\ ,\ 2\ ,\ 3\ ,\ 4\ ,\ 5\ ] \\ & & \quad\times\quad\downarrow\quad\times\quad\downarrow\quad\times \\ filter\ even\ xs & = & [\quad\quad 2\ ,\quad\quad 4\quad\quad] \end{array}$$

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

$$\begin{array}{rcl} xs & = & [\ 1\ ,\ 2\ ,\ 3\ ,\ 4\ ,\ 5\ ] \\ & & \quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow\quad\downarrow \\ foldr\ (+)\ 0\ xs & = & (1\ +\ (2\ +\ (3\ +\ (4\ +\ (5\ +\ 0\ ))))) \end{array}$$

Deze drie standaardfuncties worden in de prelude recursief gedefinieerd. Ze werden eerder besproblz. 44 ken in paragraaf 3.

$$
\begin{array}{ll}
map & :: (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = [\,] \\
map\ f\ (x : xs) & = f\ x : map\ f\ xs \\
filter & :: (a \to Bool) \to [a] \to [a] \\
filter\ p\ [\,] & = [\,] \\
filter\ p\ (x : xs) & \\
\quad \mid p\ x & = x : filter\ p\ xs \\
\quad \mid otherwise = & \quad filter\ p\ xs \\
foldr & :: (a \to b \to b) \to b \to [a] \to b \\
foldr\ op\ e\ [\,] & = e \\
foldr\ op\ e\ (x : xs) & = x\ `op`\ foldr\ op\ e\ xs
\end{array}
$$

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:

$$or = foldr\ (\vee)\ False$$

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

$$
\begin{array}{ll}
or\ [\,] & = False \\
or\ (x : xs) & = x \vee or\ xs
\end{array}
$$

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:

$$elem\ e = or \circ map\ (\equiv e)$$

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

$$
\begin{array}{ll}
elem\ e\ [\,] & = False \\
elem\ e\ (x : xs) & = x \equiv e \vee elem\ e\ xs
\end{array}
$$

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:

$$
\begin{array}{ll}
takeWhile & :: (a \to Bool) \to [a] \to [a] \\
takeWhile\ p\ [\,] & = [\,] \\
takeWhile\ p\ (x : xs) & \\
\quad \mid p\ x & = x : takeWhile\ p\ xs \\
\quad \mid otherwise & = [\,]
\end{array}
$$

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:

$$
\begin{array}{l}
dropWhile \quad :: (a \to Bool) \to [a] \to [a] \\
dropWhile\ p\ [\,] = [\,] \\
dropWhile\ p\ (x : xs) \\
\quad \mid p\ x \quad\quad = dropWhile\ p\ xs \\
\quad \mid otherwise \;= x : xs
\end{array}
$$

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:

$$
\begin{array}{rcccccccccccc}
xs & = & & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\
 & & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
foldl\ (+)\ 0\ xs & = & (((((& 0 & + & 1) & + & 2) & + & 3) & + & 4) & + & 5)
\end{array}
$$

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

$$
\begin{array}{l}
foldl\ (\oplus)\ a\ [x,y,z] = ((a\ \oplus\ x)\ \oplus\ y)\ \oplus\ z \\
foldl\ (\oplus)\ b\ [\quad y,z] = (\ b\quad\quad\ \oplus\ y)\ \oplus\ z
\end{array}
$$

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:

$$
\begin{array}{l}
foldl\ op\ e\ [\,] \quad\quad = e \\
foldl\ op\ e\ (x : xs) = foldl\ op\ (e\ `op`\ x)\ xs
\end{array}
$$

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 zogenaamd *tail-recursief*, d.w.z dat het resultaat gevormd wordt door een recursieve aanroep van de functie zelf. Een goede compiler herkent deze stituatie, en genereert hier efficiënte code voor (overeenkomed met een lus). 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.

Helaas kunnen we in Haskell (nog) niet aangeven dat operatoren bepaalde eigenschappen hebben, zoals associativiteit, zodat we hier de vertaler een beetje behulpzaam moeten zijn.

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:

$$
\begin{array}{l}
eqListInt :: \quad\quad [Int] \to [Int] \to Bool \\
eqListInt\ [\,]\ [\,] \quad\quad = True \\
eqListInt\ [\,]\ (y : ys) \quad = False \\
eqListInt\ (x : xs)\ [\,] \quad = False \\
eqListInt\ (x : xs)\ (y : ys) = x \equiv y \wedge eqListInt\ xs\ ys
\end{array}
$$

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 ($x \equiv y$), en wordt de operator recursief aangeroepen op de rest-lijsten (*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 $\equiv$ 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:

$$
\begin{aligned}
&\mathit{eqList} :: (a \to a \to \mathit{Bool}) \to [a] \to [a] \to \mathit{Bool} \\
&\mathit{eqList}\ \mathit{test}\ [\,]\ [\,] &&= \mathit{True} \\
&\mathit{eqList}\ \mathit{test}\ [\,]\ (y : \mathit{ys}) &&= \mathit{False} \\
&\mathit{eqList}\ \mathit{test}\ (x : \mathit{xs})\ [\,] &&= \mathit{False} \\
&\mathit{eqList}\ \mathit{test}\ (x : \mathit{xs})\ (y : \mathit{ys}) &&= \mathit{test}\ x\ y \wedge \mathit{eqList}\ \mathit{test}\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

Deze functie is in de prelude inderdaad gedefinieerd, en we kunnen er dus lijsten van elk gewenst type mee vergelijken:

```
? 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 normaal Haskell kunnen we gewoon $\equiv$ gebruiken tussen lijsten. Bij de behandeling van het klasse-systeem 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* ('woordenboek-volgorde'): 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 in Helium als volgt is gedefinieerd:

$$
\begin{array}{l}
ordInt :: Int \to Int \to Ordering \\
ordInt\ x\ y \\
\quad \mid x < y = LT \\
\quad \mid x \equiv y = EQ \\
\quad \mid otherwise = GT
\end{array}
$$

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 is in de prelude als volgt gedefinieerd:

$$
\begin{array}{l}
ordList :: (a \to a \to Ordering) \to [a] \to [a] \to Ordering \\
ordList\ test\ [\,]\ \ (y : ys) = LT \\
ordList\ test\ [\,]\ \ [\,]\ \ \ = EQ \\
ordList\ test\ (x : xs)\ [\,] = GT \\
ordList\ test\ (x : xs)\ (y : ys) = \\
\quad \textbf{case}\ test\ x\ y\ \textbf{of} \\
\quad\quad GT \to GT \\
\quad\quad LT \to LT \\
\quad\quad EQ \to ordList\ test\ xs\ ys
\end{array}
$$

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 te ordenen.

Lidmaatschapstest

Als we een functie hebben om elementen te vergelijken, dan kunnen we een functie maken die bepaalt of een element ergens in een lijst voorkomt. In de prelude is er zo'n functie, genaamd *elemBy* met drie parameters: een vergelijkingstest-functie, een element, en een lijst. De definitie luidt:

$$
\begin{array}{l}
elemBy :: (a \to a \to Bool) \to a \to [a] \to Bool \\
elemBy\ test\ x\ [\,] \qquad\qquad\quad = False \\
elemBy\ test\ x\ (y : ys) \mid test\ x\ y \quad = True \\
\qquad\qquad\qquad\qquad\ \mid otherwise = elemBy\ test\ x\ ys
\end{array}
$$

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 leuker is om deze functie dan weer te herschrijven met gebruikmaking van de functie-samenstel-operator $\circ$:

$$elemBy\ test\ e = or \circ map\ (test\ e)$$

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 sorteer-functie te schrijven. Er zijn verschillende mogelijkheden om het sorteer-probleem aan te pakken. Deftiger gezegd: 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.

In de versie van Helium die geen gebruik maakt van classes, komt dit tot uiting in het feit dat de functie als eerste parameter een functie verwacht, die aangeeft hoe de elementen van de lijst geordend worden:

$$sorteer :: (a \to a \to Ordering) \to [a] \to [a]$$

In volledig Haskell wordt dit uitgedrukt door het type van de sorteerfunctie:

$$sorteer :: Ord\ a \Rightarrow [a] \to [a]$$

dat wil zeggen: *sorteer* werkt tussen lijsten van willekeurig type a, mits het type a in de klasse van ordenbare types *Ord* zit.

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 volgende functie: Wanneer we in helium de ordeningsrelatie expliciet meegeven:

$$\begin{array}{lll} insert & :: & (a \to a \to Ordering) \to a \to [a] \to [a] \\ insert\ cmp\ e\ [\,] & = & [e] \\ insert\ cmp\ e\ (x : xs) & = & \textbf{case}\ e\ `cmp`\ x\ \textbf{of} \\ & & \quad LT \to e : x : xs \\ & & \quad \_ \quad \to x : insert\ cmp\ e\ xs \end{array}$$

In volledig Haskell geven we aan dat we eisen dat er een ordeningsrelatie bestaat, en wordt deze op magische wijze meegegeven:

$$\begin{array}{lll} insert & :: & Ord\ a \Rightarrow a \to [a] \to [a] \\ insert\ e\ [\,] & = & [e] \\ insert\ e\ (x : xs) & & \\ \quad \mid e \leqslant x & = & e : x : xs \\ \quad \mid otherwise & = & x : insert\ e\ xs \end{array}$$

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*: Helium:

```
? insert ordInt 5 [2,4,6,8,10]
[2, 4, 5, 6, 8, 10]
```

Haskell:

```
? insert 5 [2,4,6,8,10]
[2, 4, 5, 6, 8, 10]
```

Voor de werking van *insert* is het essentieel dat de parameter-lijst 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:

$$a \text{ `}insert\text{` } (b \text{ `}insert\text{` } (c \text{ `}insert\text{` } (d \text{ `}insert\text{` } [\,])))$$

De structuur van deze berekening is precies die van *foldr*, met *insert* als operator en $[\,]$ als startwaarde. Een mogelijk sorteer-algoritme luidt dus: Helium:

$$isort\ cmp = foldr\ (insert\ cmp)\ [\,]$$

Haskell:

$$isort = foldr\ insert\ [\,]$$

met de functies *insert* zoals hierboven gedefinieerd. Dit algoritme wordt *insertion sort* genoemd.

Sorteren door samenvoegen

Een ander sorteer-algoritme maakt gebruik van de mogelijkheid om twee gesorteerde lijsten samen te voegen tot één. Daartoe dient de functie *merge*: Helium:

$$\begin{array}{l}
merge :: (a \to a \to Ordering) \to [a] \to [a] \to [a] \\
merge\ cmp\ [\,]\ ys \qquad\qquad = ys \\
merge\ cmp\ xs\ [\,] \qquad\qquad = xs \\
merge\ cmp\ (x : xs)\ (y : ys) = \textbf{case}\ x \text{ `}cmp\text{` } y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad LT \to x : merge\ cmp \qquad xs\ (y : ys) \\
\qquad\qquad\qquad\qquad\qquad\quad _ \to \quad y : merge\ cmp\ (x : xs) \qquad ys
\end{array}$$

Haskell:

$$\begin{array}{l}
merge :: \qquad Ord\ a \Rightarrow [a] \to [a] \to [a] \\
merge\ [\,]\ ys \quad = ys \\
merge\ xs\ [\,] \quad = xs \\
merge\ (x : xs)\ (y : ys) \\
\quad \mid x \leqslant y \qquad = x : merge \qquad xs\ (y : ys) \\
\quad \mid otherwise = y : merge\ (x : xs) \qquad ys
\end{array}$$

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 kop-elementen 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 sorteer-algoritme worden gebaseerd. Dit algoritme maakt er gebruik van dat de lege lijst en singleton-lijsten (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 sorteer-algoritme. De twee gesorteerde resultaten kunnen tenslotte worden samengevoegd door *merge*.

Helium:

```
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
```

Haskell:

```
msort ::            Ord a ⇒ [a] → [a]
msort xs
  | lengte ⩽ 1 = xs
  | otherwise  = merge (msort ys) (msort zs)
  where ...
```

Dit algoritme wordt *merge sort* genoemd. In de prelude worden de functies *insert* en *merge* gedefinieerd, en een functie *sort* die werkt zoals *isort*.

Speciale lijsten

Strings

In een voorbeeld in paragraaf 3 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. blz. 48

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 aanhalingstekens. De 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 aanhalingstekens geschreven worden. Ze worden alleen weggelaten door de interpreter in het eindresultaat van een opdracht:

```
? "zon" ++ "dag"
"zondag"
```

De elementen van een string zijn van het type *Char*. Dat is een afkorting van het woord *character*. Mogelijke characters zijn niet alleen de lettertekens, maar ook de cijfer-symbolen en de leestekens. Het type *Char* is één van de vier basis-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 volegnde 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.

Characters

De waarde van een *Char* kunnen lettertekens, cijfertekens en leestekens zijn. Het is belangrijk om de aanhalingstekens rond een character neer te zetten, omdat deze tekens in Haskell 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
$\circ$	$(b \to c) \to (a \to b) \to a \to c$	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.

Er is één leesteken dat in een string problemen geeft: het aanhalingsteken. Bij een aanhalingsteken in een string zou de string immers afgelopen zijn. Als er toch een aanhalingsteken in een string nodig is, moet daar het symbool `\` (een omgekeerde deelstreep of *backslash*) vóór gezet worden. 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 4. Ook mogen deze symbolen tussen enkele aanhalingstekens staan, zoals in onderstaande definities:

```
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 'tabulatie'. 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]. Er zijn twee (ingebouwde) standaardfuncties, die de code van een character bepalen, respectievelijk het character met een bepaalde code opleveren:

$ord :: Char \to Int$
$chr :: Int \to Char$

Bijvoorbeeld:

```
? ord 'A'
65
? chr 51
'3'
```

Een overzicht van alle characters met hun ISO/ASCII-codenummers staat in appendix A. De characters zijn geordend volgens deze codering. Het type *Char* maakt daarmee deel uit van de klasse *Ord*. 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: In Helium schrijven we (zonder impliciet gebruik te maken van classes):

```
? isort ordString ["aap", "noot", "Mies", "Wim"]
["Mies", "Wim", "aap", "noot"]
```

en in volledig Haskell volstaat:

```
? sort ["aap", "noot", "Mies", "Wim"]
["Mies", "Wim", "aap", "noot"]
```

Functies op characters en strings

In de prelude worden een aantal functies gedefinieerd op characters, waarmee bepaald kan worden wat voor soort teken een gegeven character is:

$isSpace, isUpper, isLower, isAlpha, isDigit, isAlphaNum :: Char \to Bool$
$isSpace\ c \quad = ord\ c \equiv ord\ \texttt{' '} \vee ord\ c \equiv ord\ \texttt{'\t'} \vee ord\ c \equiv ord\ \texttt{'\n'}$
$isUpper\ c \quad = ord\ c \geqslant ord\ \texttt{'A'} \wedge ord\ c \leqslant ord\ \texttt{'Z'}$
$isLower\ c \quad = ord\ c \geqslant ord\ \texttt{'a'} \wedge ord\ c \leqslant ord\ \texttt{'z'}$
$isAlpha\ c \quad = isUpper\ c \vee isLower\ c$
$isDigit\ c \quad = ord\ c \geqslant ord\ \texttt{'0'} \wedge ord\ c \leqslant ord\ \texttt{'9'}$
$isAlphaNum\ c = isAlpha\ c \vee isDigit\ c$

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

[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.

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 prelude 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
```

Merk hierbij op dat in het resultaat geen aanhalingstekens staan: deze worden altijd weggelaten als het resultaat van een expressie een string is.

Een variant op de functie *unlines* is de functie *layn*. Deze nummert de regels in het resultaat:

```
? layn ["eerste regel", "tweede regel"]
   1) eerste regel
   2) tweede regel
```

De precieze definitie van deze functies is op te zoeken in de prelude; het belangrijkste voor dit moment is ze te kunnen gebruiken in expressies, om een overzichtelijk resultaat te krijgen.

Oneindige lijsten

Het aantal elementen in een lijst kan oneindig groot zijn. De hier volgende functie *vanaf* levert een oneindig lange lijst op:

$$vanaf\ n = n : vanaf\ (n+1)$$

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, {\em control-C}
{Interrupted!}
?
```

Op het moment dat je genoeg elementen hebt gezien, kun je de berekening stoppen door op control-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 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.

Lazy evaluatie

De evaluatiemethode (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 ècht 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 niet mogelijk.

blz. 47

Lazy evaluatie heeft nog meer voordelen. Bekijk bijvoorbeeld de functie *priem* uit paragraaf 3, die kijkt of een getal een priemgetal is:

$$\begin{aligned} &priem :: \quad Int \to Bool \\ &priem\ x = delers\ x \equiv [1, x] \end{aligned}$$

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 $\equiv$

'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 $\equiv$ wordt veroorzaakt door zijn definitie. De definitie van gelijkheid op lijsten ziet er als volgt uit:

$$\begin{aligned} &(x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys \\ &[\,] \quad\ \equiv [\,] \quad\ = True \\ &[\,] \quad\ \equiv \_ \quad\ = False \\ &\_ \quad\ \equiv [\,] \quad\ = false \end{aligned}$$

Als $x \equiv y$

de waarde *False* oplevert, hoeft $xs \equiv ys$ niet meer uitgerekend te worden: het totale resultaat is toch altijd *False*. Dit lazy gedrag dankt de operator $\wedge$ op zijn beurt aan zijn definitie:

$$\begin{aligned} &False \wedge x = False \\ &True \wedge x\ = x \end{aligned}$$

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 $\wedge$. De definitie in paragraaf 2 is ook goed, maar vertoont niet het gewenste lazy gedrag). We zeggen dat $\wedge$ strict is in zijn eerste argument, maar niet in zijn tweede.

blz. 25

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 uiterte 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. . .).

Functies op oneindige lijsten

In de prelude worden een aantal functies gedefinieerd die oneindige lijsten opleveren.

De functie *vanaf* uit paragraaf 6 heet in werkelijkheid *enumFrom*. De functie wordt 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 6 werd besproken). blz. 79 blz. 64

Een oneindige lijst waarin steeds één element herhaald wordt, kan worden gemaakt met de functie *repeat*:

$$\begin{aligned} &\mathit{repeat} :: \quad a \to [a] \\ &\mathit{repeat}\ x = x : \mathit{repeat}\ x \end{aligned}$$

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:

$$\begin{aligned} &\mathit{replicate} :: \quad \mathit{Int} \to a \to [a] \\ &\mathit{replicate}\ n\ x = \mathit{take}\ n\ (\mathit{repeat}\ x) \end{aligned}$$

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:

$$\begin{array}{lll} \mathit{iterate}\ (+1)\ 3 & \text{is} & [3,4,5,6,7,8,\ldots \\ \mathit{iterate}\ (*2)\ 1 & \text{is} & [1,2,4,8,16,32,\ldots \\ \mathit{iterate}\ (/10)\ 5678 & \text{is} & [5678,567,56,5,0,0,\ldots \end{array}$$

De definitie van *iterate*, die in de prelude staat, is als volgt:

$$\begin{aligned} &\mathit{iterate} :: \quad (a \to a) \to a \to [a] \\ &\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

Deze functie lijkt een beetje op de functie *until*, die in paragraaf 3 werd gedefinieerd. 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... blz. 44

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:

$$
\begin{array}{r}
5678 \\
\downarrow \mathit{iterate}\ (/10) \\
[5678, 567, 56, 5, 0, 0, \ldots \\
\downarrow \mathit{takeWhile}\ (\not\equiv 0) \\
[5678, 567, 56, 5] \\
\downarrow \mathit{map}\ (`\mathit{rem}`10) \\
[8, 7, 6, 5] \\
\downarrow \mathit{reverse} \\
[5, 6, 7, 8] \\
\downarrow \mathit{map}\ \mathit{digitChar} \\
[\texttt{'5'}, \texttt{'6'}, \texttt{'7'}, \texttt{'8'}]
\end{array}
$$

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 ($\circ$) de betekenis 'na' heeft:

$$
\begin{array}{l}
\mathit{intString} :: \ \mathit{Int} \to [\mathit{Char}] \\
\mathit{intString} = \mathit{map}\ \mathit{digitChar} \\
\qquad\qquad \circ \mathit{reverse} \\
\qquad\qquad \circ \mathit{map}\ (`\mathit{rem}`10) \\
\qquad\qquad \circ \mathit{takeWhile}\ (\not\equiv 0) \\
\qquad\qquad \circ \mathit{iterate}\ (/10)
\end{array}
$$

Functioneel programmeren is programmeren met functies!

De lijst van alle priemgetallen

blz. 47

In paragraaf 3 werd een functie *priem* gedefinieerd, die bepaalt of een getal een priemgetal is. De (oneindige) lijst van alle priemgetallen kan daarmee worden berekend door

$$\mathit{filter}\ \mathit{priem}\ [2\,..]$$

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:

$$
\begin{array}{l}
\mathit{iterate}\ \mathit{streepweg}\ [2\,..] \\
\textbf{where}\ \mathit{streepweg}\ (x : xs) = \mathit{filter}\ (\neg \circ \mathit{veelvoud}\ x)\ xs \\
\quad \mathit{veelvoud}\ x\ y = \mathit{deelbaar}\ y\ x
\end{array}
$$

(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 super-lijst is als volgt opgebouwd:

$$
\begin{array}{l}
[[2,3,4,5,6,7,8,9,10,11,12,13,14,\ldots \\
,[3,5,7,9,11,13,15,17,19,21,23,25,27,\ldots \\
,[5,7,11,13,17,19,23,25,29,31,35,37,41,\ldots \\
,[7,11,13,17,19,23,29,31,37,41,43,47,49,\ldots \\
,[11,13,17,19,23,29,31,37,41,43,47,51,53,\ldots \\
,\ldots
\end{array}
$$

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 de complete lijst hoeft niet zichtbaar gemaakt te worden: de gewenste priemgetallen zijn steeds het eerste element van de lijst. De priemgetallen worden dus bepaald door van elke lijst de *head* te nemen:

$$
\begin{array}{ll}
\mathit{priemgetallen} :: & [\mathit{Int}] \\
\mathit{priemgetallen} = & \mathit{map}\ \mathit{head}\ (\mathit{iterate}\ \mathit{streepweg}\ [2\,..]) \\
\quad \textbf{where} & \mathit{streepweg}\ (x : xs) = \mathit{filter}\ (\neg \circ \mathit{veelvoud}\ x)\ xs
\end{array}
$$

Door de lazy evaluatie wordt van elke lijst in de super-lijst precies het gedeelte uitgerekend dat nodig is voor het gewenste deel van het antwoord. Wil je het volgende priemgetal weten, dan wordt 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 geoptimaliseerd.

Lijst-comprehensies

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 verzameling-comprehensie, is in Haskell een vergelijkbare notatie beschikbaar om lijsten te construeren. Deze notatie heet dan ook een *lijst-comprehensie*. Een eenvoudig voorbeeld van deze notatie is de volgende expressie:

$$[x * x \mid x \leftarrow [1\,..\,10]]$$

Deze expressie kan worden uitgesproken als 'x kwadraat voor x uit 1 tot 10'. In een lijst-comprehensie staat voor de verticale streep een expressie, waarin een variabele mag voorkomen. Deze variabele (x in het voorbeeld) wordt gebonden in het gedeelte achter de verticale streep. De notatie '$x \leftarrow xs$' 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

$$\mathit{map}\ \mathit{kwadraat}\ [1\,..\,10]$$

waarbij de functie *kwadraat* is gedefinieerd als

$$\mathit{kwadraat}\ x = x * x$$

Het voordeel van de comprehensie-notatie is dat de functie die steeds wordt uitgerekend (*kwadraat* in het voorbeeld) niet eerst een naam hoeft te krijgen.

De lijst-comprehensie notatie 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 resultaat-lijst worden dus alleen díe x verwerkt, waarvoor *even* x de waarde *True* heeft.

Door elk pijltje ($\leftarrow$) 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 lijst-comprehensie notatie overbodig. Hetzelfde effect kan bereikt worden door combinaties van *map*, *filter* en *concat*. De comprehensie-notatie is, zeker in ingewikkelde gevallen, echter veel gemakkelijker te begijpen. Bovenstaand voorbeeld zou anders geschreven moeten worden als

$$\begin{array}{l} \mathit{concat}\ (\mathit{map}\ f\ (\mathit{filter}\ \mathit{even}\ [1\,..\,5])) \\ \textbf{where}\ f\ x = \mathit{map}\ g\ [1\,..\,x] \\ \qquad\qquad \textbf{where}\ g\ y = (x, y) \end{array}$$

hetgeen veel minder inzichtelijk is.

Een lijst-comprehensie wordt door de interpreter direct vertaald naar een overeenkomstige expressie met *map*, *filter* en *concat*. Net als de notatie voor intervallen is de comprehensie-notatie dus puur bedoeld voor het gemak van de programmeur.

Tupels

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, 'a')`	een tupel met als elementen de integer 1 en het character `'a'`;
`("aap", True, 2)`	een tupel met drie elementen: de string `"aap"`, de boolean *True* en het getal 2;
$([1, 2], \mathit{sqrt})$	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:

$$
\begin{array}{ll}
(1, \texttt{'a'}) & :: (Int, Char) \\
(\texttt{"aap"}, True, 2) & :: ([Char], Bool, Int) \\
([1,2], sqrt) & :: ([Int], Float \to Float) \\
(1, (2,3)) & :: (Int, (Int, Int))
\end{array}
$$

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.

$$
\begin{array}{lll}
fst & :: (a,b) & \to a \\
fst & (x,y) & = x \\
snd & :: (a,b) & \to b \\
snd & (x,y) & = y \\
fst3 & :: (a,b,c) & \to a \\
fst3 & (x,y,z) & = x \\
snd3 & :: (a,b,c) & \to b \\
snd3 & (x,y,z) & = y \\
thd3 & :: (a,b,c) & \to c \\
thd3 & (x,y,z) & = z
\end{array}
$$

Deze functies zijn polymorf, maar het is natuurlijk ook mogelijk om functies te schrijven die maar op één specifiek tupel-type werken:

$$
\begin{array}{lll}
f & :: (Int, Char) & \to [Char] \\
f & (n, \quad c) & = intString\ n +\!\!+ [c]
\end{array}
$$

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 lijst-versie, *afstandT* de tupelversie hiervan:

$$
\begin{array}{lll}
afstandL & :: [Float] & \to Float \\
afstandL & [x,y] & = sqrt\ (x *.\ x +.\ y *.\ y) \\
afstandT & :: (Float, Float) & \to Float \\
afstandT & (x, \quad y \quad) & = sqrt\ (x *.\ x +.\ y *.\ y)
\end{array}
$$

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 dankzij het Currying-mechanisme; functies die meerdere

resultaten hebben, zijn echter alleen mogelijk 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:

$$\begin{aligned} &splitAt :: && Int \to [a] \to ([a],[a]) \\ &splitAt\ n\ xs = && (take\ n\ xs, drop\ n\ xs) \end{aligned}$$

Het werk van beide functies kan echter in één keer worden gedaan, vandaar dat *splitAt* uit efficiëntie-overwegingen als volgt is gedefinieerd:

$$\begin{aligned} &splitAt\ :: Int \to [a] \to ([a],[a]) \\ &splitAt\ 0\ xs = && ([],xs) \\ &splitAt\ n\ [] = && ([],[]) \\ &splitAt\ n\ (x:xs) = && (x:ys, zs) \\ &\qquad \textbf{where}\ (ys,zs) = splitAt\ (n-1)\ xs \end{aligned}$$

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 resultaat-tupel gebruikt kan worden: door het te onderwerpen aan een patroon-analyse ((ys, zs) in het voorbeeld).

Type-definities

Bij veelvuldig gebruik van lijsten en tupels worden type-declaraties vaak nogal ingewikkeld. Bijvoorbeeld bij het schrijven van functies op punten, zoals de functie *afstand* hierboven. De eenvoudigste functies zijn nog wel te overzien:

$$\begin{aligned} &afstand :: (Float, Float) \to Float \\ &verschil :: (Float, Float) \to (Float, Float) \to Float \end{aligned}$$

Maar lastiger wordt het bij lijsten van punten, en vooral bij hogere-orde functies:

$$\begin{aligned} &opp\_veelhoek :: && [(Float, Float)] \to Float \\ &transf\_veelhoek :: && ((Float, Float) \to (Float, Float)) \\ &&& \to [(Float, Float)] \to [(Float, Float)] \end{aligned}$$

In zo'n geval komt een *type-definitie* van pas. Met een type-definitie is het mogelijk om een (duidelijkere) naam te geven aan een type, bijvoorbeeld:

$$\textbf{type}\ Punt = (Float, Float)$$

Na deze type-definitie zijn de type-declaraties eenvoudiger te schrijven:

$$\begin{aligned} &afstand && :: Punt \to Float \\ &verschil && :: Punt \to Punt \to Float \\ &opp\_veelhoek && :: [Punt] \to Float \\ &transf\_veelhoek && :: (Punt \to Punt) \to [Punt] \to [Punt] \end{aligned}$$

Nog beter is het om ook voor 'veelhoek' een type-definitie te maken:

$$\begin{aligned} &\textbf{type}\ Veelhoek && = [Punt] \\ &opp\_veelhoek && :: Veelhoek \to Float \\ &transf\_veelhoek && :: (Punt \to Punt) \to Veelhoek \to Veelhoek \end{aligned}$$

Een paar dingen om in de gaten te houden bij type-definities:

- 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 type-*declaratie* specificeert het type van een functie; een type-*definitie* 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:

$$\begin{aligned}&\textbf{type } Punt &&= (Float, Float)\\ &\textbf{type } Complex &&= (Float, Float)\end{aligned}$$

dan mogen die namen door elkaar gebruikt worden. Een *Punt* is hetzelfde als een *Complex* is hetzelfde als een $(Float, Float)$. In paragraaf 8 wordt een methode beschreven hoe *Punt* als een echt *nieuw* type gedefinieerd kan worden. blz. 117

Rationale getallen

Een toepassing waarbij tupels goed gebruikt kunnen worden is een implementatie van de *rationale getallen*. De rationale getallen vormen de wiskundige verzameling **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 type-definitie ligt daarom voor de hand:

$$\textbf{type } Ratio = (Int, Int)$$

Een aantal veelgebruikte breuken kunnen een aparte naam krijgen:

$$\begin{aligned}qNul &= (0, 1)\\ qEen &= (1, 1)\\ qTwee &= (2, 1)\\ qHalf &= (1, 2)\\ qDerde &= (1, 3)\\ qKwart &= (1, 4)\end{aligned}$$

Het is de bedoeling om functies te schrijven die de belangrijkste rekenkundige operaties op rationale getallen uitvoeren:

$$\begin{aligned}qMaal &:: Ratio \to Ratio \to Ratio\\ qDeel &:: Ratio \to Ratio \to Ratio\\ qPlus &:: Ratio \to Ratio \to Ratio\\ qMin &:: Ratio \to Ratio \to Ratio\end{aligned}$$

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 min-teken 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
blz. 47 paragraaf 3:

```
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 ($=x - (x \mathbin{/} y) * y$) 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)
```

Tupels en lijsten

Tupels komen vaak voor als elementen van een lijst. Veel gebruikt wordt bijvoorbeeld een lijst van twee-tupels, die als opzoeklijst (woordenboek, telefoonboek enz.) kan dienen. De opzoek-functie 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)'.

$$
\begin{array}{l}
zoekOp :: Eq\ a \Rightarrow [(a,b)] \to a \to b \\
zoekOp\ ((x,y) : ts)\ z \\
\quad \mid x \equiv z \quad\ \ = y \\
\quad \mid otherwise = zoekOp\ ts\ z
\end{array}
$$

De functie is polymorf, dus werkt op lijsten 2-tupels van willekeurig type. Wel moeten de op te zoeken elementen vergeleken kunnen worden, dus het type a moet in de klasse *Eq* zitten.

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:

$$
\begin{array}{l}
telefoonNr = zoekOp\ telefoonboek \\
vertaling \quad = zoekOp\ woordenboek
\end{array}
$$

waarbij *telefoonboek* en *woordenboek* apart als constante gedefinieerd kunnen worden.

Een andere functie waarin lijsten 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 parameter-lijsten niet even lang zijn, is de lengte van de kortste van de twee bepalend. De definitie is zeer rechtstreeks:

$$
\begin{array}{lllll}
zip :: [a] \to [b] & & & \to [(a,b)] \\
zip & [\,] & ys & = [\,] \\
zip & xs & [\,] & = [\,] \\
zip & (x : xs) & (y : ys) & = (x,y) : zip\ xs\ ys
\end{array}
$$

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:

$$
\begin{array}{lllll}
zipWith :: (a \to b \to c) \to [a] \to & [b] & & \to [c] \\
zipWith & f & [\,] & ys & = [\,] \\
zipWith & f & xs & [\,] & = [\,] \\
zipWith & f & (x : xs) & (y : ys) & = f\ x\ y : zipWith\ f\ xs\ ys
\end{array}
$$

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:

$$
\begin{array}{l}
zip = \quad zipWith\ maak2tupel \\
\quad \textbf{where}\ maak2tupel\ x\ y = (x,y)
\end{array}
$$

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 weten inmiddels beter: deze functie heeft één parameter, 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 tupel-methode. Gecurryde functies zijn immers partieel te parametriseren, en functies met een tupel-parameter niet. Alle standaardfuncties met meer dan één parameter werken dan ook volgens de Curry-methode.

In de prelude wordt een functie-transformatie (functie met functie als parameter en andere functie als resultaat) gedefinieerd, die van een gecurryde functie een functie met tupel-parameter maakt. Deze functie heet *uncurry*:

$$\begin{aligned} &\mathit{uncurry} :: (a \to b \to c) \to ((a, b) \to c) \\ &\mathit{uncurry}\ f\ (a, b) = f\ a\ b \end{aligned}$$

Andersom is er een functie *curry* die van een functie met tupel-parameter een gecurryde functie maakt. Dus *curry plus*, met *plus* zoals hierboven, kan wèl partieel geparametriseerd worden.

Opgaven

6.1 Schrijf een functie die het laatste element van een lijst oplevert.

6.2 Schrijf een functie die het een-na-laatste element van een lijst oplevert.

6.3 Schrijf een functie die het i-de element van een lijst oplevert.

6.4 Schrijf een functie die het aantal elementen van een lijst oplevert.

6.5 Schrijf een functie die een lijst omdraait.

6.6 Schrijf een functie die bepaalt of een lijst een palindroom is.

6.7 Schrijf een functie die een lijst van lijsten platslaat tot een lijst: $[[1,2],[3],[],[4,5]]$ wordt bijvoorbeeld $[1,2,3,4,5]$.

6.8 Schrijf een functie die opeenvolgende duplicaten uit een lijst verwijdert: $[1,2,2,3,2,4]$ wordt bijvoorbeeld $[1,2,3,2,4]$.

6.9 Schrijf een functie die opeenvolgende duplicaten samenvoegt in sublijsten: $[1,2,2,3,2,4]$ wordt bijvoorbeeld $[[1],[2,2],[3],[2],[4]]$.

6.10 Schrijf een functie die de zogenaamde 'run-length encoding' van een lijst bepaalt: $[1,2,2,3,2,4]$ wordt bijvoorbeeld

$$[(1,1),(2,2),(1,3),(1,2),(1,4)].$$

6.11 Ga na dat $[1,2] +\!\!+ [\,]$ volgens de definitie van $+\!\!+$ inderdaad $[1,2]$ oplevert. Hint: schrijf $[1,2]$ als $1 : (2 : [\,])$.

6.12 Schrijf de functie *concat* als aanroep van *foldr*.

6.13 Welke van de volgende expressies levert *True* op voor alle lijsten xs, en welke *False*:

$$\begin{aligned} [[\,]] +\!\!+ xs &\equiv xs \\ [[\,]] +\!\!+ xs &\equiv [xs] \\ [[\,]] +\!\!+ xs &\equiv [[\,], xs] \\ [[\,]] +\!\!+ [xs] &\equiv [[\,], xs] \\ [xs] +\!\!+ [\,] &\equiv [xs] \\ [xs] +\!\!+ [xs] &\equiv [xs, xs] \end{aligned}$$

6.14 De functie *filter* kan gedefiniëerd worden in termen van *concat* en *map*:

$$\begin{aligned} &\mathit{filter}\ p = \mathit{concat} \circ \mathit{map}\ \mathit{box} \\ &\quad \textbf{where}\ \mathit{box}\ x = \end{aligned}$$

Completeer de definitie van de functie *box* die hierin is gebruikt.

6.15 Schrijf met behulp van de functie *iterate* een niet-recursieve definitie van *repeat*.

6.16 Schrijf een functie met twee lijsten als parameter, die van elk element uit de tweede lijst het eerste voorkomen in de eerste lijst verwijdert. (Deze functie is in de prelude als operator gedefinieerd en heet λ*listdiffoperator*).

6.17 Gebruik de functies *map* en *concat* in plaats van de lijstcomprehensie-notatie om de volgende lijst te definiëren:

$$[(x, y + z) \mid x \leftarrow [1\,..\,10], y \leftarrow [1\,..\,x], z \leftarrow [1\,..\,y]]$$

6.18 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 $\equiv$. Deze functie levert *True* op als twee breuken dezelfde waarde hebben, ook als de breuken niet vereenvoudigd zijn.

6.19 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 deel-functie eerst een formule af voor $\frac{1}{a+bi}$ door x en y op te lossen uit $(a+bi) * (x+yi) = (1+0i)$.

6.20 Schrijf een functie *stringInt*, die van een string de overeenkomstige integer maakt. Bijvoorbeeld: *stringInt* {λ*kwoot* 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?

6.21 Bewijs dat de volgende eigenschap geldt: $\mathit{reverse} \circ \mathit{reverse} = \mathit{id}$, waarbij de functie *id* gedefiniëerd is als:

$$\mathit{id}\ x = x$$

6.22 We kunnen een matrix representeren als een lijst van even lange lijsten. Schrijf een functie $\mathit{transpose} :: [[a]] \rightarrow [[a]]$, 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:

$$\begin{aligned} &\mathit{zipWith}\ \mathit{op}\ (x : \mathit{xs})\ (y : \mathit{ys}) = (x\ `\mathit{op}`\ y) : \mathit{zipWith}\ \mathit{xs}\ \mathit{ys} \\ &\mathit{zipWith}\ \mathit{op}\ \_\qquad\ \_\qquad\ = [\,] \end{aligned}$$

Hoofdstuk 7

Algoritmen op lijsten

Combinatorische functies

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 parameter-lijst, zijn het polymorfe functies:

$$\begin{array}{lll} \mathit{inits}, \mathit{tails}, \mathit{segs} & :: & [a] \to [[a]] \\ \mathit{subs}, \mathit{perms} & :: & [a] \to [[a]] \\ \mathit{combs} & :: \mathit{Int} \to & [a] \to [[a]] \end{array}$$

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:

$$inits\ [\,] = [[\,]]$$

Voor het geval $(x : xs)$ kijken we naar de gewenste uitkomsten voor de lijst $[1, 2, 3, 4]$.

$$\begin{array}{lcl} inits\ [1,2,3,4] & = & [\ [\,]\ ,\ [1]\ ,\ [1,2]\ ,\ [1,2,3]\ ,\ [1,2,3,4]\ ] \\ inits\ [2,3,4] & = & [\ [\,]\ ,\ [2]\ ,\ [2,3]\ ,\ [2,3,4]\ ] \end{array}$$

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*:

$$\begin{array}{l} inits\ [\,] = [[\,]] \\ inits\ (x : xs) = [\,] : map\ (x:)\ (inits\ xs) \end{array}$$

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 $[1, 2, 3, 4]$:

$$\begin{array}{lcl} tails\ [1,2,3,4] & = & [\ [1,2,3,4]\ ,\ [2,3,4]\ ,\ [3,4]\ ,\ [4]\ ,\ [\,]\ ] \\ tails\ [2,3,4] & = & [\ [2,3,4]\ ,\ [3,4]\ ,\ [4]\ ,\ [\,]\ ] \end{array}$$

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).

Gebruik makend van dit idee kan de complete definitie geschreven worden:

$$\begin{array}{l} tails\ [\,] = [[\,]] \\ tails\ (x : xs) = (x : xs) : tails\ xs \end{array}$$

De haakjes in de tweede regel zijn essentieel: zonder 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 singleton-lege-lijst.

Om op het spoor van de definitie van *segs* $(x : xs)$ te komen, passen we de beproefde methode weer toe:

$$\begin{array}{lcl} segs\ [1,2,3,4] & = & [\ []\ ,\ [4]\ ,\ [3]\ ,\ [3,4]\ ,\ [2]\ ,\ [2,3]\ ,\ [2,3,4]\ ,\ [1]\ ,\ [1,2]\ ,\ [1,2,3]\ ,\ [1,2,3,4]\ ] \\ segs\ [2,3,4] & = & [\ []\ ,\ [4]\ ,\ [3]\ ,\ [3,4]\ ,\ [2]\ ,\ [2,3]\ ,\ [2,3,4]\ ] \end{array}$$

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:

$$\begin{array}{l} segs\ [\,] = [[\,]] \\ segs\ (x : xs) = segs\ xs +\!\!+ tail\ (inits\ (x : xs)) \end{array}$$

Een andere manier om de lege lijst uit de *inits* te verwijderen is:

$$\begin{array}{l} segs\ [\,] = [[\,]] \\ segs\ (x : xs) = segs\ xs +\!\!+ map\ (x:)\ (inits\ xs) \end{array}$$

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:

$$\begin{array}{lcl} subs\ [1,2,3,4] & = & [\ [1,2,3,4]\ ,\ [1,2,3]\ ,\ [1,2,4]\ ,\ [1,2]\ ,\ [1,3,4]\ ,\ [1,3]\ ,\ [1,4]\ ,\ [1] \\ & & \ ,\ [2,3,4]\ ,\ [2,3]\ ,\ [2,4]\ ,\ [2]\ ,\ [3,4]\ ,\ [3]\ ,\ [4]\ ,\ []\ ] \\ subs\ [2,3,4] & = & [\ [2,3,4]\ ,\ [2,3]\ ,\ [2,4]\ ,\ [2]\ ,\ [3,4]\ ,\ [3]\ ,\ [4]\ ,\ []\ ] \end{array}$$

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:

$$\begin{array}{l} subs\ [\,] = [[\,]] \\ subs\ (x : xs) = map\ (x:)\ (subs\ xs) +\!\!+ subs\ xs \end{array}$$

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:

$$\begin{array}{l} subs\ [\,] = [[\,]] \\ subs\ (x : xs) = map\ (x:)\ subsxs +\!\!+ subsxs \\ \quad \textbf{where} \\ \quad\quad subsxs = subs\ xs \end{array}$$

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 integer-parameter de gevallen 0 en $n+1$ onderscheiden. In het geval $n+1$ worden ook voor de lijst-parameter twee gevallen onderscheiden. De definitie krijgt dus de volgende vorm:

```
combs 0 xs           = ...
combs (n+1) []       = ...
combs (n+1) (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 een singleton-lege-lijst. Het maakt daarbij niet uit of *xs* leeg is of niet.
- Het patroon $n+1$ betekent '1 of meer'. Het kiezen van minstens één element uit de lege lijst is onmogelijk. Het resultaat van *combs* $(n+1)$ [] 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 $n+1$ elementen uit de lijst $x : xs$ is wel mogelijk, mits het kiezen van n 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 $n+1$ 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+1) []       = []
combs (n+1) (x : xs) = map (x:) (combs n xs) ++ combs (n+1) xs
```

Bij deze functie kunnen de twee recursieve aanroepen niet gecombineerd worden, zoals bij *subs*. De twee aanroepen hebben hier namelijk verschillende parameters.

De @-notatie

De definitie van *tails* heeft iets omslachtigs:

```
tails []       = [[]]
tails (x : xs) = (x : xs) : tails xs
```

In de tweede regel wordt de parameter-lijst 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:

$$
\begin{aligned}
&\mathit{tails}\ [\,] &&= [[\,]]\\
&\mathit{tails}\ \mathit{xs} &&= \mathit{xs} : \mathit{tails}\ (\mathit{tail}\ \mathit{xs})
\end{aligned}
$$

Nu is het opnieuw opbouwen van de parameter-lijst 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:

$$
\begin{aligned}
&\mathit{tails}\ [\,] &&= [[\,]]\\
&\mathit{tails}\ \mathit{lyst}@(x : \mathit{xs}) &&= \mathit{lyst} : \mathit{tails}\ \mathit{xs}
\end{aligned}
$$

Hoewel het symbool @ in operator-symbolen gebruikt mag worden, is een losse @ speciaal gereserveerd voor deze constructie.

Bij functies die al eerder in dit diktaat gedefinieerd werden, komt een @-patroon ook goed van pas. Bijvoorbeeld in de functie *dropWhile*:

$$
\begin{aligned}
&\mathit{dropWhile}\ p\ [\,] = [\,]\\
&\mathit{dropWhile}\ p\ \mathit{ys}@(x : \mathit{xs})\\
&\quad \mid p\ x \qquad\quad = \mathit{dropWhile}\ p\ \mathit{xs}\\
&\quad \mid \mathit{otherwise}\ = \mathit{ys}
\end{aligned}
$$

Dit is ook de manier waarop *dropWhile* in werkelijkheid in de prelude is gedefinieerd.

Matrixrekening

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 een-dimensionale lijn, het twee-dimensionale platte vlak en de drie-dimensionale 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:

$$\textbf{type}\ \mathit{Vector} = [\mathit{Float}]$$

Om achter getalconstanten niet steeds `.0` te hoeven schrijven om de type-checker tevreden te stellen, zullen we in deze sectie veronderstellen dat de `+i` optie van de interpreter aan staat, zodat *Int*-getallen zonodig automatisch naar *Float* geconverteerd worden.

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
blz. 117 om een 'beschermd type' te definiëren zoals beschreven in paragraaf 8:

$$\textbf{data}\ \mathit{Vector} = \mathit{Vec}\ [\mathit{Float}]$$

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 lijnstuk-interpretatie, de *lengte* van het lijnstuk:

$$vecLengte :: Vector \to Float$$

Bij twee vectoren (in dezelfde ruimte) kan bepaald worden of ze *loodrecht* op elkaar staan, en meer algemeen welke *hoek* ze met elkaar maken:

$$\begin{aligned} vecLoodrecht &:: Vector \to Vector \to Bool \\ vecHoek &:: Vector \to Vector \to Float \end{aligned}$$

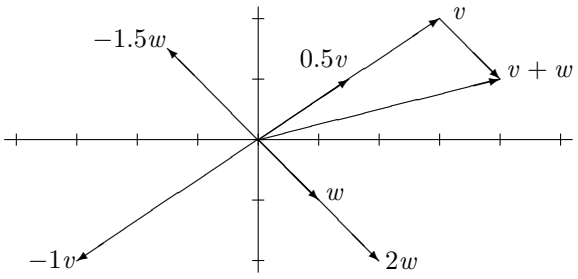
Verder kan een vector met een getal worden 'vermenigvuldigd' door alle getallen in de lijst (alle coördinaten) met dat getal te vermenigvuldigen. Twee vectoren worden 'opgeteld' door alle coördinaten op te tellen:

$$\begin{aligned} vecScale &:: Float \to Vector \to Vector \\ vecPlus &:: Vector \to Vector \to Vector \end{aligned}$$

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:



$$\begin{aligned} v &= [3,2] \\ w &= [1,-1] \\ vecPlus\ v\ w &= [4,1] \\ vecScale\ 0\ v &= [0,0] \\ vecScale\ 0.5\ v &= [1.5,1] \\ vecScale\ -1\ v &= [-3,-2] \\ vecScale\ 2\ w &= [2,-2] \\ vecScale\ -1.5\ w &= [-1.5,1.5] \end{aligned}$$

Functies van vectoren naar vectoren heten *afbeeldingen*. Van bijzonder belang zijn *lineaire* afbeeldingen. Dat zijn afbeeldingen waarbij elke coördinaat van de beeld-vector 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 kolom-representatie 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

$$matTransp \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

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 $(\circ)$) 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 identieke afbeelding is ook een lineaire afbeelding. Hij wordt beschreven door de *identiteits-matrix*. 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:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}$$

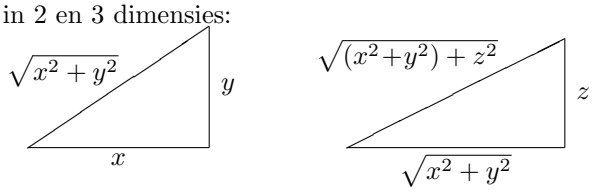
Het is duidelijk dat de begrippen 'identiteitsmatrix' en 'inverse' alleen zin hebben voor vierkante matrices, dus matrices met dezelfde origineel- en beeldruimte. En zelfs voor vierkante matrices is de inverse niet altijd gedefinieerd. De matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ bijvoorbeeld heeft geen inverse.

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:

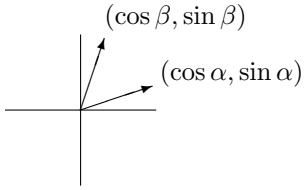


In het algemeen (willekeurige dimensie) kan de lengte van een vector uitgerekend worden door de wortel van de som van de kwadraten van de coördinaten te berekenen. De functie luidt dus:

$$vecLengte\ (Vec\ xs) = sqrt\ (sum\ (map\ kwadraat\ xs))$$

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
blz. 54 worden met de functie *inverse* uit paragraaf 4.

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*:

$$\begin{aligned} &vecScale && :: Float \to Vector \to Vector \\ &vecScale\ k\ (Vec\ xs) && = Vec\ (map\ (k*)\ xs) \\ &vecPlus && :: Vector \to Vector \to Vector \\ &vecPlus\ (Vec\ xs)\ (Vec\ ys) && = Vec\ (zipWith\ (+)\ xs\ ys) \end{aligned}$$

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:

$$\begin{aligned} &mapp && :: (a \to b) \to [[a]] \to [[b]] \\ &mapp\ f && = map\ (map\ f) \end{aligned}$$

Anders gezegd:

$$mapp = map \circ map$$

Voor *zippWith* geldt iets dergelijks:

$$\begin{aligned} &zippWith :: (a \to b \to c) \to [[a]] \to [[b]] \to [[c]] \\ &zippWith = zipWith \circ zipWith \end{aligned}$$

Deze functies kunnen gebruikt worden in *matScale* en *matPlus*:

$$\begin{aligned} &matScale && :\ Float \to Matrix \to Matrix \\ &matScale\ k\ (Mat\ xss) && = Mat\ (mapp\ (k*)\ xss) \\ &matPlus && :: Matrix \to Matrix \to Matrix \\ &matPlus\ (Mat\ xss)\ (Mat\ yss) && = Mat\ (zippWith\ (+)\ xss\ yss) \end{aligned}$$

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

$$transpose :: [[a]] \to [[a]]$$

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 deel-oplossing 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). Als functie-parameter 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* [ ]. blz. 44

Een 'lege matrix' kan beschouwd worden al 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 eenvoudigweg

$$transpose = foldr\ (zipWith\ (:))\ (repeat\ [\,])$$

Functioneel programmeren is programmeren met functies...

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:

$$f\ [4,5,6]\ =\ 4x + 5y + 6z$$

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:

$$\begin{aligned} &matApply && :: Matrix \to Vector \to Vector \\ &matApply\ (Mat\ m)\ v = Vec\ (map\ f\ m) \\ &\quad \textbf{where}\ f\ rij = vecInprod\ (Vec\ rij)\ v \end{aligned}$$

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 identieke afbeelding. 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
```

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.

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 $\binom{0\ 0}{0\ 0}$ beeldt bijvoorbeeld elk punt af op de oorsprong, en heeft dus geen inverse. De matrix $\binom{1\ 1}{2\ 2}$ beeldt elk punt af op een punt van de lijn $y = 2x$, en heeft dus ook geen inverse. Ook de matrix $\binom{1\ 2}{3\ 6}$ 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 $\binom{a\ b}{c\ d}$ 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 $\binom{d\ e\ f}{g\ h\ i}$. Dan bereken je de determinanten van de 2×2-matrices die je krijgt door uit de 'overige rijen' steeds één kolom weg te laten ($\binom{e\ f}{h\ i}$, $\binom{d\ f}{g\ i}$ en $\binom{d\ e}{g\ h}$). Die vermenigvuldig je met de overeenkomstige elementen uit de eerste rij. Tenslotte tel je ze op, waarbij om de andere term een min-teken krijgt. Dit werkt ook in hogere dimensies dan 3.

Definitie van de determinant

Van deze informele beschrijving van de determinant gaan we een functie-definitie 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 rest-rijen *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
blz. 114
kan met de combinatorische functie *gaps* uit opgave 7.8. Het resultaat is een lijst van n lijsten van lijsten. Die moeten weer terug-getransponeerd worden, en er moeten met *Mat* kleine matrixjes van gemaakt worden:

```
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, moet 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
```

Het functie-definitie 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 terug-transponeren 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 functie-samenstelling 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 min-tekens, 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 matrix-inverse 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*,
blz. 114 enzovoort. Het wordt aan de lezer overgelaten om de details uit te werken (zie opgave 7.11).

Polynomen

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:

$$
\begin{array}{ll}
pPlus & :: Poly \to Poly \to Poly \\
pMin & :: Poly \to Poly \to Poly \\
pMaal & :: Poly \to Poly \to Poly \\
pEq & :: Poly \to Poly \to Bool \\
pGraad & :: Poly \to Int \\
pEval & :: Float \to Poly \to Float \\
polyString & :: Poly \to String
\end{array}
$$

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 gelijkheids-operator 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 twee-tupels. We maken er echter meteen maar een *data*type van met de volgende definitie:

$$
\begin{array}{l}
\textbf{data}\ Poly\ = Poly\ [\,Term\,] \\
\textbf{data}\ Term = Term\ (Float, Int)
\end{array}
$$

Let op: de namen *Poly* en *Term* worden dus zowel als naam van het type gebruikt, als 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 type-declaraties zoals

$$pEq :: Poly \to Poly \to Bool$$

maar het is een constructorfunctie in functiedefinities zoals

$$pGraad\ (Poly\ [\,]) = \ldots$$

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 6 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: blz. 87

$$
\begin{array}{l}
Poly\ [\,Term\ (1.0, 2), Term\ (7.0, 0)] \\
Poly\ [\,Term\ (7.0, 0), Term\ (1.0, 2)] \\
Poly\ [\,Term\ (1.0, 2), Term\ (3.0, 0), Term\ (4.0, 0)]
\end{array}
$$

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.

Vereenvoudiging

Voor het vereenvoudigen van polynomen schrijven we een functie

$$pEenvoud :: Poly \to Poly$$

Deze functie voert de drie genoemde aspecten van het vereenvoudigen uit, en kan dus geschreven worden als functie-samenstelling:

```
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.

blz. 73 In paragraaf 6 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 sorteer-functie 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 sorteer-criterium.

Daarom schrijven we eerst een functie *sortVolgens*, die dan gebruikt kan worden bij het schrijven van *sortTerms*.

De algemene sorteer-functie *sortVolgens*, heeft als type:

$$sortVolgens :: (a \to a \to Bool) \to [a] \to [a]$$

Behalve de te sorteren lijst heeft de functie een functie als parameter. Die parameter-functie levert *True* op als zijn eerste parameter vóór zijn tweede parameter moet komen. De definitie van blz. 73 *sortVolgens* lijkt sterk op die van *isort* in paragraaf 6. Het verschil is dat nu de als extra parameter meegegeven vergelijk-functie wordt gebruikt in plaats van $<$. De definitie wordt dan:

```
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:

$$
\begin{aligned}
&samenvExpo\ \ :: [\,Term] \to [\,Term]\\
&samenvExpo\ [\,] = \ [\,]\\
&samenvExpo\ [t] = [t]\\
&samenvExpo\ (Term\ (c1, e1) : Term\ (c2, e2) : ts)\\
&\quad \mid e1 \equiv e2 \ \ = \ \ samenvExpo\ (Term\ (c1 + c2, e1) : ts)\\
&\quad \mid otherwise \ \ = \ \ Term\ (c1, e1) : samenvExpo\ (Term\ (c2, e2) : ts)
\end{aligned}
$$

De derde benodigde functie is eenvoudig te maken:

$$
\begin{aligned}
&verwijderNul :: [\,Term] \to [\,Term]\\
&verwijderNul = filter\ coefNietNul\\
&\quad \textbf{where}\\
&\quad\quad coefNietNul\ (Term\ (c, e)) = c \not\equiv 0.0
\end{aligned}
$$

Desgewenst kunnen de drie functies lokaal gedefinieerd worden in *pEenvoud*:

$$
\begin{aligned}
&pEenvoud\ (Poly\ xs) = Poly\ (eenvoud\ xs)\\
&\quad \textbf{where}\\
&\quad\quad eenvoud && = vN \circ sE \circ sT\\
&\quad\quad sT && = sortVolgens\ expoGroter\\
&\quad\quad sE\ [\,] && = [\,]\\
&\quad\quad sE\ [t] && = [t]\\
&\quad\quad sE\ (Term\ (c1, e1) : Term\ (c2, e2) : ts)\\
&\quad\quad\quad \mid e1 \equiv e2 && = sE\ (Term\ (c1 + c2, e1) : ts)\\
&\quad\quad\quad \mid otherwise && = Term\ (c1, e1) : sE\ (Term\ (c2, e2) : ts)\\
&\quad\quad vN && = filter\ coefNietNul\\
&\quad\quad coefNietNul\ (Term\ (c, e)) && = c \not\equiv 0.0\\
&\quad\quad Term\ (c1, e1)\ `expoGroter`\ Term\ (c2, e2) && = e1 > e2
\end{aligned}
$$

De functie *pEenvoud* verwijdert *alle* termen waarvan de coëfficiënt nul is. Het nul-polynoom wordt dus gerepresenteerd door $Poly\ [\,]$, een lege lijst termen. Daarmee verschilt het nul-polynoom van andere polynomen waarin de variabele niet voorkomt. Het polynoom '3' wordt bijvoorbeeld gerepresenteerd door $Poly\ [\,Term\ (3.0, 0)]$.

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 nul-termen verwijderd worden:

$$
\begin{aligned}
&pPlus && :: Poly \to Poly \to Poly \\
&pPlus\ (Poly\ xs)\ (Poly\ ys) && = pEenvoud\ (Poly\ (xs +\!\!+ ys))
\end{aligned}
$$

Voor het aftrekken van twee polynomen tellen we het eerste polynoom op bij het tegengestelde van het tweede:

$$
\begin{aligned}
&pMin && :: Poly \to Poly \to Poly \\
&pMin\ p1\ p2 && = pPlus\ p1\ (pNeg\ p2)
\end{aligned}
$$

Blijft natuurlijk de vraag hoe het tegengestelde van een polynoom berekend wordt: daartoe moet het tegengestelde van elke term berekend worden.

$$
\begin{aligned}
&pNeg && :: Poly \to Poly \\
&pNeg\ (Poly\ xs) && = Poly\ (map\ tNeg\ xs) \\
&tNeg && :: Term \to Term \\
&tNeg\ (Term\ (c, e)) && = Term\ (-c, e)
\end{aligned}
$$

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:

$$
\begin{aligned}
&cpWith && :: (a \to b \to c) \to [a] \to [b] \to [c] \\
&cpWith\ f\ [\,]\ ys && = [\,] \\
&cpWith\ f\ (x : xs)\ ys && = map\ (f\ x)\ ys +\!\!+ cpWith\ f\ xs\ ys
\end{aligned}
$$

Deze definitie kan meteen gebruikt worden bij het vermenigvuldigen van polynomen:

$$
\begin{aligned}
&pMaal && :: Poly \to Poly \to Poly \\
&pMaal\ (Poly\ xs)\ (Poly\ ys) && = pEenvoud\ (Poly\ (cpWith\ tMaal\ xs\ ys))
\end{aligned}
$$

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:

$$
\begin{aligned}
&tMaal && :: Term \to Term \to Term \\
&tMaal\ (Term\ (c1, e1))\ (Term\ (c2, e2)) && = Term\ (c1 * c2, e1 + e2)
\end{aligned}
$$

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 nul-polynoom hebben we een aparte definitie nodig.

$$
\begin{aligned}
&pGraad && :: Poly \to Int \\
&pGraad\ (Poly\ [\,]) && = 0 \\
&pGraad\ (Poly\ (Term\ (c, e) : ts)) && = e
\end{aligned}
$$

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:

$$
\begin{aligned}
&pEq && :: Poly \to Poly \to Bool \\
&pEq\ (Poly\ xs)\ (Poly\ ys) && = length\ xs \equiv length\ ys
\end{aligned}
$$

$$
\begin{array}{ll}
 & \wedge\ and\ (zipWith\ tEq\ xs\ ys) \\
tEq & :: Term \to Term \to Bool \\
tEq\ (Term\ (c1, e1))\ (Term\ (c2, e2)) & = eqFloat\ c1\ c2 \wedge e1 \equiv e2
\end{array}
$$

De functie *pEval* moet een polynoom uitrekenen met een specifieke waarde voor x ingevuld. Daartoe moeten alle termen geëvalueerd worden, en de resultaten opgeteld:

$$
\begin{array}{ll}
pEval & :: Float \to Poly \to Float \\
pEval\ w\ (Poly\ xs) & = sum\ (map\ (tEval\ w)\ xs) \\
tEval & :: Float \to Term \to Float \\
tEval\ w\ (Term\ (c, e)) & = c * w\ \hat{}.\ e
\end{array}
$$

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:

$$
\begin{array}{ll}
polyString & :: Poly \to String \\
polyString\ (Poly\ [\,]) & = \texttt{"0"} \\
polyString\ (Poly\ [t]) & = termString\ t \\
polyString\ (Poly\ (t : ts)) & = termString\ t +\!\!+ \texttt{" + "} \\
 & \quad +\!\!+ polyString\ (Poly\ ts)
\end{array}
$$

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:

$$
\begin{array}{ll}
termString & :: Term \to String \\
termString\ (Term\ (c, 0)) & = floatString\ c \\
termString\ (Term\ (1.0, 1)) & = \texttt{"x"} \\
termString\ (Term\ (1.0, e)) & = \texttt{"x\^{}"} +\!\!+ intString\ e \\
termString\ (Term\ (c, e)) & = floatString\ c +\!\!+ \texttt{"x\^{}"} +\!\!+ intString\ e
\end{array}
$$

De functie *floatString* is nog niet gedefinieerd. Met een boel moeite is dat wel mogelijk, maar dat is niet nodig: in paragraaf 9 wordt hiervoor de functie *show* geïntroduceerd. blz. 136

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. 93.

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 uitdrukkenmet behup van een *foldr*?.

7.7 Schrijf een functie $bins :: Int \to [[Char]]$ 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 functie-samenstellings-operator ($\circ$).

7.10 Ga na dat de recursieve definitie van matrix-determinant *det* overeenkomt met de expliciete definitie voor determinanten van 2×2-matrices uit paragraaf 7.
blz. 106

7.11 Schrijf de functie *matInv*.

7.12 In paragraaf 7 werd de functie *transpose* beschreven als generalisatie van *zip*. In paragraaf 6 werd *zip* geschreven als *zipWith maak2tupel*. De functie *cp* 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?
blz. 102
blz. 88

7.13 Een andere mogelijke representatie van polynomen is 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.

Hoofdstuk 8

Datastructuren

Datatypes

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*.

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:

$$\textbf{data}\ \mathit{Richting} = \mathit{Noord} \mid \mathit{Oost} \mid \mathit{Zuid} \mid \mathit{West}$$

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:

$$\begin{aligned} &\mathit{verticaleRichtingen} :: [\mathit{Richting}] \\ &\mathit{verticaleRichtingen} = [\mathit{Noord}, \mathit{Zuid}] \end{aligned}$$

Functies op dit soort types kunnen gewoon met behulp van patronen worden geschreven, bijvoorbeeld:

$$\begin{aligned} &\mathit{move} :: \mathit{Richting} \to (\mathit{Int}, \mathit{Int}) \to (\mathit{Int}, \mathit{Int}) \\ &\mathit{move}\ \mathit{Noord}\ (x, y) = (x\ \ \ \ , y + 1) \\ &\mathit{move}\ \mathit{Oost}\ \ (x, y) = (x + 1, y\ \ \ \ ) \end{aligned}$$

$$\begin{array}{ll} move\ Zuid & (x, y) = (x \quad\ , y - 1) \\ move\ West & (x, y) = (x - 1, y \quad\ ) \end{array}$$

De voordelen van zo'n eindig type boven een codering met integers of characters zijn:

- functie-definities zijn duidelijker doordat de namen van de elementen gebruikt kunnen worden, in plaats van obscure coderingen;
- het type-systeem 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:

$$\textbf{data}\ Bool = False \mid True$$

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:

$$\textbf{data}\ Ordering = LT \mid EQ \mid GT$$

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 data-definitie is het mogelijk een type *IntOfChar* te maken, die als elementen zowel de integers als de characters heeft:

$$\begin{array}{l} \textbf{data}\ IntOfChar = EenInt\ Int \\ \qquad\qquad\qquad\quad \mid\ EenChar\ Char \end{array}$$

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 *constructor-functies*. Om een waarde te construeren van het type *IntOfChar* dienen we een van de constructor-functies toe te passen op een parameter van het juiste type:

$$\begin{array}{l} getal :: IntOfChar \\ getal = EenInt\ 4 \\ letter :: IntOfChar \\ letter = EenChar\ \texttt{'a'} \end{array}$$

We kunnen hiermee ook een 'gemengde' lijst maken:

$$\begin{array}{l} xs :: [IntOfChar] \\ xs = [EenInt\ 1, EenChar\ \texttt{'a'}, EenInt\ 2, EenInt\ 3] \end{array}$$

De enige prijs die je moet betalen, is dat elk element gemarkeerd moet worden met de constructor-functie *EenInt* of *EenChar*. Deze functies zijn te beschouwen als conversiefuncties:

$$\begin{array}{ll} EenInt & :: Int \to IntOfChar \\ EenChar & :: Char \to IntOfChar \end{array}$$

waarvan het gebruik vergelijkbaar is met dat van ingebouwde conversiefuncties zoals

$$\begin{array}{ll} \mathit{truncate} & :: \mathit{Float} \to \mathit{Int} \\ \mathit{chr} & :: \mathit{Int} \to \mathit{Char} \end{array}$$

Met patroonherkenning kunnen we functies schrijven die iets met waarden van het nieuwe type doen:

$$\begin{array}{ll} \mathit{toonIntOfChar} & :: \mathit{IntOfChar} \to \mathit{String} \\ \mathit{toonIntOfChar}\ (\mathit{EenInt}\ i) & = \mathit{showInt}\ i \\ \mathit{toonIntOfChar}\ (\mathit{EenChar}\ c) & = [c] \end{array}$$

Merk op dat de ronde haakjes nodig zijn omdat het er anders uitziet alsof *toonIntOfChar* twee parameters krijgt!

Beschermde types

Beschermde types

In paragraaf 6 werd een nadeel genoemd van type-definities: als twee types op dezelfde manier worden gedefinieerd, bijvoorbeeld blz. 86

$$\begin{array}{ll} \textbf{type}\ \mathit{Datum} & = (\mathit{Int}, \mathit{Int}) \\ \textbf{type}\ \mathit{Ratio} & = (\mathit{Int}, \mathit{Int}) \end{array}$$

dan kunnen ze door elkaar worden gebruikt. 'Datums' kunnen daardoor ineens worden verwerkt alsof het 'rationale getallen' zijn, zonder dat dat foutmeldingen van de type-checker oplevert.

Met data-definities is het mogelijk om echte nieuwe types te maken, zodat bijvoorbeeld een *Ratio* niet meer zonder meer uitwisselbaar is met elke andere $(\mathit{Int}, \mathit{Int})$. In plaats van de type-definitie wordt daartoe de volgende data-definitie gegeven:

$$\textbf{data}\ \mathit{Ratio} = \mathit{Rat}\ (\mathit{Int}, \mathit{Int})$$

Er is dus slechts één constructor-functie. 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 $\mathit{Rat}\ (3, 5)$. Net als bij verenigings-types kan *Rat* worden beschouwd als conversiefunctie van $(\mathit{Int}, \mathit{Int})$ naar *Ratio*. Het is eigenlijk wel zo handig om ook constructor-functies te curryen. In dat geval krijgen ze niet een tupel als parameter, maar twee losse waarden. De bijbehorende datatype-definitie is:

$$\textbf{data}\ \mathit{Ratio} = \mathit{Rat}\ \mathit{Int}\ \mathit{Int}$$

Deze methode wordt veel gebruikt om *beschermde types* te maken. Een beschermd type bestaat uit een data-definitie 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 constructor-functie '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 constructor-functie wordt vaak dezelfde naam als de naam van het type gekozen, dus bijvoorbeeld

$$\textbf{data}\ \mathit{Ratio} = \mathit{Ratio}\ \mathit{Int}\ \mathit{Int}$$

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 constructor-functie).

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 wortel trekken 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:

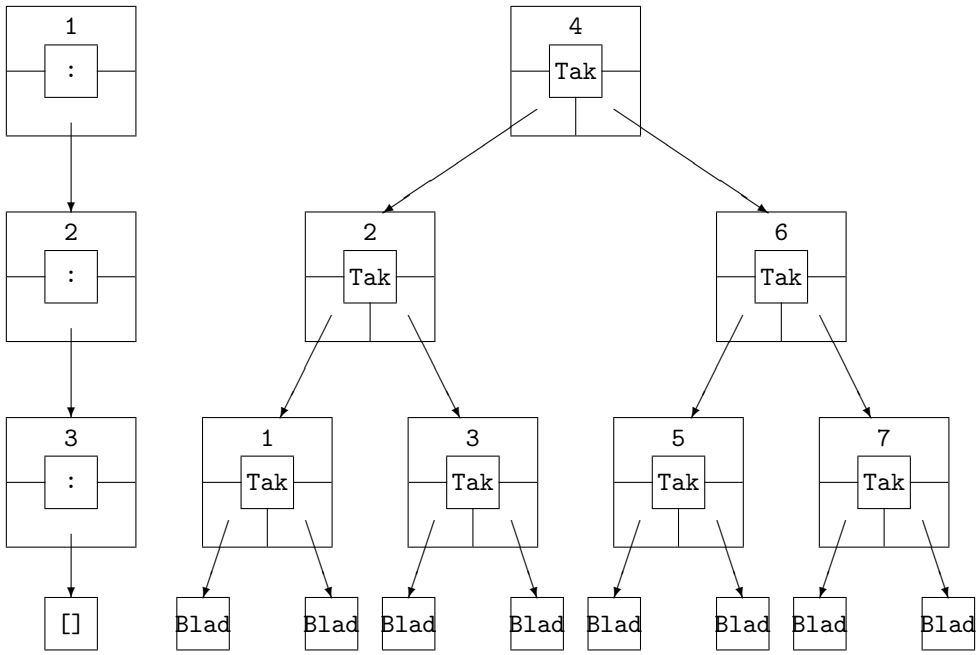
```
Ja  :: a → Misschien a
Nee :: Misschien a
```

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
```



lijststructuur *boomstructuur*

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)
      )
```

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. 119) maakt de boomstructuur nog veel duidelijker.

Functies op een boom kunnen gedefinieerd worden door voor elke constructor-functie 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.

Lijsten: een speciaal soort 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*))). $[\,]$ 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 mooi 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 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
  ```

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 6 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 blz. 68

```
elem    :: Eq a ⇒ a → [a] → Bool
elem e xs = or (map (≡ e) xs)
```

of direct met recursie

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

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.

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 de elementen nu niet alleen vergelijkbaar moeten zijn (klasse *Eq*), maar ook ordenbaar (klasse *Ord*):

```
elem'                  :: Ord a ⇒ a → [a] → Bool
elem' e []             = False
elem' e (x:xs) | e < x = False
```

$$
\begin{array}{l}
\mid e \equiv x = \mathit{True} \\
\mid e > x = \mathit{elem}'\ e\ \mathit{xs}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Boom}\ a = \mathit{Tak}\ a\ (\mathit{Boom}\ a)\ (\mathit{Boom}\ a) \\
\qquad\qquad \mid \mathit{Blad}
\end{array}
$$

Op elk splitspunt is een element opgeslagen, en twee (kleinere) bomen: een 'linker' deelboom en een 'rechter' deelboom (zie de figuur op blz. 119). In een zoekboom wordt nu bovendien geëist dat alle waarden in de linker deelboom *kleiner* zijn dan de waarde in het splitspunt, en alle waarden in de rechter deelboom *groter*. De waarden in de voorbeeldboom in de genoemde figuur zijn zo gekozen, dat de afgebeelde boom inderdaad een zoekboom is.

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 linker deelboom (in de rechter deelboom zitten immers grotere waarden). Andersom, als de gezochte waarde groter is dan de opgeslagen waarde, moet juist in de rechter deelboom worden doorgezocht. De functie *elemBoom* is dus als volgt:

$$
\begin{array}{lll}
\mathit{elemBoom} & & :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Boom}\ a \to \mathit{Bool} \\
\mathit{elemBoom}\ e\ \mathit{Blad} & & = \mathit{False} \\
\mathit{elemBoom}\ e\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) & \mid e \equiv x & = \mathit{True} \\
 & \mid e < x & = \mathit{elemBoom}\ e\ \mathit{li} \\
 & \mid e > x & = \mathit{elemBoom}\ e\ \mathit{re}
\end{array}
$$

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 door-
blz. 88 zocht. Ook in bijvoorbeeld de functie *zoekOp* uit paragraaf 6 is met behulp van zoekbomen een dramatische snelheidswinst te boeken.

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 constructor-functies. 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
blz. 73 paragraaf 6), 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:

$$
\begin{array}{lll}
\mathit{insertBoom} & & :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Boom}\ a \to \mathit{Boom}\ a \\
\mathit{insertBoom}\ e\ \mathit{Blad} & & = \mathit{Tak}\ e\ \mathit{Blad}\ \mathit{Blad} \\
\mathit{insertBoom}\ e\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) & \mid e \leqslant x & = \mathit{Tak}\ x\ (\mathit{insertBoom}\ e\ \mathit{li})\ \mathit{re} \\
 & \mid e > x & = \mathit{Tak}\ x\ \mathit{li}\ (\mathit{insertBoom}\ e\ \mathit{re})
\end{array}
$$

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 rechter deelboom 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 6. blz. 73

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 sorteer-algoritme. 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 linker deelboom èn de rechter deelboom. 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. 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 rechter-deelboom.

```
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 $e \equiv x$), 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 linker deelboom leeg is, is samenvoegen natuurlijk ook geen probleem:

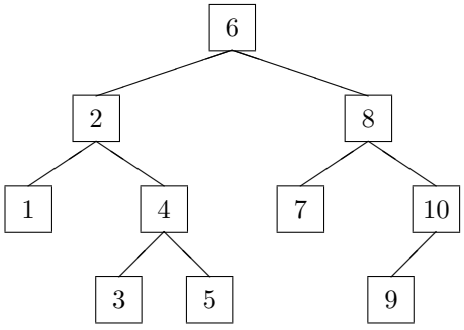
```
samenvoegen           :: Boom a → Boom a → Boom a
samenvoegen Blad b2 = b2
samenvoegen b1 b2   = Tak x b1' b2
  where
    (x, b1') = grootsteUit b1
```

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 rechter deelboom af te dalen:

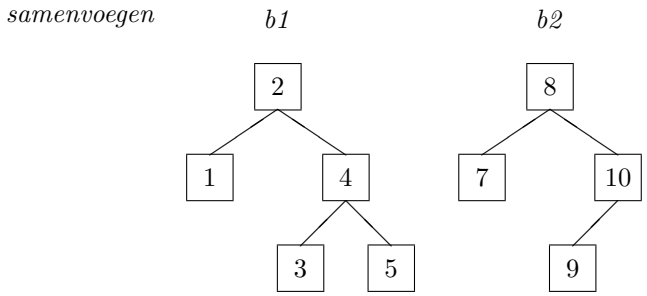
```
grootsteUit                    :: Boom a → (a, Boom a)
grootsteUit (Tak x b1 Blad) = (x, b1)
grootsteUit (Tak x b1 b2)   = (y, Tak x b1 b2')
  where
    (y, b2') = grootsteUit b2
```

Om de werking van *deleteBoom* te demonstreren bekijken we een voorbeeld, waarbij we voor de duidelijkheid de bomen grafisch voorstellen. Bij de aanroep van

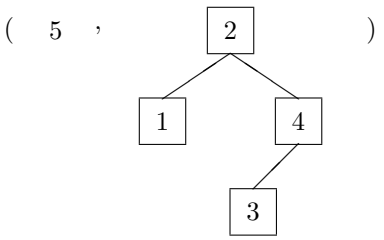
deleteBoom 6



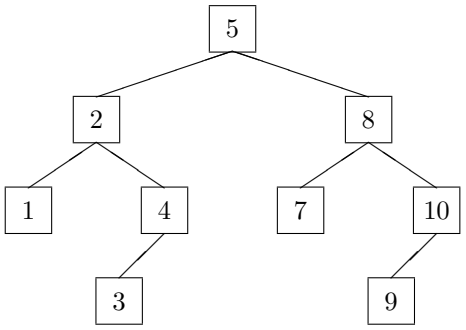
wordt de functie *samenvoegen* aangeroepen met de linker- en de rechter deelboom als parameter:



Door *samenvoegen* wordt de functie *grootsteUit* aangeroepen met *b1* als parameter. Dat levert een tweetupel $(x, b1')$ op:



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

8.1 Schrijf een zoekboom-versie van de functie *zoekOp*, zoals *elemBoom* een zoekboom-versie is van *elem*. Geef ook het type van de functie.

8.2 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* ?

8.3 Geef een definitie van *until* die gebruik maakt van *iterate* en *dropWhile*.

8.4 Geef een directe definitie van de operator $<$ op lijsten. Deze definitie mag dus geen gebruik maken van operatoren zoals $\leqslant$ op lijsten. (Als je deze definitie daadwerkelijk met de Haskell-interpretator wilt uitproberen, gebruik dan een andere naam dan $<$, omdat de operator $<$ al in de prelude wordt gedefinieerd.)

8.5 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?

blz. 73

8.6 In paragraaf 6 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 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*?

8.7 Beschouw de functie *groepeer* met het volgende type:

$$groepeer :: Int \to [a] \to [[a]]$$

Deze functie deelt de een gegeven lijst in deel-lijsten (die in een lijst van lijsten worden opgeleverd), waarbij de deel-lijsten een gegeven lengte hebben. Alleen de laatste deel-lijst 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]]
```

blz. 80

Schrijf deze functie volgens hetzelfde principe als de functie *intString* in paragraaf 6, dat wil zeggen door samenstelling van een aantal partiële parametrisaties van *iterate*, *takeWhile* en *map*.

8.8 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.

8.9 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.

8.10 Schrijf een functie *toonBoom*, die een aantrekkelijk representatie als string van een boom zoals gedefinieerd op blz. 119 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.

8.11 Stel dat een boom b diepte n heeft. Wat is het minimale en het maximale aantal bladeren dat b kan bevatten?

8.12 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.

8.13 Schrijf functies die de waarden van type a in de knopen van een boom van type:

```
data Boom a = Knoop (Boom a) a (Boom a)
            | Blad
```

in depth-first, infix en breadth-first order oplevert.

8.14 Schrijf een functie die alle paden van type $[a]$ oplevert van de wortel tot een blad in een boom van type *Boom a*.

8.15 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.

Hoofdstuk 9

Klassen en hun instanties

Numerieke types

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 element-type worden toegepast. Voorbeelden van polymorfe functies: *length*, *concat* en *map*.
- *Overloading*. Een overloaded functie kan op een aantal verschillende types werken die niets met elkaar te maken hebben. De operator + werkt bijvoorbeeld zowel op type *Int* als op type *Float*. De operator $\leqslant$ kan op *Int* en *Float* werken, en daarnaast ook op *Char*, twee-tupels en lijsten.

blz. 128

In paragraaf 9 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* vormen samen bijvoorbeeld *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 +:

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

De tekst *Num a* kan gelezen worden als 'type *a* zit in klasse *Num*'. Het geheel heeft de betekenis: '+ heeft het type $a \to a \to a$ 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:

$$(\equiv) :: Eq\ a \Rightarrow a \to a \to Bool$$
$$(\leqslant) :: Ord\ a \Rightarrow a \to a \to Bool$$

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 type-variabele in een bepaalde klasse zit, rechts ervan staat een type waar deze typevariabele in gebruikt wordt.

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 zelf-gedefinieerde).

Het definiëren van een klasse heet een *klasse-declaratie*, het toevoegen van een type aan een klasse een *instance-declaratie*.[1] De drie standaard-klassen zijn niet ingebouwd in Haskell. Ze worden in de prelude gedefinieerd door middel van gewone klasse-declaraties. Ook het feit dat *Int* en *Float* deel uitmaken van de klasse *Num* wordt in de prelude gedefinieerd door middel van instance-declaraties. Er is dus niets speciaals aan de drie standaard-klassen.

De definitie van de klasse *Num* is een goed voorbeeld van een klasse-declaratie. Deze ziet er, iets vereenvoudigd, als volgt uit:

class *Num a* **where**
  $(+), (-), (*), (/) :: a \to a \to a$
  negate $:: a \to a$

Een klasse-declaratie bestaat dus uit de volgende onderdelen:

- het (speciaal voor dit doel) gereserveerde woord **class**;
- de naam van de klasse (*Num* in het voorbeeld);
- een type-variabele (*a* in het voorbeeld);
- het gereserveerde woord **where**;
- type-declaraties voor operatoren en functies, waarbij de genoemde type-variabele gebruikt mag worden.

In een instance-declaratie wordt voor de aldus gedefinieerde operatoren een definitie gegeven. De instance-declaratie 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 instance-declaratie 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 klasse-declaratie werden gedeclareerd.

Bij de instance-declaratie *Num Int* zijn de functie-definities een beetje flauw, omdat simpelweg wordt aangegeven dat voor elke functie de betreffende ingebouwde functie op integers genomen moet worden. De instance-declaratie *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*

[1] Het woord *instance* betekent letterlijk 'voorbeeld' (vergelijk de uitdrukking *for instance*). Het type *Int* is als het ware een 'voorbeeld' van een type in de klasse *Num*. Zo wordt dat overigens in het Nederlands meestal niet gezegd. Het woord *instance* wordt doorgaans onvertaald gelaten, of vervangen door het anglicisme 'instantie'.

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.

Nieuwe numerieke types

Als je zelf een type hebt gedefinieerd, waarop de numerieke operatoren zouden moeten werken, dan kan het nieuwe type met een instance-declaratie lid gemaakt worden van de klasse *Num*. In
blz. 87 paragraaf 6 werd bijvoorbeeld het type *Ratio* der rationale getallen gedefinieerd (de verzameling $\mathbf{Q}$). 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 instance-declaratie:

```
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 functie-definitie ook direct in de instance-declaratie geschreven worden. Vaak worden de instance-declaraties direct na de type-declaratie 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
blz. 117 te beschermen met een beschermd datatype (zie paragraaf 8). Je gebruikt 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 klasse-declaratie van *Num* heeft die operator het type $a \to a \to a$, waarbij a het type is van een instance van *Num*. In deze instance-declaratie 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*.

Numerieke constanten

Door het klasse-mechanisme 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:

$$\begin{array}{lcl} verdubbel & :: & Num\ a \Rightarrow a \to a \\ verdubbel\ x & = & x + x \\ halveerInt & :: & Int \to Int \\ halveerInt\ n & = & n\ /\ 2 \\ halveerFloat & :: & Float \to Float \\ halveerFloat\ x & = & x\ /\ 2.0 \end{array}$$

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 klasse-declaratie luidt dus:

class $Num\ a$ **where**
$(+), (-), (*), (/) :: a \to a \to a$
$negate :: a \to a$
$fromInteger :: Int \to a$

Hiermee wordt gespecificeerd dat er voor elk instance-type van *Num* een conversie-functie moet zijn van *Int* naar dat type. Die conversiefunctie moet gedefinieerd worden in de instance-declaratie. Voor het type *Int* is dat gemakkelijk:

instance $Num\ Int$ **where**
$\ldots$
$fromInteger\ n = n$

Voor het type *Float* zit er niets anders op dan een ingebouwde functie te gebruiken

instance $Num\ Float$ **where**
$\ldots$
$fromInteger = primIntToFloat$

Voor zelf-gedefinieerde 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* $a \Rightarrow a \to a$
halveer x = x / *fromInteger* 2

Omdat het in de praktijk tamelijk vervelend is om op iedere getal-constante de functie *fromInteger* toe te passen, is het in Haskell mogelijk om dit automatisch te laten doen. Nadat aan de interpreter
blz. 17 de opdracht `:set +i` is gegeven (zie paragraaf 2), wordt 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 integer-constanten te kunnen werken:

```
? sqrt 2
1.41421
```

'Lijken te werken', want wat er eigenlijk uitgerekend wordt is *sqrt* (*fromInteger* 2).

Ordening en gelijkheid

Default-definities

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 klasse-declaratie is als volgt:

class *Eq* a **where**
  $(\equiv), (\not\equiv)$ :: $a \to a \to$ *Bool*

Bij elke instance-declaratie voor deze klasse moeten dus de operatoren $\equiv$ (gelijkheid) en $\not\equiv$ (ongelijkheid) worden gedefinieerd. De vier standaard-types *Int*, *Float*, *Char* en *Bool* worden in de prelude alle als instance van *Eq* gedefinieerd:

instance *Eq Int* **where**
  $x \equiv y$ = *primEqInt* x y
  $x \not\equiv y$ = $\neg\ (x \equiv y)$
instance *Eq Float* **where**
  $x \equiv y$ = *primEqFloat* x y
  $x \not\equiv y$ = $\neg\ (x \equiv y)$
instance *Eq Char* **where**
  $x \equiv y$ = *ord* $x \equiv$ *ord* y
  $x \not\equiv y$ = $\neg\ (x \equiv y)$
instance *Eq Bool* **where**
  True $\equiv$ *True* = *True*
  False $\equiv$ *False* = *True*
  True $\equiv$ *False* = *False*
  False $\equiv$ *True* = *False*
  $x \not\equiv y$ = $\neg\ (x \equiv y)$

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 gedefineerde operator $\equiv$ op integers. Gelijkheid op *Bool*'s tenslotte wordt direct door middel van patronen gedefinieerd.

In alle vier de gevallen wordt ongelijkheid ($\not\equiv$) gedefinieerd door het resultaat van $\equiv$ om te keren met $\neg$. De definitie is in alle gevallen precies hetzelfde (behalve natuurlijk dat telkens de $\equiv$ uit een andere instance wordt gebruikt). Dit soort definities mag ook reeds in de klasse-declaratie 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[2] van een definitie in de instance-declaraties wordt gebruikt. Wordt een functie waarvoor een default-definitie bestaat tòch in de instance-declaratie gedefinieerd, dan gaat die definitie voor.

De klasse-declaratie *Eq* zoals die werkelijk in de prelude staat is dus als volgt:

class *Eq a* **where**
$(\equiv), (\not\equiv) :: a \to a \to Bool$
$x \not\equiv y \quad = \neg\,(x \equiv y)$

De definitie van $\not\equiv$ in de instance-declaraties is weggelaten, omdat de default-definitie voldoet.

Ook zelfgedefinieerde types kunnen tot instance van *Eq* gemaakt worden. Voor het type *Ratio* kan de gelijkheids-definitie uit opgave 8.18 gebruikt worden: blz. 91

instance *Eq Ratio* **where**
$Rat\ (x, y) \equiv Rat\ (p, q) = x * q \equiv y * p$

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 $\not\equiv$ kan, net als in de instance-declaraties in de prelude, achterwege blijven. De default-definitie is ook in dit geval immers bruikbaar.

Klassen met voorwaarden

De types waarvan de elementen ordenbaar zijn (met operatoren zoals $\leqslant$) zijn instances van de klasse *Ord*. In de klasse-declaratie voor *Ord* wordt aangegeven dat een type ook een instance van *Eq* moet zijn, wil het ordenbaar zijn:

class *Eq a* $\Rightarrow$ *Ord a* **where**
$(\leqslant), (<), (\geqslant), (>) :: a \to a \to Bool$
$max, min \qquad :: a \to a \to a$

Alle operatoren en functies in deze klasse met uitzondering van $\leqslant$ hebben een default-definitie. De enige operator die in instance-declaraties gedefinieerd hoeft te worden is dus $\leqslant$. Het is vanwege deze default-definities dat geëist wordt dat instances van *Ord* ook instances van *Eq* zijn. In de default-definitie van $<$ wordt namelijk de operator $\not\equiv$ gebruikt. De default-definities luiden:

$x < y \qquad = x \leqslant y \wedge x \not\equiv y$
$x \geqslant y \qquad = y \leqslant x$
$x > y \qquad = y < x$
$max\ x\ y \mid x \geqslant y = x$
$\qquad\quad \mid y \geqslant x = y$
$min\ x\ y \mid x \leqslant y = x$
$\qquad\quad \mid y \leqslant x = y$

[2] *default* betekent letterlijk 'ontbreken'

De instance-declaratie *Ord Int* en *Ord Float* doen voor de definitie van $\leqslant$ een beroep op een ingebouwde functie. Voor characters wordt de ordening bepaald door de integer-ordening van hun iso/ascii codes:

instance *Ord Char* **where**
$x \leqslant y = ord\ x \leqslant ord\ y$

Net als *Ord* eist de klasse-declaratie voor *Num* in de prelude dat de instances ook een instance van de klasse *Eq* zijn. De, nu helemaal complete klasse-declaratie voor *Num* luidt derhalve:

class *Eq a* $\Rightarrow$ *Num a* **where**
$(+), (-), (*), (/) :: a \to a \to a$
negate $:: a \to a$
fromInteger $:: Int \to a$

De vergelijkbaarheid van de elementen van instances van *Num* wordt echter niet gebruikt in default-definities, 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, onzinnig beschouwd worden.

Instances met voorwaarden

Ook bij instance-declaraties 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* $\Rightarrow$ *Eq* $[a]$ **where**
$[\,] \equiv [\,]$ $= True$
$[\,] \equiv (y : ys)$ $= False$
$(x : xs) \equiv [\,]$ $= False$
$(x : xs) \equiv (y : ys) = x \equiv y \wedge xs \equiv ys$

De eerste regel van deze instance-declaratie 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 $\equiv$ is opmerkelijk: aan de rechterkant komt twee maal een aanroep van $\equiv$ voor. De eerste stelt gelijkheid op (de eerste) elementen van de lijst voor (dat kan, want dat was immers de voorwaarde van de instance-declaratie). De tweede aanroep van $\equiv$ 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...

blz. 70

In paragraaf 6 werd al opgemerkt dat lijsten een ordening kennen: de lexicografische ordening. Deze ordening wordt gedefinieerd door een instance-declaratie in de prelude:

instance *Ord a* $\Rightarrow$ *Ord* $[a]$ **where**
$[\,] \leqslant ys$ $= True$
$(x : xs) \leqslant [\,]$ $= False$
$(x : xs) \leqslant (y : ys) = x < y \vee (x \equiv y \wedge xs \leqslant ys)$

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 ordenings-operatoren hoeven niet gedefinieerd te worden: daarvoor worden de default-definities gebruikt.

Een instance-declaratie kan meer dan één voorwaarde hebben. Die moeten dan tussen haakjes genoteerd worden, met komma's ertussen. Hiermee kan bijvoorbeeld de gelijkheid van twee-tupels gedefinieerd worden, zoals in de prelude gebeurt:

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)$ **where**
$(x, y) \equiv (u, v) = x \equiv u \wedge y \equiv v$

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 ($\Rightarrow$) kan gebruikt worden in type-declaraties, in instance-declaraties en in klasse-declaraties. Let op het verschil in betekenis van deze drie vormen:

- $f :: Num\ a \Rightarrow a \rightarrow a$ is een type-declaratie: f is een functie met type $a \rightarrow a$ mits a een type is in de klasse *Num*.
- **instance** $Eq\ a \Rightarrow Eq\ [a]$ is een instance-declaratie: $[a]$ is een instance van *Eq* mits a dat ook is.
- **class** $Eq\ a \Rightarrow Ord\ a$ is een klasse-declaratie: alle instances van de nieuwe klasse *Ord* moeten ook instances zijn van *Eq*.

Standaard-klassen

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 index-types;
- *Text*, de klasse van afdrukbare types.

De eerste drie werden al eerder besproken. Hieronder volgt een korte beschrijving van de andere drie (die minder vaak gebruikt worden).

de klasse Enum

De klasse *Enum* is als volgt gedefinieerd:

class $Ord\ a \Rightarrow Enum\ a$ **where**
$enumFrom :: a \rightarrow [a]$
$enumFromThen :: a \rightarrow a \rightarrow [a]$
$enumFromTo :: a \rightarrow a \rightarrow [a]$
$enumFromThenTo :: a \rightarrow a \rightarrow a \rightarrow [a]$

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.

Standaard-instances van deze klasse zijn *Int*, *Float* en *Char*. Een paar voorbeelden van het resultaat van de functies voor verschillende types:

enumFrom 4	=	$[4, 5, 6, 7, 8, \ldots$
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 6 werden besproken. Deze notaties kunnen dus blz. 64 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.

De instance-declaratie *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 instance-declaratie 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 default-definitie in de klasse-declaratie:

```
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 ordenings-operatoren 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 instance-declaraties 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 klasse-declaratie 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 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 klasse Text

De klasse *Text* declareert twee functies, waarmee respectievelijk een element en een lijst van het instance-type omgezet kunnen worden in een *String*. Deze functies worden zelden direct gebruikt. Meestal wordt in plaats daarvan de functie *show* gebruikt, die in de prelude wordt gedefinieerd:

```
show :: Text a ⇒ a → String
```

Met de functie *show* kan elke waarde van een type dat een instance is van *Text* worden omgezet naar een *String*. Standaard-instances van *Text* zijn *Int*, *Float*, *Char* en lijsten en tweetupels waarvan de element-types ook instances zijn van *Text*.

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 $\equiv$ niet zonder meer op drietupels toegepast worden. Een oplossing van dit probleem is de ontbrekende instance-declaratie aan het programma toe te voegen.

Deze foutmelding wordt ook gegeven als je functies probeert te vergelijken. Functie-types 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 14 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). blz. 191

'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 instance-declaratie een speciaal geval is van een type waarvoor al een andere instance-declaratie bestaat. Bijvoorbeeld: in de prelude worden tweetupels gedeclareerd als instance van *Eq*:

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)$ **where**
$(x, y) \equiv (u, v) = x \equiv u \wedge y \equiv v$

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**
$(x, y) \equiv (u, v) = x * v \equiv u * y$

Bij een aanroep van $(1, 2) \equiv (2, 4)$ kan de interpreter nu niet kiezen: wordt de standaard tupel-gelijkheid bedoeld of de *Ratio*-gelijkheid? Bij het analyseren van de instance-declaratie 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
blz. 131 beschreven in paragraaf 9). Een onschuldig ogende expressie als $1 + 2$ heeft dan immers al een unresolved overloading tot gevolg.

Meestal is deze fout te herstellen door de interpreter een extra hint te geven over het gewenste type. Dat kan bijvoorbeeld door een type-declaratie te geven voor de kritieke functies, of door de dubieuze expressie direct te typeren:

```
? fromInteger 1 + fromInteger 2 :: Int
3
```

Klassen en wetten

Wetten voor standaardklassen

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**
  $(n, xs) \leqslant (k, ys) = n + k \equiv length\ ys$

(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**
  $Rat\ (x, y) \leqslant Rat\ (u, v) = x * v \leqslant u * y$

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 klasse-declaratie 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.[3]

Wil de operator $\equiv$ zijn naam 'gelijkheids-operator' 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*:

$$gemeen\ (Rat\ (t, n)) = t + n$$

Als het om beschermde datatypes gaat, wordt de congruentie-eis 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 patroon-herkenning direct gebruik maken van de representatie van het datatype, hoeft de congruentiewet niet te gelden.

De wetten waar de ordenings-operator $\leqslant$ 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 zelf-gedefinieerde 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 default-definitie van *min* en *max* in paragraaf 9 niet *otherwise* in de tweede regel. blz. 133
Als twee elementen 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 14. De bewijzen in die paragraaf waren in feite het bewijs dat het type *Nat* een waarlijk numeriek type is. blz. 208
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 $+$ $\qquad y + (x - y) = x$

[3] 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.

Een overeenkomstige wet voor de delings-operator / geeft echter problemen:

$$/ \text{ is de inverse van } * \qquad \text{als } y \neq 0 \text{ dan } y * (x/y) = x$$

Deze wet is bijvoorbeeld niet geldig voor de ingebouwde deling op gehele getallen. Bovendien is er sprake van een getal 0, en wat moeten we daarvoor nemen in een kandidaat-numeriek type?

In Haskell zijn alle numerieke types een instance van de klasse *Num*. De klasse-declaratie en
blz. 131 de instance-declaraties die daarvoor nodig zijn staan in de prelude, en werden in paragraaf 9 besproken. Voor een precieze beschrijving van het onderscheid tussen verschillende numerieke types is een indeling in meerdere klassen eigenlijk geschikter.

Opgaven

9.1 Schrijf een declaratie waardoor de operatoren $+$, $-$, $*$ en / ook op complexe getallen (zie
blz. 91 opgave 8.19) gebruikt kunnen worden.

9.2 Verklaar uit de manier waarop + gedefinieerd is, dat het uitrekenen van $1 + 2$ de tweede keer
blz. 181 één reductie minder kost dan de eerste keer(zie ook paragraaf 14).

9.3 Definieer een type *Set a* dat verzamelingen voorstelt met elementen uit *a*. Gebruik lijsten om dit type te implementeren. Definieer een functie

$$subset :: Set\ a \to Set\ a \to Bool$$

die controleert of een verzameling een deelverzameling is van een andere. Schrijf vervolgens een instance-declaratie 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 gedefinieerd worden als beschermd datatype, dus met behulp van **data** en niet met **type**?

9.4 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;
- $(a \to b)$, mits *a* en *b* eindig zijn en *Eq a* (moeilijk, voor de liefhebber).

Maak nu $(a \to b)$ tot instance van *Eq* mits *a* eindig is. Zijn er nog meer voorwaarden voor deze instance-declaratie?

Hoofdstuk 10

Case Study: A Class Hiërarchy

Een klasse-hiërarchie

In de wiskunde is het al heel lang gebruikelijk om verschillende soorten verzamelingen te onderscheiden, al naar gelang de operatoren die er op gedefinieerd zijn en de wetten die ervoor gelden. Het indelen van verzamelingen in soorten is in feite niets anders dan het indelen van types in klassen. In deze en de volgende paragrafen wordt de gebruikelijke wiskundige indeling dan ook beschreven aan de hand van klasse- en instance-declaraties in Haskell. Wat eerst een programmeertaal was, is daarmee een gereedschap geworden om wiskunde te bedrijven.

We zullen, overeenkomstig het wiskundige gebruik, de numerieke types indelen in vijf klassen: *monoïden*, *groepen*, *ringen*, *euclidische ringen* en *lichamen*. Deze vijf klassen vormen een hiërarchie: elke klasse voegt een aantal operatoren (en wetten) toe aan de vorige klasse. In de klasse-declaraties komt dat tot uiting in het feit dat de 'hogere' klassen de 'lagere' klassen als voorwaarde hebben:

class *Eq a* ⇒ *Monoid a* **where** ...
class *Monoid a* ⇒ *Groep a* **where** ...
class *Groep a* ⇒ *Ring a* **where** ...
class *Ring a* ⇒ *Euclid a* **where** ...
class *Euclid a* ⇒ *Lichaam a* **where** ...

Een type kan dus alleen een instance van *Ring* zijn als het ook een instance van *Groep* is; daarvoor moet het weer een instance van *Monoid* zijn, en dat kan alleen als het een instance van *Eq* is. Anders gezegd: er zijn veel types die een instance zijn van *Monoid*, maar naarmate er meer operatoren en wetten nodig zijn, vallen er steeds meer types af; uiteindelijk zijn er maar een paar types instance van *Lichaam*.

In de vijf klassen worden operatoren, onder andere 'plus' en 'maal', gedefinieerd. Om ze niet te verwarren met de operatoren + en ∗ die in *Num* worden gedefinieerd, zullen we ze hieronder noteren als <+> en <*>. Als je onderstaande definities zou beschouwen als *vervanging* in plaats van als *aanvulling* van de prelude, dan zou je ze ook gewoon + en ∗ kunnen noemen.

Hieronder volgen de klasse-declaraties van de vijf genoemde klassen. Bij elke klasse staan de wetten genoemd waaraan de operatoren moeten voldoen. Deze definities komen misschien wat abstract over, maar in de paragrafen paragraaf 10 tot paragraaf 10 staan allerlei instance-declaraties, waarmee concrete types ingedeeld worden in de abstracte klassen. Een wiskundige zou zeggen: in die paragrafen worden *voorbeelden* gegeven van de abstracte soorten (wat betekent 'instance' ook alweer?).

blz. 144
blz. 150

Monoïden

Er is maar heel weinig nodig om een type een monoïde te kunnen noemen. Er moet een *associatieve operator* zijn, die een *neutraal element* heeft. De klasse-declaratie is als volgt:

```
class Eq a ⇒ Monoid a where
  0       :: a
  (<+>) :: a → a → a
```

De wetten waar deze operatoren aan moeten voldoen zijn:

M1 $x \mathbin{<+>} (y \mathbin{<+>} z) = (x \mathbin{<+>} y) \mathbin{<+>} z$
M2 $\mathbf{0} \mathbin{<+>} x = x$
M3 $x \mathbin{<+>} \mathbf{0} = x$

Enigszins suggestief hebben we de operator <+> genoemd, en het neutrale element **0**. Dat je hierbij
blz. 144 niet per se aan optellen en het getal 0 hoeft te denken blijkt in paragraaf 10.

In de klasse-declaratie wordt als voorwaarde gesteld dat de elementen van het type vergelijkbaar zijn (een instance van *Monoid* moet ook instance zijn van *Eq*). De gelijkheid tussen elementen is nodig, omdat je anders bezwaarlijk wetten kunt opstellen: daarin wordt immers gelijkheid van bepaalde elementen gesproken.

Groepen

Wil een monoïde ook als groep beschouwd kunnen worden, dan is er naast <+> nog een functie nodig. Deze functie geeft bij elk element een *tegenovergestelde*. De klasse-declaratie is als volgt:

```
class Monoid a ⇒ Groep a where
  neg :: a → a
```

In de wetten voor een *Groep* wordt gesteld dat een element opgeteld bij zijn tegenovergestelde de waarde **0** oplevert. Bovendien moet de operator <+> nu behalve associatief ook *commutatief* zijn[1]. De wetten voor een *Groep* luiden:

G1 $x \mathbin{<+>} \mathit{neg}\ x = \mathbf{0}$
G2 $x \mathbin{<+>} y = y \mathbin{<+>} x$

Ringen

Bij een ring doet de operator <*> de intrede. Deze operator moet associatief zijn, en een neutraal element ('**1**') hebben. De instance-declaratie luidt:

```
class Groep a ⇒ Ring a where
  1       :: a
  (<*>) :: a → a → a
```

In de wetten voor een *Ring* wordt naast associativiteit van <*> en neutraliteit van **1**, gesteld dat de operator <*> *distribueert* over <+>. Er zijn twee distributieve wetten nodig, omdat <*> niet commutatief hoeft te zijn. De wetten voor *Ring* luiden:

R1 $x \mathbin{<*>} (y \mathbin{<*>} z) = (x \mathbin{<*>} y) \mathbin{<*>} z$
R2 $\mathbf{1} \mathbin{<*>} x = x$
R3 $x \mathbin{<*>} \mathbf{1} = x$
R4 $x \mathbin{<*>} (y \mathbin{<+>} z) = (x \mathbin{<*>} y) \mathbin{<+>} (x \mathbin{<*>} z)$
R5 $(y \mathbin{<+>} z) \mathbin{<*>} x = (y \mathbin{<*>} x) \mathbin{<+>} (z \mathbin{<*>} x)$

Hoewel de namen '<*>' en '**1**' enigszins suggestief gekozen zijn, moet je je wel realiseren dat <*> niet in elke instance 'vermenigvuldigen' hoeft te betekenen.

[1] Je ziet ook vaak definities van 'groep' waarin commutativiteit van de operator niet wordt verondersteld. Als de operator wel commutatief is, spreekt men van een 'commutatieve groep'. Omdat we het hier verder alleen over commutatieve groepen zullen hebben, noemen we dat eenvoudigweg 'groep'.

Domeinen

Een *domein* is een ring waarin nog enkele extra wetten gelden. Er worden echter geen nieuwe operatoren toegevoegd, dus we maken er geen aparte klasse van. De wetten die in een domein gelden zijn de volgende:

D1 $x \mathbin{<\!\ast\!>} y = y \mathbin{<\!\ast\!>} x$
D2 als $x \neq \mathbf{0}$ en $y \neq \mathbf{0}$, dan is $x \mathbin{<\!\ast\!>} y \neq \mathbf{0}$
D3 $\mathbf{1} \neq \mathbf{0}$

Euclidische ringen

Een Euclidische ring is een domein waarin *deling met rest* mogelijk is. De klasse-declaratie is:

```
class Ring a ⇒ Euclid a where
  orde :: a → Int
  quot :: a → a → a
  rest :: a → a → a
```

Bij elk twee elementen kan dus een 'quotiënt' en een 'rest' berekend worden. Deze enigszins suggestieve namen worden gerechtvaardigd door de wetten. De 'rest' moet kleiner zijn dan de 'noemer' van de deling, en als je het 'quotiënt' met de noemer vermenigvuldigt, houd je precies de rest over. In deze wet wordt het begrip 'kleiner' gebruikt. In plaats van de voorwaarde te stellen dat een instance van *Euclid* ook een instance van *Ord* is, is er een functie *orde* in de klasse opgenomen. Elementen kunnen dan 'geordend' worden door hun respectievelijke ordes te ordenen.

E1 $(n \mathbin{<\!\ast\!>} quot\ t\ n) \mathbin{<\!+\!>} rest\ t\ n = t$
E2 $orde\ (rest\ t\ n) < orde\ n$

De definitie van *rest* kan als default-definitie in de klasse-declaratie worden opgenomen. Op grond van wet E1 geldt namelijk:

$$rest\ t\ n = t \ominus (n \mathbin{<\!\ast\!>} quot\ t\ n)$$

Lichamen

Een *Lichaam* heeft de rijkste structuur van de vijf genoemde klassen. Naast alle eerder genoemde functies is er in een lichaam bij elk element (behalve $\mathbf{0}$) een 'omgekeerde':

```
class Euclid a ⇒ Lichaam a where
  omg :: a → a
```

In de wetten wordt gespecificeerd dat het product van een element en zijn omgekeerde de waarde $\mathbf{1}$ oplevert:

L1 $x \mathbin{<\!\ast\!>} omg\ x = \mathbf{1}$

Afgeleide operatoren

Gebruikmakend van de operatoren en functies in de verschillende klassen, is het mogelijk om nieuwe operatoren te definiëren. Deze operatoren worden daardoor vanzelf ook overloaded.

Op types uit de klasse *Groep* kan de operator $\ominus$ gedefinieerd worden als het optellen van het tegengestelde:

$(\ominus) \quad :: Groep\ a \Rightarrow a \to a \to a$
$x \ominus y = x \mathbin{<\!+\!>} neg\ y$

Voor deze operator gelden allerlei wetten, die volgen uit de gegeven wetten en de definitie. In een ring geldt bijvoorbeeld dat 'vermenigvuldigen' distribueert over 'aftrekken': $x \mathbin{<\!\ast\!>} (y \ominus z) = (x \mathbin{<\!\ast\!>} y) \ominus (x \mathbin{<\!\ast\!>} z)$ (zie opgave 9.1). blz. 150

In een ring kun je vermenigvuldigen. Machtsverheffen kan gedefinieerd worden als herhaald vermenigvuldigen:

$$
\begin{array}{ll}
(\oslash) & :: \mathit{Ring}\ a \Rightarrow a \to \mathit{Int} \to a \\
x \oslash 0 & = \mathbf{1} \\
x \oslash (n+1) & = x \mathbin{<\!\ast\!>} (x \oslash n)
\end{array}
$$

Het komt goed van pas dat er in de ring een element **1** beschikbaar is om te dienen als resultaat van $x \oslash 0$. Merk op dat de rechter parameter van $<\backslash macht>$ een gewone integer is, waarvoor we dus 0 schrijven, en niet **0**. Omdat $\mathbin{<\!\ast\!>}$ een associatieve operator is, maakt het niet uit of we links of rechts van de recursieve aanroep vermenigvuldigen met x.

In een Euclidische ring kun je 'delen met rest'. Je kunt dus ook bepalen of de rest gelijk aan nul is, en daarmee het begrip 'deelbaar' definiëren:

$$
\begin{array}{ll}
\mathit{deelbaar} & :: \mathit{Euclid}\ a \Rightarrow a \to a \to \mathit{Bool} \\
\mathit{deelbaar}\ x\ y & = \mathit{rest}\ x\ y \equiv \mathbf{0}
\end{array}
$$

Omdat de elementen dankzij de functie *orde* ordenbaar zijn, is het in een Euclidische ring mogelijk om van een 'grootste gemene deler' te spreken. Daarvoor kan het algoritme in paragraaf 6 gegeneraliseerd worden naar een willekeurige Euclidische ring: blz. 88

$$
\begin{array}{ll}
\mathit{ggd}\ x\ y \mid y \equiv \mathbf{0} & = x \\
\quad \mid \mathit{otherwise} & = \mathit{ggd}\ y\ (\mathit{rest}\ x\ y)
\end{array}
$$

Instances van de klassen

Diverse types die in dit diktaat aan de orde zijn geweest, kunnen als instance gedeclareerd worden van de klassen in de hiërarchie. Het type *Int* bijvoorbeeld is een *Monoid*, een *Groep*, een *Ring* en een *Euclid* (maar geen *Lichaam*):

```
instance Monoid Int where
  (<+>) = (+)
  0     = 0
instance Groep Int where
  neg   = negate
instance Ring Int where
  (<*>) = (*)
  1     = 1
instance Euclid Int where
  orde  = abs
  quot  = (/)
  rest  = rem
```

Het type *Float* is behalve dat ook een instance van *Lichaam*. Dat *Float* een instance is van *Euclid* is haast te flauw om te definiëren: elke deling gaat in *Float* altijd precies op, dus de rest is altijd nul (dat geldt ook in andere lichamen).

```
instance Monoid Float where
  (<+>)  = (+)
```

```
  0       = 0.0
instance Groep Float where
  neg     = negate
instance Ring Float where
  (<*>)   = (*)
  1       = 1.0
instance Euclid Float where
  orde x  = 0
  quot    = (/)
  rest x y = 0.0
instance Lichaam Float where
  omg     = (1.0/)
```

Ook de rationale getallen zijn een instance van alle vijf de klassen. Daarbij kan de benadering gekozen worden uit paragraaf 6 (alle rationale getallen worden na een berekening direct vereenvoudigd), of die uit opgave 8.18 (gelijkheid wordt zodanig gedefinieerd, dat hij ook werkt op niet-vereenvoudigde breuken). Hieronder is voor die laatste benadering gekozen. De rationale getallen zijn als beschermd datatype gedefinieerd.

blz. 87 blz. 91

```
data Ratio                  = Rat (Int, Int)
instance Eq Ratio where
  Rat (a, b) ≡ Rat (c, d)   = a * d ≡ c * b
instance Monoid Ratio where
  Rat (a, b) <+> Rat (c, d) = Rat (a * d + c * b, b * d)
  0                         = Rat (0, 1)
instance Groep Ratio where
  neg (Rat (a, b))          = Rat (−a, b)
instance Ring Ratio where
  Rat (a, b) <*> Rat (c, d) = Rat (a * c, b * d)
  1                         = Rat (1, 1)
instance Lichaam Ratio where
  omg (Rat (a, b))          = Rat (b, a)
```

De hele indeling in vijf klassen was erom begonnen dat er typen zijn die niet tot alle klassen behoren. Lijsten zijn daar een voorbeeld van: de enige van de vijf klassen waarvan lijsten een instance zijn, is de *Monoid*. De rol van de operator wordt gespeeld door lijstconcatenatie (++), wat immers een associatieve operator is met een neutraal element:

```
instance Eq a ⇒ Monoid [a] where
  (<+>) = (++)
  0     = [ ]
```

Lijsten vormen geen groep, omdat er geen functie *neg* op lijsten gedefinieerd kan worden met de gewenste eigenschap.

Het type *Bool* vormt een *Ring*. De rol van de operator <+> wordt daarbij gespeeld door de ongelijkheids-operator (een voorbeeld van het feit dat je bij <+> niet direct aan optelling hoeft te denken). De 'and'-operator speelt de rol van <*>. De instance-declaraties luiden:

```
instance Monoid Bool where
  (<+>) = (≢)
  0     = False
instance Groep Bool where
  neg   = id
```

```
instance Ring Bool where
  (<*>) = (∧)
  1     = True
```

Een ander voorbeeld van een ring zijn lijsten van integers. De operaties worden daarbij 'puntsgewijs' uitgevoerd door middel van *zipWith* (voor operatoren met twee parameters), *map* (voor operatoren met één parameter) en *repeat* (voor constanten). De instance-declaraties luiden:

```
instance Monoid [Int] where
  (<+>) = zipWith (+)
  0     = repeat 0
instance Groep [Int] where
  neg   = map negate
instance Ring [Int] where
  (<*>) = zipWith (*)
  1     = repeat 1
```

We kunnen niet op deze manier doorgaan, en lijsten van integers tot Euclidische ring maken. De gedefinieerde operatoren voldoen namelijk niet aan wet D2. Wel kunnen de declaraties nog wat algemener gemaakt worden. De elementen van de lijst hoeven namelijk niet per se integers te zijn; het is voldoende als het element-type een instance is van *Ring*:

```
instance Monoid a ⇒ Monoid [a] where
  (<+>) = zipWith (<+>)
  0     = repeat 0
instance Groep a ⇒ Groep [a] where
  neg   = map neg
instance Ring a ⇒ Ring [a] where
  (<*>) = zipWith (<*>)
  1     = repeat 1
```

Met deze declaratie zijn bijvoorbeeld lijsten van *Bool*'s een ring, en daarom ook lijsten van lijsten van *Bool*'s, enzovoort. Het enige probleem van deze declaratie is dat hij zich niet verdraagt met de eerder gegeven instance-declaratie voor *Monoid* [*a*]. Wat zou immers het resultaat moeten zijn van $[1,2] <+> [3,4]$ – de concatenatie van de twee lijsten $[1,2,3,4]$, of de puntsgewijze optelling $[4,6]$? Als beide instance-declaraties in één programma staan, is het gevolg daarom een foutmelding.

Polynoomringen

blz. 108

In paragraaf 7 werd een datastructuur *Poly* gedefinieerd, waarmee polynomen beschreven kunnen worden:

```
data Poly  = Poly [Term]
data Term = Term (Float, Int)
```

blz. 111

In paragraaf 7 werden vervolgens een aantal functies gedefinieerd: *pPlus* om polynomen op te tellen, *pNeg* om het tegenovergestelde van een polynoom te bepalen, en *pMaal* om polynomen te vermenigvuldigen. Met deze drie functies vormen de polynomen een *Ring*:

```
instance Monoid Poly where
  (<+>) = pPlus
  0     = Poly []
instance Groep Poly where
```

```
  neg    = pNeg
instance Ring Poly where
  (<*>)  = pMaal
  1      = Poly [Term (1.0, 0)]
```

Nagegaan kan worden, dat de genoemde operaties inderdaad aan de vereiste eigenschappen voldoen.

Een polynoom is een lijst termen, waarbij elke term bestaat uit een *coëfficiënt* en een *exponent*. Een voorbeeld van een term is $1.5x^3$. Dat de exponent een geheel getal is (zelfs een natuurlijk getal), is wezenlijk in een polynoom. Maar waarom zou de coëfficiënt een *Float* moeten zijn? Er zijn evengoed polynomen denkbaar met integers als coëfficiënten, of met complexe getallen.

Door naar de definities van de functies op polynomen te kijken, is het te achterhalen welke eigenschappen van de coëfficiënten er precies nodig zijn. In de functie *pPlus* wordt de functie *pEenvoud* gebruikt. In deze functie (gedefinieerd in paragraaf 7) worden coëfficiënten opgeteld en met nul vergeleken. In de functie *pNeg* wordt de functie *tNeg* gebruikt, die van een coëfficiënt het tegenovergestelde bepaalt. In de functie *pMaal* tenslotte wordt de functie *tMaal* gebruikt, die coeëfficiënten vermenigvuldigt. Nergens worden coëfficiënten op elkaar gedeeld. Oftewel: het is voldoende als de coëfficiënten een *Ring* vormen. Een Euclidische ring, of zelfs een lichaam, is dus niet vereist. blz. 110

De vrijheid in de keuze van het type van de coëfficënten komt tot uitdrukking in een nieuwe definitie van polynomen:

```
data Poly a  = Poly [Term a]
data Term a = Term (a, Int)
```

Dit is een polymorfe definitie van *Poly a*, uit te spreken als 'polynomen met coëfficiënten in *a*'.

Een type *Poly a* is een instance van *Ring* als het type *a* dat is. Dit komt tot uiting in de voorwaardelijke instance-declaratie

```
instance Ring a => Ring (Poly a) where ...
```

De definities van de functies op polynomen kunnen aangepast worden, zodat ze werken in een willekeurige ring. Eerst de hulpfuncties op termen (vergelijk de overeenkomstige definities in paragraaf 7): blz. 111

```
tNeg                  :: Groep a => Term a -> Term a
tNeg (Term (c, e)) = Term (neg c, e)
tMaal                                  :: Ring a => Term a -> Term a -> Term a
tMaal (Term (c1, e1)) (Term (c2, e2)) = Term (c1 <*> c2, e1 + e2)
```

Ook de functie *pEenvoud* kan zo aangepast worden (opgave 9.5). blz. 151

De nieuwe aangepaste definities van de functies op polynomen kunnen direct in de instance-declaraties worden gegeven:

```
instance Monoid a => Monoid (Poly a) where
  Poly xs <+> Poly ys = pEenvoud (Poly (xs ++ ys))
  0                   = Poly []
instance Groep a => Groep (Poly a) where
  neg (Poly xs)       = Poly (map tNeg xs)
instance Ring a => Ring (Poly a) where
  Poly xs <*> Poly ys  = pEenvoud (Poly (cpWith tMaal xs ys))
  1                    = Poly [Term (1, 0)]
```

De relevante afgeleide functies uit paragraaf 10, zoals $\ominus$ en $< \backslash macht >$, krijgen we kado. De functie *kwadraat* bijvoorbeeld werkt immers op een willekeurige instance van *Ring*, dus ook op polynomen. In een sessie kan bijvoorbeeld het polynoom dat het resultaat is van $(x+1)^4$ berekend worden: blz. 143

```
? (Poly [ Term(1,1), Term(1,0) ]) <^> 4
Poly [ Term(1,4), Term(4,3), Term(6,2), Term(4,1), Term(1,0) ]
```

(Dat klopt, want $(x+1)^4$ is $x^4+4x^3+6x^2+4x+1$).

Polynomen als Euclidische ring

Het is mogelijk om op polynomen een 'deling met rest' uit te voeren. De graad van het polynoom
blz. 111
(de functie *pGraad* uit paragraaf 7) vervult daarbij de rol van *orde*-functie. Een voorbeeld is de volgende deling:

$$\frac{2x^4+5x^3+4x^2-3x+2}{x^2+x+1}$$

Het quotiënt van deze deling is $2x^2+3x-1$, en de rest is $-5x+3$. Deze uitkomsten voldoen aan de vereiste wetten E1 (reken maar na) en E2 (de graad van de rest (1) is kleiner dan de graad van de noemer (2)).

Voor het algoritme en de daarvoor benodigde voorwaarden bekijken we eerst hoe het voorbeeldresultaat berekend wordt. De polynomen kunnen gedeeld worden met een soort staartdeling:

$$\begin{array}{r}
x^2{+}x{+}1 \,/\, 2x^4{+}5x^3{+}4x^2{-}3x{+}2 \,\backslash\, 2x^2{+}3x{-}1 \\
\underline{2x^4{+}2x^3{+}2x^2} \\
3x^3{+}2x^2{-}3x \\
\underline{3x^3{+}3x^2{+}3x} \\
-x^2{-}6x{+}2 \\
\underline{-x^2\ -x{-}1} \\
-5x{+}3
\end{array}$$

Om te beginnen wordt gekeken naar de eerste term van de teller ($2x^4$) en de eerste term van de noemer (x^2). Deze twee worden door elkaar gedeeld, en dat levert de eerste term van het resultaat ($2x^2$). Vervolgens wordt deze $2x^2$ vermenigvuldigd met de complete noemer (x^2+x+1). Het product ($2x^4{+}2x^3{+}2x^2$) wordt afgetrokken van de teller. De term met de hoogste exponent ($2x^4$) is daarmee verdwenen. Het procédé wordt herhaald met het overgebleven deel ($3x^3{+}2x^2{-}3x{+}2$). Dat levert de tweede term van het resultaat ($3x$). De herhaling gaat door, totdat de eerste coëfficiënt van de teller een lagere graad heeft dan de noemer (in het voorbeeld $-5x+3$). Dat is de gezochte rest, en inmiddels is ook het hele quotiënt opgebouwd.

In dit algoritme wordt gebruik gemaakt van het vermenigvuldigen en aftrekken van polynomen. Dat is geen probleem, want polynomen vormen een ring. Maar daarnaast is het nodig dat termen door elkaar gedeeld kunnen worden (bijvoorbeeld $3x^3/x^2=3x$). Daarvoor moeten de coëfficiënten door elkaar gedeeld worden, en de exponenten afgetrokken. Deze deling moet exact geschieden (dus zonder rest). Conclusie:

Polynomen vormen een Euclidische ring,
mits de coëfficiënten een lichaam vormen.

Deze voorwaarde wordt in onderstaande instance-declaratie gesteld. Het gebruikte algoritme verloopt precies zoals in het behandelde voorbeeld: eerst wordt een polynoom h bepaald, die één term van het resultaat bevat. Daarbij wordt het resultaat van een recursieve aanroep opgeteld. In de recursieve aanroep wordt de teller (f) verminderd met h keer de noemer (g).

```
instance Lichaam a => Euclid (Poly a) where
  orde (Poly [])                = -1
  orde (Poly (Term (c,e) : ts)) = e
  quot f g  | n < m             = 0
            | n >= m            = h <+> quot (f <-> h <*> g) g
    where
```

$$
\begin{aligned}
n &= orde\ f \\
m &= orde\ g \\
h &= Poly\ [\,Term\ (coef1\ f <//> coef1\ g, n - m)] \\
&\qquad coef1\ (Poly\ (Term\ (c, e) : ts)) = c
\end{aligned}
$$

Voor de functie *rest* kan de default-definitie uit de klasse-declaratie gebruikt worden.

Omdat de rationale getallen (*Ratio*) een lichaam vormen, kan van polynomen met rationale getallen als coëfficiënten nu ook het quotiënt en de rest bepaald worden. Er zijn geen nieuwe functie-definities nodig; door de typering weet de interpreter alle gegeven definities op de juiste manier te combineren. Zo heeft bijvoorbeeld de deling

$$\frac{\frac{1}{3}x^3 + \frac{1}{5}x - \frac{1}{2}}{\frac{1}{2}x + 1}$$

als quotiënt het polynoom $\frac{2}{3}x^2 - \frac{4}{3}x + \frac{46}{15}$, en als rest het polynoom met alleen de constante term $-\frac{107}{30}$.

Doordat de polynomen nu een Euclidische ring vormen, krijgen we de functie *ggd* kado. Wel moeten de coëfficiënten van de polynoom daarvoor een lichaam vormen.

Quotiëntenlichamen

De rationale getallen vormen een lichaam, zoals gedefinieerd in paragraaf 10. Net als de definitie van polynomen kan deze definitie gegeneraliseerd worden. De teller en de noemer van een breuk hoeven immers geen integer te zijn. Als je naar de definitie van de functies op rationale getallen kijkt, zie je wat van de teller en noemer verwacht wordt: ze moeten optelbaar en vermenigvuldigbaar zijn. Nergens worden ze meegegeven aan functies die per se een integer verwachten, zoals *take* of *repeat*. Teller en noemer van een breuk mogen dus als type een willekeurige instance van *Ring* hebben.

blz. 145

Deze generalisatie van de rationale getallen noemen we *Quot a*: dit is het polymorfe type 'breuken met teller en noemer van type *a*'. Dit type kan instance gemaakt worden van alle vijf de klassen uit deze sectie (*Monoid*, *Groep*, *Ring*, *Euclid* en *Lichaam*), en bovendien van de klasse *Eq*. Voorwaarde is steeds dat het type van teller en noemer een ring is. Zelfs voor het *Eq*-zijn van het quotiënttype geldt deze voorwaarde al: om te bepalen of twee breuken gelijk zijn moeten ze immers kruiselings vermenigvuldigd worden.

De complete definitie van quotiëntenlichamen is als volgt (dit is een rechtstreekse generalisatie van de rationale getallen):

```
data Quot a                    = Quot (a, a)
instance Ring a => Eq (Quot a) where
  Quot (a, b) == Quot (c, d)   = a <*> d == c <*> b
instance Ring a => Monoid (Quot a) where
  Quot (a, b) <+> Quot (c, d) = Quot (a <*> d <+> c <*> b, b <*> d)
  0 = Quot (0, 1)
instance Ring a => Groep (Quot a) where
  neg (Quot (a, b))            = Quot (neg a, b)
instance Ring a => Ring (Quot a) where
  Quot (a, b) <*> Quot (c, d)  = Quot (a <*> c, b <*> d)
  1                            = Quot (1, 1)
instance Ring a => Lichaam (Quot a) where
  omg (Quot (a, b))            = Quot (b, a)
```

De definitie van *Euclid* (*Quot a*) is hier maar weggelaten, omdat die (net als bij *Float*) triviaal is.

Omdat quotiënten van elementen van een ring een lichaam vormen, spreken we van 'quotiëntlichamen'. Zo bestaat er bijvoorbeeld het lichaam 'quotiënten van polynomen met integers als coëfficiënten'.

Matrixringen

blz. 98 Ook de definitie van matrices uit sectie 7 kan gegeneraliseerd worden. De elementen van een matrix hoeven niet per se *Float*'s te zijn; het kunnen elementen van een willekeurige ring zijn. Alleen voor de matrix-inverteerfunctie is lichaam-zijn van de elementen vereist.

Als de elementen in een matrix een ring zijn, vormen de matrices zelf ook een ring. Daarbij speelt matrixvermenigvuldiging de rol van $\ll*\gg$, en de identiteitsmatrix de rol van $\mathbf{1}$. Aan de voorwaarden is voldaan. Zo is wet R1 bijvoorbeeld geldig omdat matrixvermenigvuldiging werd gedefinieerd als het na elkaar toepassen van lineaire afbeeldingen. Het na elkaar toepassen van afbeeldingen is
blz. 50 associatief, zoals in opgave 3.4 werd bewezen.

De matrix-functies kunnen eenvoudig worden aangepast, zodat de volgende declaraties kunnen worden gedaan:

```
data Matrix a = Mat [[a]]
instance Eq a       ⇒ Eq       (Matrix a) where ...
instance Monoid a ⇒ Monoid (Matrix a) where ...
instance Ring a     ⇒ Ring     (Matrix a) where ...
```

Door matrices in te passen in de klasse-hiërarchie komen er allerlei nieuwe mogelijkheden beschikbaar. Er kunnen bijvoorbeeld berekeningen gedaan worden met matrices van polynomen met
blz. 151 complexe getallen als coëfficiënten (zie opgave 9.8 voor een toepassing hiervan). En wat te denken van integer-matrices als coëfficiënten van polynomen? En daar weer het quotiëntlichaam van... de mogelijkheden zijn onbegrensd.

Opgaven

10.1 Bewijs dat in elke ring geldt: $a*(b-c) = a*b - a*c$, en $a*0 = 0$. Gebruik de definitie van
blz. 143 $-$ uit paragraaf 10. Let goed op dat je alleen maar de voor ringen geldende wetten gebruikt!

blz. 144 **10.2** In de laatste zin van paragraaf 10 is sprake van een 'foutmelding'. Welke foutmelding wordt bedoeld?

10.3 Beschouw het type *Klein*, dat gedefinieerd is als:

```
data Klein = E | A | B | C
```

(genoemd naar de Duitse wiskundige Felix Klein). Dit type wordt een instance van *Groep* gemaakt, door:

```
instance Monoid Klein where
  (<+>) = f
  0     = E
instance Groep Klein where
  neg   = id
```

Hoe kan de functie f worden gedefinieerd?

10.4 In welke klasse zou de functie *fromInteger* geplaatst moeten worden: *Monoid*, *Groep*, *Ring*, *Euclid* of *Lichaam*? Geef een default-definitie.

blz. 110
blz. 146

10.5 Welke wijzigingen zijn er nodig in de functie *pEenvoud* uit paragraaf 7 om te werken voor polynomen met coëfficiënten in een willekeurige ring, zoals beschreven in paragraaf 10? Wat wordt het type daardoor?

10.6 Wat gebeurt er als de functie *quot* in de instance-declaratie van polynomen als *Euclid* (paragraaf 10), als tweede parameter het nul-polynoom meekrijgt? blz. 148

10.7 Geef een zinvolle definitie van $(a \to a)$ als instance van *Monoid* (stoor je even niet aan het feit dat functies geen instance zijn van *Eq*). Welke deelverzameling van $(Float \to Float)$ is een instance van *Groep*?

10.8 Bij het oplossen van differentiaalvergelijkingen (zoals die in de natuurkunde veel voorkomen) spelen *eigenwaardes* een belangrijke rol. Eigenwaardes zijn als volgt gedefinieerd. Als A een vierkante matrix is, dan heet k een eigenwaarde van A als voor alle vectoren v geldt: $A@v = k \cdot v$. (Met '@' wordt hier de functie *matApply* bedoeld, en met '$\cdot$' de functie *vecScale*). Schrijf een functie die gegeven een matrix de vergelijking bepaalt waaraan de eigenwaardes moeten voldoen.

Hoofdstuk 11

Huffman-codering

Bij het verzenden van grote hoeveelheden gegevens, zoals bijvoorbeeld door een faxmachine, 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.

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`. ∎

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.

Frequenties

In onze Haskell-implementatie van Huffman-codering zullen we boodschappen eenvoudigweg representeren als lijsten van lettertekens:

$$\begin{aligned} &\textbf{type } \mathit{Symbol} \;\; = \mathit{Char} \\ &\textbf{type } \mathit{Message} = [\mathit{Symbol}]. \end{aligned}$$

Gebruikmakend van een tweetal functies uit de standaardmodule *Data.List*,

$$\begin{aligned} \mathit{sort} \;\; &:: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\ \mathit{group} &:: (\mathit{Eq}\ a) \;\; \Rightarrow [a] \to [[a]], \end{aligned}$$

waarbij *group* de functie is die opeenvolgende gelijken in een lijst samenvoegt, als in bijvoorbeeld

$$\mathit{group}\ \texttt{"onmiddellijk"} \rightsquigarrow [\texttt{"o"}, \texttt{"n"}, \texttt{"m"}, \texttt{"i"}, \texttt{"dd"}, \texttt{"e"}, \texttt{"ll"}, \texttt{"i"}, \texttt{"j"}, \texttt{"k"}],$$

kunnen we nu het alfabet en de frequenties van een boodschap bepalen met

$$\begin{array}{l} \mathit{freqs} :: \mathit{Message} \to [(\mathit{Symbol}, \mathit{Int})] \\ \mathit{freqs} = \mathit{map}\ \mathit{freq} \circ \mathit{group} \circ \mathit{sort} \\ \quad \textbf{where} \\ \quad\quad \mathit{freq}\ \mathit{syms}@(\mathit{sym} : \_) = (\mathit{sym}, \mathit{length}\ \mathit{syms}). \end{array}$$

Voor de boodschap `text` hebben we bijvoorbeeld

$$\mathit{sort}\ \texttt{"text"} \rightsquigarrow \texttt{"ettx"}$$

en

$$\mathit{group}\ \texttt{"ettx"} \rightsquigarrow [\texttt{"e"}, \texttt{"tt"}, \texttt{"x"}],$$

en dus

$$\mathit{freqs}\ \texttt{"text"} \rightsquigarrow [(\texttt{'e'}, 1), (\texttt{'t'}, 2), (\texttt{'x'}, 1)].$$

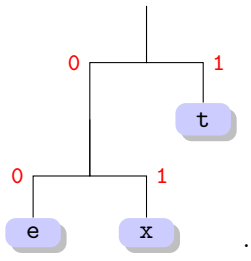
Merk op dat de gegeven implementatie van de functie *freqs* nog ruimte voor verbetering biedt: na het sorteren van de boodschap wordt een groepering opgebouwd, die vervolgens meteen weer afgebroken wordt om de lengte van de groepen te bepalen. Het is iets efficiënter om deze operaties samen te voegen tot een enkele gang over de lijst met behulp van een lokale functie *collate*:

$$\begin{array}{l} \mathit{freqs} :: \mathit{Message} \to [(\mathit{Symbol}, \mathit{Int})] \\ \mathit{freqs} = \mathit{collate} \circ \mathit{sort} \\ \quad \textbf{where} \\ \quad\quad \begin{array}{ll} \mathit{collate}\ [\,] & = [\,] \\ \mathit{collate}\ [\mathit{sym}] & = [(\mathit{sym}, 1)] \\ \mathit{collate}\ (\mathit{sym}_1 : \mathit{syms}@(\mathit{sym}_2 : \_)) & \\ \quad \mid \mathit{sym}_1 \equiv \mathit{sym}_2 & = \textbf{let}\ (\mathit{sym}, n) : \mathit{frqs} = \mathit{collate}\ \mathit{syms} \\ & \quad\ \textbf{in}\ \ (\mathit{sym}, n+1) : \mathit{frqs} \\ \quad \mid \mathit{otherwise} & = (\mathit{sym}_1, 1) : \mathit{collate}\ \mathit{syms}. \end{array} \end{array}$$

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 $\{\texttt{t}, \texttt{e}, \texttt{x}\}$, 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,

$$\textbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Symbol} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Tree}.$$

De hierboven afgebeelde boom laat zich dan schrijven als

$$\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ \texttt{'e'})\ (\mathit{Leaf}\ \texttt{'x'}))\ (\mathit{Leaf}\ \texttt{'t'}).$$

Coderen

Bitreeksen worden gerepresenteerd als lijsten van waarden van het type *Bit*:

$$\begin{aligned}&\textbf{data}\ \mathit{Bit}\ \ = \mathit{Zero} \mid \mathit{One}\\ &\textbf{type}\ \mathit{Bits} = [\mathit{Bit}].\end{aligned}$$

Gegeven een Huffman-boom kunnen we voor een symbool de bijbehorende bitreeks als volgt bepalen:

$$\begin{aligned}
&\mathit{encode}_{\mathit{Symbol}} :: \mathit{Tree} \to \quad \mathit{Symbol} \to \mathit{Bits}\\
&\mathit{encode}_{\mathit{Symbol}}\ \ \mathit{tree}\ \ \ \ \ \mathit{sym}\ \ \ = \textbf{case}\ \mathit{enc}\ \mathit{tree}\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{undefined}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ \mathit{bits} \to \mathit{bits}\\
&\quad \textbf{where}\\
&\qquad \mathit{enc}\ (\mathit{Leaf}\ \mathit{sym}')\\
&\qquad\quad \mid \mathit{sym} \equiv \mathit{sym}' = \mathit{Just}\ [\,]\\
&\qquad\quad \mid \mathit{otherwise}\ \ \ = \mathit{Nothing}\\
&\qquad \mathit{enc}\ (\mathit{Node}\ l\ r)\ \ = \textbf{case}\ \mathit{enc}\ l\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \textbf{case}\ \mathit{enc}\ r\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ \mathit{bits} \to \mathit{Just}\ (\mathit{One} : \mathit{bits})\\
&\qquad\qquad\qquad\qquad\quad \mathit{Just}\ \mathit{bits} \to \mathit{Just}\ (\mathit{Zero} : \mathit{bits}).
\end{aligned}$$

Of, iets korter, zonder tussenkomst van het *Maybe*-type,

$$\begin{aligned}
&\mathit{encode}_{\mathit{Symbol}} :: \mathit{Tree} \to \mathit{Symbol} \to \mathit{Bits}\\
&\mathit{encode}_{\mathit{Symbol}}\ \ \mathit{tree}\ \ \ \ \mathit{sym}\ \ \ \ = \mathit{enc}\ \mathit{tree}\ \mathit{id}\ \mathit{undefined}\\
&\quad \textbf{where}\\
&\qquad \mathit{enc}\ (\mathit{Leaf}\ \mathit{sym}')\ \mathit{prefix}\ \mathit{bits}\\
&\qquad\quad \mid \mathit{sym} \equiv \mathit{sym}' \qquad\quad = \mathit{prefix}\ [\,]\\
&\qquad\quad \mid \mathit{otherwise} \qquad\quad = \mathit{bits}\\
&\qquad \mathit{enc}\ (\mathit{Node}\ l\ r)\ \ \mathit{prefix}\ \mathit{bits} = \mathit{enc}\ l\ (\mathit{prefix} \circ (\mathit{Zero}:))\ (\mathit{enc}\ r\ (\mathit{prefix} \circ (\mathit{One}:))\ \mathit{bits})
\end{aligned}$$

en zelfs

$$\begin{aligned}
&\mathit{encode}_{\mathit{Symbol}} :: \mathit{Tree} \to \mathit{Symbol} \to \mathit{Bits}\\
&\mathit{encode}_{\mathit{Symbol}}\ \ \mathit{tree}\ \ \ \ \mathit{sym}\ \ \ \ = \mathit{enc}\ \mathit{tree}\ \mathit{id}\ \mathit{undefined}\\
&\quad \textbf{where}\\
&\qquad \mathit{enc}\ (\mathit{Leaf}\ \mathit{sym}')\ \mathit{prefix}\\
&\qquad\quad \mid \mathit{sym} \equiv \mathit{sym}' \qquad\quad = \mathit{const}\ (\mathit{prefix}\ [\,])\\
&\qquad\quad \mid \mathit{otherwise} \qquad\quad = \mathit{id}\\
&\qquad \mathit{enc}\ (\mathit{Node}\ l\ r)\ \ \mathit{prefix} = \mathit{enc}\ l\ (\mathit{prefix} \circ (\mathit{Zero}:)) \circ \mathit{enc}\ r\ (\mathit{prefix} \circ (\mathit{One}:)).
\end{aligned}$$

Merk op dat als het te coderen symbool niet in de boom voorkomt, we de speciale foutwaarde *undefined* ('ongedefinieerd') opleveren. 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:

$$\begin{aligned} &encode :: Tree \to Message \to Bits \\ &encode \quad tree \;=\; concatMap \circ encode_{Symbol}. \end{aligned}$$

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 11.1 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

$$f_{\mathtt{i}} = f_{\mathtt{d}} = f_{\mathtt{l}} = 2.$$

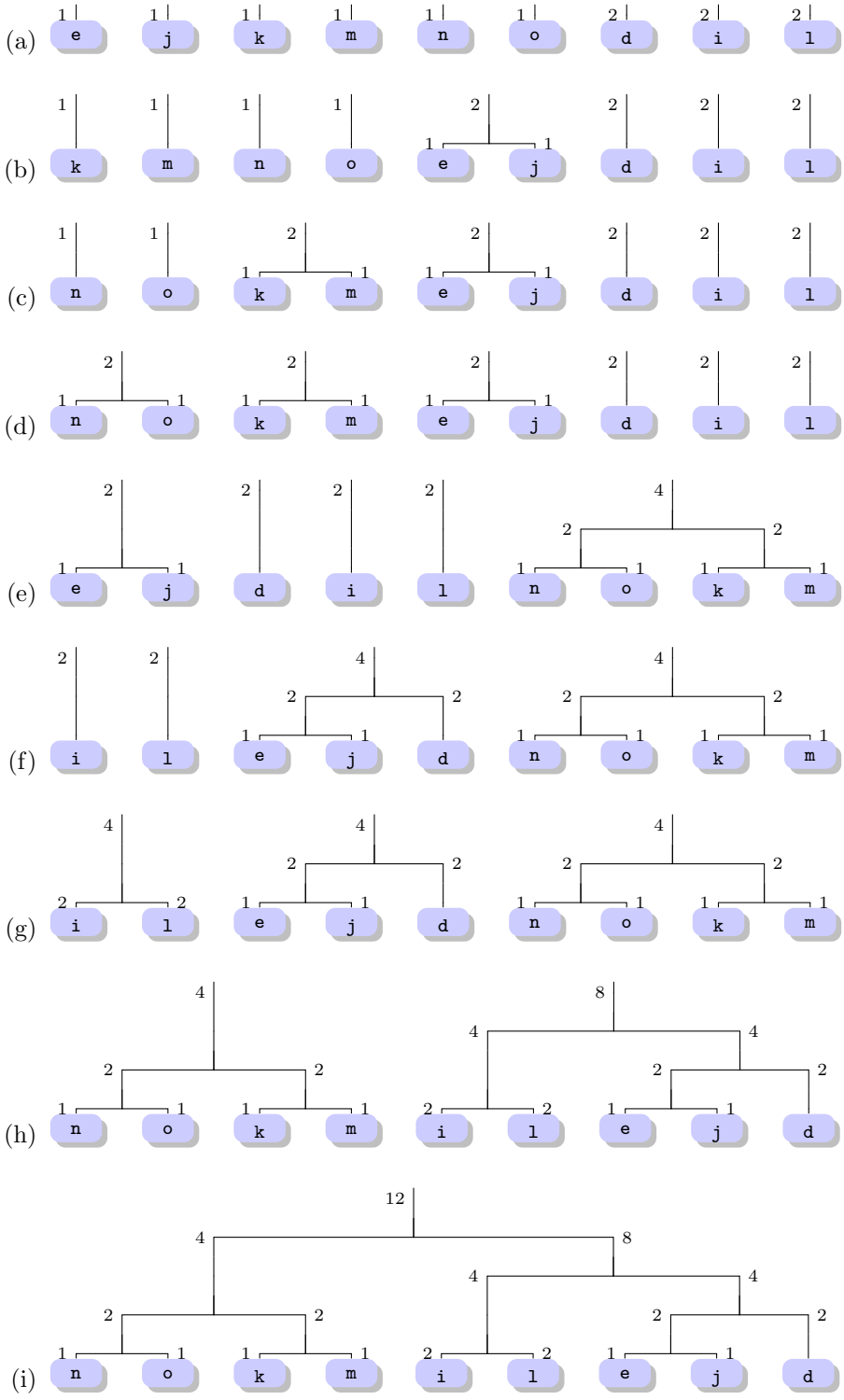
In deelfiguur 11.1(a) is te zien hoe voor elk symbool een blad aangemaakt is. In deelfiguur 11.1(b)–(i) worden vervolgens telkens de twee bomen met het laagste totale frequentie samengevoegd tot een nieuwe boom. In elk van de deelfiguren zijn de bomen gesorteerd op oplopend frequentietotaal en zijn alle 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.
```

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

$$sortBy :: (a \to a \to Ordering) \to [a] \to [a],$$



Figuur 11.1: Constructie van een Huffman-boom

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*:

$$\mathit{compare} :: (\mathit{Ord}\ a) \Rightarrow a \to a \to \mathit{Ordering}.$$

De ordening op de paren in de frequentietabel wordt gemaakt door een hulpfunctie *on*,

$$\begin{array}{lll} \mathit{on} :: (b \to b \to c) \to (a \to b) \to a \to a \to c & & \\ \mathit{on} \quad (\oplus) \qquad\qquad f \qquad\qquad x \quad y = f\ x \oplus f\ y, & & \end{array}$$

los te laten op de functies *snd* en *compare*.

Van alle paren (sym, 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 (l, n_l) en (r, n_r) van de lijst van boom-frequentieparen en voegt ze samen tot een nieuwe boom *Node l r*, die getupeld wordt met zijn frequentietotaal $n_l + n_r$. Dit nieuwe boom-frequentiepaar wordt op de juiste plaats in de lijst *pairs* van overgebleven bomen en frequenties opgeslagen. Hiervoor gebruiken we de functie

$$\mathit{insertBy} :: (a \to a \to \mathit{Ordering}) \to a \to [a] \to [a]$$

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 boom-frequentiepaar 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

$$\mathit{encoding} :: \mathit{Message} \to \mathit{Bits}$$

die een boodschap codeert in een minimale prefixvrije symbool-voor-symboolcodering.

Opgave 2. Schrijf een functie

$$\mathit{cost} :: [(\mathit{Symbol}, \mathit{Int})] \to \mathit{Tree} \to \mathit{Int}$$

die, gegeven een frequentietabel, de kosten van een Huffman-boom bepaalt (cf. de sommatie $(*)$ op pagina 154).

Opgave 3. Schrijf een functie

$$decode :: Tree \to Bits \to Message$$

die, gegeven een Huffman-boom, en een gecodeerde boodschap de oorspronkelijke boodschap produceert. (Hint: definieer eerst een functie $decodes :: Tree \to Bits \to (Symbol, Bits)$ die een enkel symbool decodeert en dat symbool samen met de overgebleven bits oplevert.)

Opgave 4. Geef een definitie voor de functie *group* (pagina 154).

Opgave 5.

(i) Geef een definitie voor de functie *sortBy* (pagina 157).

(ii) Geef een definitie van de functie $sort :: (Ord\ a) \Rightarrow [a] \to [a]$ die gebruikmaakt van *sortBy*.

(iii) Geef een definitie voor de functie *insertBy* (pagina 159).

(iv) Geef een definitie van de functie $insert :: (Ord\ a) \Rightarrow a \to [a] \to [a]$ die gebruikmaakt van *insertBy*.

Opgave 6.

(i) Geef een type voor de lokale functie *freq* (pagina 155).

(ii) Geef een type voor de lokale functie *collate* (pagina 155).

(iii) Geef een type voor de lokale functie *huff* (pagina 157).

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 $Tree \to [Bits]$. 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 $Tree \to Symbol \to Maybe\ Bits$, 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 schrijf een versie van type $Tree \to Message \to Maybe\ Bits$.

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*:

$$\textbf{type}\ Message\ a = [a].$$

(ii) Pas de implementatie zo aan dat boodschappen kunnen bestaan uit symbolen van een willekeurig type uit de klasse *Eq*.

Hoofdstuk 12

Case Study: Symbolische berekeningen

Rekenkundige expressies

Data-declaraties 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 data-declaratie:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = {} & \mathit{Con}\ \mathit{Float}\\ \mid {} & \mathit{Var}\ \mathit{String}\\ \mid {} & \mathit{Expr}{:}+{:}\ \mathit{Expr}\\ \mid {} & \mathit{Expr}\ {:}-{:}\ \mathit{Expr}\\ \mid {} & \mathit{Expr}\ {:}*{:}\ \mathit{Expr}\\ \mid {} & \mathit{Expr}\ {:}/{:}\ \mathit{Expr}\end{aligned}$$

De data-declaratie 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 infix-operator, 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 data-declaratie ook op een dubbele punt. De drie symbolen in :+: vormen één operator, en moeten dan ook zonder spatie ertussen geschreven worden.

De constructor-operatoren uit de data-declaratie kunnen van een prioriteit en een associatievolgorde worden voorzien met behulp van een infix-declaratie. Deze werd ingevoerd in paragraaf 3. blz. 39
Het is het handigste om de operatoren van dezelfde prioriteit te voorzien als de gewone rekenkundige operatoren:

```
infixl 7 :*:
infix  7 :/:
infixl 6 :+:, :-:
```

Na deze declaraties kan bijvoorbeeld de expressie $3x + 4y$ als datastructuur worden gerepresenteerd

door de Haskell-expressie

$$Con\ 3.0 :*: Var\ \texttt{"x"} :+: Con\ 4.0 :*: Var\ \texttt{"y"}$$

Het is belangrijk om onderscheid te maken tussen expressies in Haskell, en rekenkundige expressies in het taaltje dat door de data-declaratie wordt beschreven. Zo is *Var* `"x"` een Haskell-expressie die als waarde de datastructuur heeft, die 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 (*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 = level t
  where leval (Con f)   = f
        leval (Var s)   = env s
        leval (l :+: r) = (leval l) + (leval r)
        leval (l :-: r) = (leval l) - (leval r)
        leval (l :*: r) = (leval l) * (leval r)
        leval (l :/: r) = (leval l) / (leval 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 = replace 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)
```

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 4 de volgende functie werd geschreven: blz. 51

```
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 expressie-bomen en symbolisch differentiëren is dat wèl mogelijk.

De symbolische differentieerfunctie heeft behalve een expressie een *Char* 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.

Andere expressiebomen

Naast de data-declaratie die rekenkundige expressies beschrijft, zijn ook andere soorten expressies te beschrijven met data-declaraties. Onderstaande data-declaratie beschrijft bijvoorbeeld Boolese expressies oftewel *proposities*:

```
data Prop = Cons Bool
          | Vari Char
          | Not Prop
          | Prop :∧: Prop
          | Prop :∨: Prop
          | Prop :⇒: Prop
```

De propositie $b \vee \neg b$ wordt bijvoorbeeld gerepresenteerd door de datastructuur

$$Vari\ \texttt{'b'} :\vee: Not\ (Vari\ \texttt{'b'})$$

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.

$$\begin{aligned}
verw\ (Cons\ b) &= Cons\ b \\
verw\ (Vari\ x) &= Vari\ x \\
verw\ (Not\ p) &= Not\ (verw\ p) \\
verw\ (p :\wedge: q) &= verw\ p :\wedge: verw\ q \\
verw\ (p :\vee: q) &= Not\ (Not\ (verw\ p) :\wedge: Not\ (verw\ q)) \\
verw\ (p :\Rightarrow: q) &= verw\ (Not\ p :\vee: q)
\end{aligned}$$

Met data-declaraties kunnen naast expressies ook taalconstructies uit programmeertalen worden beschreven. Statements uit een Pascal-achtige taal kunnen bijvoorbeeld worden beschreven door de volgende data-declaratie:

$$\begin{aligned}
\textbf{data}\ Stat = {} & Assign\ Char\ Expr \\
\mid {} & If\ Prop\ Stat\ Stat \\
\mid {} & While\ Prop\ Stat \\
\mid {} & Repeat\ Stat\ Prop \\
\mid {} & Compound\ [Stat]
\end{aligned}$$

Door functies te schrijven die op dit soort datastructuren werken, is het mogelijk om Pascal-programma's (althans de boom-representaties daarvan) te transformeren volgens 'rekenregels' die daarvoor gelden.

Stringrepresentatie van een boom

blz. 161

We keren weer terug naar de expressiebomen uit paragraaf 12. De Haskell-expressies 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 $Var\ \texttt{"x"} :*: Var\ \texttt{"x"} :+: Con\ 1.0$. 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"`.

blz. ??

In paragraaf **??** zullen we een functie

$$ontleed :: String \rightarrow Expr$$

schrijven, die zo'n string omzet in de overeenkomstige boomstructuur. Maar eerst bekijken we het omgekeerde probleem: een functie *weerg* die een expressie-boom weergeeft als string. Als beide functies beschikbaar zijn, dan kunnen we bijvoorbeeld de differentieer-functie *afg* 'inpakken' zodat het een functie tussen strings wordt:

$$\begin{aligned}
&afgel && :: String \rightarrow String \rightarrow String \\
&afgel\ exprstr\ dx && = weerg\ (afg\ (ontleed\ exprstr)\ dx)
\end{aligned}$$

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 expressie-manipulatie functies 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 expressie-bomen, en wordt daarom met zes patronen gedefinieerd voor alle constructoren van *Expr*.

$$
\begin{aligned}
&\mathit{weerg} && :: \mathit{Expr} \to \mathit{String}\\
&\mathit{weerg}\ (\mathit{Con}\ n) &&= \mathit{showFloat}\ n\\
&\mathit{weerg}\ (\mathit{Var}\ x) &&= x\\
&\mathit{weerg}\ (a :+: b) &&= \texttt{"("} +\!\!+ \mathit{weerg}\ a +\!\!+ \texttt{"+"} +\!\!+ \mathit{weerg}\ b +\!\!+ \texttt{")"}\\
&\mathit{weerg}\ (a :-: b) &&= \texttt{"("} +\!\!+ \mathit{weerg}\ a +\!\!+ \texttt{"-"} +\!\!+ \mathit{weerg}\ b +\!\!+ \texttt{")"}\\
&\mathit{weerg}\ (a :*: b) &&= \mathit{weerg}\ a +\!\!+ \texttt{"*"} +\!\!+ \mathit{weerg}\ b\\
&\mathit{weerg}\ (a :/: b) &&= \mathit{weerg}\ a +\!\!+ \texttt{"/"} +\!\!+ \mathit{weerg}\ b
\end{aligned}
$$

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 boom-expressie (Var `"a"`$:+:$ Var `"b"`)$:*:$ (Var `"c"`$:+:$ Var `"d"`) worden omgezet in de string `"a+b*c+d"`, die de verkeerde interpretatie heeft.

Opgaven

12.1 Breid de data-declaratie van *Expr* uit zodat er vier nieuwe expressievormen onststaan: 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: $(f \circ g)' = (f' \circ g) * g'$.

12.2 Schrijf een functie *norep* die gegeven een *Stat* alle *Repeat*-statements daaruit verwijdert, door ze te vervangen door een equivalente *While*-statements.

12.3 Kun je een functie *foldExpr* schrijven, met behulp waarvan je de functie *weerg* dan definieert?

12.4 (*)[1]We breiden nu het type van *Expr* uit zodat we er een functioneel programma mee kunnen representeren:

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Con}\ \mathit{Float}\\
\mid {} & \mathit{Var}\ \mathit{String}\\
\mid {} & \mathit{Expr} :+: \mathit{Expr}\\
\mid {} & \mathit{Expr} :-: \mathit{Expr}\\
\mid {} & \mathit{Expr} :*: \mathit{Expr}\\
\mid {} & \mathit{Expr} :/: \mathit{Expr}\\
\mid {} & \mathit{Apply}\ \ \mathit{Expr}\ \ \mathit{Expr}\\
\mid {} & \mathit{Lambda}\ \mathit{String}\ \mathit{Expr}
\end{aligned}
$$

We zien, na enig nadenken, dat we de functie *eval* die we eerder gebruikt hebben niet zomaar kunnen uitbreiden; hoe zouden we immers van een *Lambda* `"x"` ... een waarde van type *Float* kunnen maken. Verander nu eerst de functie *eval* zo dat hij altijd weer iets van type *Expr* oplevert. Kun je hem nu uitbreiden zodat hij expressies van type *Expr* evalueert tot een eenvoudigere vorm?

[1]De opgaven met een (*) zijn niet de gemakkelijkste

12.5 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.

Hoofdstuk 13

Embedded Domain Specific Languages

Introduction

When programming one often addresses some specific domain of discourse; one want to produce graphics, animations, bills and bookkeeping information, music, game plays, HTMl- or XML-based text, etc. There are two approaches for adressing such domain-secicness:

- define a domain-specific language (DSL) for the domain at hand, which contains precisely the basic concepts and composition mechanism we need
- defining a library of classes, functions, methods, and whatever other concept the specific programming language at hand provides.

.

If we take the latter approach and then take look at code that makes heavily use of such a specific library, we notice that most of the code consists of references to notions defined by the library; it is almost as if we are programming in another language, which was especially defined for the problem domain we are adressing. Since this percieved DSL is embedded ina conventional language, we talk in this case about an Embedded Domain Specific Language or EDSL.

As we will see Haskell provides a combination of language constructs which smoothly cooperate to make this concepts of defining and using EDSL's fly. We will demonstrate this by developing an EDSL for describing structure of texts; such descriptions automatically become functions which map texts towards Haskell values. If you are not satisfied by the *read* functions which are implicitly constructed by the **deriving** clauses in your data type definitions, you can roll your own using the library we are about to develop.

A Basic Combinator Parser Library

In this section we describe how to embed grammatical descriptions into the programming language Haskell in such a way that the expressions we write closely resemble context- free grammars, but actually are descriptions of parsers for such languages. This technique has a long history, which can be traced back to the use of recursive descent parsers [**?**], which became popular because of their ease of use, their nice integration with semantic processing, and the absence of the need to

(write and) use an off-line parser generator. Although the concept of a context-free grammar has not been touched upon yet, you may think about them as expressions describing the grammatical diagrams in the appendix.

Just like most normal programming languages, embedded domain specific languages are composed of two things:

(i) a collection of primitive constructs

(ii) ways to compose and name constructs

and when embedding grammars things are no different. The basic grammatical concepts are *terminal* and *non-terminal* symbols, or *terminals* and *non-terminals* for short. They can be combined by *sequential composition* (multiple constructs occurring one after another) or by *alternative composition* (a choice from multiple alternatives).

Note that one may argue that non-terminals are actually not primitive, but result from the introduction of a naming scheme; we will see that in the case of parser combinators, non-terminals are not introduced as a separate concept, but just are Haskell names referring to values which represent parsers.

The Types

Since grammatical expressions will turn out to be normal Haskell expressions, we start by discussing the types involved; and not just the types of the basic constructs, but also the types of the composition mechanisms. For most embedded languages the decisions taken here heavily influence the shape of the library to be defined, its extendability and eventually its success.

Basically, a parser takes a list of symbols and produces a tree. Introducing type variables to abstract from the symbol type s and the tree type t, a first approximation of our *Parser* type is:

$$\textbf{type}\ \mathit{Parser}\ s\ t = [s] \to t$$

Parsers do not need to consume the entire input list. Thus, apart from the tree, they also need to return the part of the input string that was not consumed:

$$\textbf{type}\ \mathit{Parser}\ s\ t = [s] \to (t, [s])$$

The symbol list $[s]$ can be thought of as a *state* that is transformed by the function while building the tree result.

Parsers can be ambiguous: there may be multiple ways to parse a string. Instead of a single result, we therefore have a list of possible results, each consisting of a parser tree and unconsumed input:

$$\textbf{type}\ \mathit{Parser}\ s\ t = [s] \to [(t, [s])]$$

This idea was dubbed by Wadler [**?**] as the "list of successes" method, and it underlies many backtracking applications. An added benefit is that a parser can return the empty list to indicate failure (no successes). If there is exactly one solution, the parser returns a singleton list.

Wrapping the type with a constructor P in a **newtype** definition we get the actual *Parser* type that we will use in the following sections:

$$\begin{aligned}&\textbf{newtype}\ \mathit{Parser}\ s\ t = P\ ([s] \to [(t, [s])])\\ &\mathit{unP}\ (P\ p) = p\end{aligned}$$

Basic Combinators: *pSym*, *pReturn* and *pFail*

As an example of the use of the *Parser* type we start by writing a function which recognises the letter `'a'`: keeping the "list of successes" type in mind we realise that either the input starts with an `'a'` character, in which case we have precisely one way to succeed, i.e. by removing this letter from the input, and returning this character as the witness of success paired with the unused part of the input. If the input does not start with an `'a'` (or is empty) we fail, and return the empty list, as an indication that there is no way to proceed from here:

```
pLettera :: Parser Char Char
pLettera = P (λinp → case inp of
                       (s : ss) | s ≡ 'a' → [('a', ss)]
                       otherwise →          []
             )
```

Of course, we want to abstract from this as soon as possible; we want to be able to recognise other characters than `'a'`, and we want to recognise symbols of other types than *Char*. We introduce our first basic parser constructing function *pSym*:

```
pSym ::    Eq s ⇒ s → Parser s s
pSym a = P (λinp → case inp of
                     (s : ss) | x ≡ a → [(s, ss)]
                     otherwise        → []
           )
```

Since we want to inspect elements from the input with terminal symbols of type s, we have added the *Eq s* constraint, which gives us access to equality ($\equiv$) for values of type s. Note that the function *pSym* by itself is strictly speaking not a parser, but a function which returns a parser. Since the argument is a run-time value it thus becomes possible to construct parsers at run-time.

One might wonder why we have incorporated the value of s in the result, and not the value a? The answer lies in the use of *Eq s* in the type of *Parser*; one should keep in mind that when $\equiv$ returns *True* this does not imply that the compared values are guaranteed to be bit-wise equal. Indeed, it is very common for a scanner –which pre-processes a list of characters into a list of tokens to be recognised– to merge different tokens into a single class with an extra attribute indicating which original token was found. Consider e.g. the following *Token* type:

```
data Token = Identifier       -- terminal symbol used in parser
           | Ident String     -- token constructed by scanner
           | Number Int
           | If_Symbol
           | Then_Symbol
```

Here, the first alternative corresponds to the terminal symbol as we find it in our grammar: we want to see an identifier and from the grammar point of view we do not care which one. The second alternative is the token which is returned by the scanner, and which contains extra information about which identifier was actually scanned; this is the value we want to use in further semantic processing, so this is the value we return as witness from the parser. That these symbols are the same, as far as parsing is concerned, is expressed by the following line in the definition of the function $\equiv$:

```
instance Eq Token where
  (Ident _) ≡ Identifier = True
  ...
```

If we now define:

$$pIdent = pSym\ Identifier$$

we have added a special kind of terminal symbol.

The second basic combinator we introduce in this subsection is *pReturn*, which corresponds to the ϵ-production. The function always succeeds and as a witness returns its parameter; as we will see the function will come in handy when building composite witnesses out of more basic ones. The name was chosen to resemble the monadic *return* function, which injects values into the monadic computation:

$$\begin{aligned} &pReturn :: a \to Parser\ s\ a \\ &pReturn\ a = P\ (\lambda inp \to [(a, inp)]) \end{aligned}$$

We could have chosen to let this function always return a specific value (e.g. ()), but as it will turn out the given definition provides a handy way to inject values into the result of the overall parsing process.

The final basic parser we introduce is the one which always fails:

$$pFail = P\ (const\ [\,])$$

One might wonder why one would need such a parser, but that will become clear in the next section, when we introduce *pChoice*.

Combining Parsers: <*>, <|>, <$> and *pChoice*.

A grammar production usually consists of a sequence of terminal and non-terminal symbols, and a first choice might be to use values of type $[Parser\ s\ a]$ to represent such productions. Since we usually will associate different types to different parsers, this does not work out. Hence we start out by restricting ourselves to productions of length 2 and introduce a special operator <*> which combines two parsers into a new one. What type should we choose for this operator? An obvious choice might be the type:

$$Parser\ s\ a \to Parser\ s\ b \to Parser\ s\ (a, b)$$

in which the witness type of the sequential composition is a pair of the witnesses for the elements of the composition. This approach was taken in early libraries [**?**]. A problem with this choice is that when combining the resulting parser with further parsers, we end up with a deeply nested binary Cartesian product. Instead of starting out with simple types for parsers, and ending up with complicated types for the composed parsers, we have taken the opposite route: we start out with a complicated type and end with a simple type. This interface was pioneered by Röjemo [**?**], made popular through the library described by Swierstra and Duponcheel [**?**], and has been incorporated into the Haskell libraries by McBride and Paterson [**?**]. Now it is know as the *applicative interface*. It is based on the idea that if we have a value of a complicated type $b \to a$, and a value of type b, *we can compose them into a simpler type by applying the first value to the second one*. Using this insight we can now give the type of <*>, together with its definition:

$$\begin{aligned} &(<\!\!*\!\!>) \qquad\qquad :: Parser\ s\ (b \to a) \to Parser\ s\ b \to Parser\ s\ a \\ &P\ p_1 <\!\!*\!\!> P\ p_2 = P\ (\lambda inp \to [(v_1\ v_2, ss_2) \mid (v_1, ss_1) \leftarrow p_1\ inp \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , (v_2, ss_2) \leftarrow p_2\ ss_1 \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad) \end{aligned}$$

The resulting function returns all possible values $v_1\ v_2$ with remaining state ss_2, where v_1 is a witness value returned by parser p_1 with remaining state ss_1. The state ss_1 is used as the starting state for the parser p_2, which in its turn returns the witnesses v_2 and the corresponding final states

ss_2. Note how the types of the parsers were chosen in such a way that the value of type $v_1\ v_2$ matches the witness type of the composed parser.

As a very simple example, we give a parser which recognises the letter 'a' twice, and if it succeeds returns the string `"aa"`:

$$\begin{aligned} pString\_aa = &(pReturn\ (:) <\!\!*\!\!> pLettera) \\ &<\!\!*\!\!> \\ &(pReturn\ (\lambda x \to [x]) <\!\!*\!\!> pLettera) \end{aligned}$$

Let us take a look at the types. The type of $(:)$ is $a \to [a] \to [a]$, and hence the type of $pReturn\ (:)$ is $Parser\ s\ (a \to [a] \to [a])$. Since the type of $pLettera$ is $Parser\ Char\ Char$, the type of $pReturn\ (:) <\!\!*\!\!> pLettera$ is $Parser\ Char\ ([Char] \to [Char])$. Similarly the type of the right hand side operand is $Parser\ Char\ [Char]$, and hence the type of the complete expression is $Parser\ Char\ [Char]$. Having chosen $<\!\!*\!\!>$ to be left associative, the first pair of parentheses may be left out. Thus, many of our parsers will start out by producing some function, followed by a sequence of parsers each providing an argument to this function.

Besides sequential composition we also need *choice*. Since we are using lists to return all possible ways to succeed, we can directly define the operator $<|>$ by returning the concatenation of all possible ways in which either of its arguments can succeed:

$$\begin{aligned} &(<|>) &&:: Parser\ s\ a \to Parser\ s\ a \to Parser\ s\ a \\ &P\ p_1 <|> P\ p_2 = P\ (\lambda inp \to p_1\ inp +\!\!+ p_2\ inp) \end{aligned}$$

Now we have seen the definition of $<|>$, note that $pFail$ is both a left and a right unit for this operator:

$$pFail <|> p \equiv p \equiv p <|> pFail$$

which will play a role in expressions like

$$pChoice\ ps = foldr\ (<|>)\ pFail\ ps$$

One of the things left open thus far is what precedence level these newly introduced operators should have. It turns out that the following minimises the number of parentheses:

$$\begin{aligned} &\textbf{infixl}\ 5\ <\!\!*\!\!> \\ &\textbf{infixr}\ 3\ <|> \end{aligned}$$

As an example to see how this all fits together, we write a function which recognises all correctly nested parentheses – such as `"()(()())"`– and returns the maximal nesting depth occurring in the sequence. The language is described by the grammar $S \to$ `'('` S `')'` $S \mid \epsilon$, and its transcription into parser combinators reads:

$$\begin{aligned} &parens :: Parser\ Char\ Int \\ &parens = pReturn\ (\lambda\_\ b\ \_\ d \to (1 + b)\ `max`\ d) \\ &\qquad <\!\!*\!\!> pSym\ '('\ <\!\!*\!\!> parens\ <\!\!*\!\!> pSym\ ')'\ <\!\!*\!\!> parens \\ &\quad <|> pReturn\ 0 \end{aligned}$$

Since the pattern $pReturn \ldots <\!\!*\!\!>$ will occur quite often, we introduce a third combinator, to be defined in terms of the combinators we have seen already. The combinator <$> takes a function of type $b \to a$, and a parser of type $Parser\ s\ b$, and builds a value of type $Parser\ s\ a$, by applying the function to the witness returned by the parser. Its definition is:

$$\textbf{infix}\ 7\ <\$>$$

$$\begin{aligned} &(<\$>) :: (b \to a) \to (Parser\ s\ b) \to Parser\ s\ a \\ &f <\$> p = pReturn\ f <\!\!*\!\!> p \end{aligned}$$

Using this new combinator we can rewrite the above example into:

```
parens = (λ_ b _ d → (1 + b) `max` d)
           <$> pSym '(' <*> parens <*> pSym ')' <*> parens
       <|> pReturn 0
```

Notice that left argument of the <$> occurrence has type $a \to (Int \to (b \to (Int \to Int)))$, which is a function taking the four results returned by the parsers to the right of the <$> and constructs the result sought; this all works because we have defined <*> to associate to the left.

Although we said we would restrict ourselves to productions of length 2, in fact we can just write productions containing an arbitrary number of elements. Each extra occurrence of the <*> operator introduces an anonymous non-terminal, which is used only once.

Before going into the development of our library, there is one nasty point to be dealt with. For the grammar above, we could have just as well chosen $S \to S$ '(' S ')' $\mid \epsilon$, but unfortunately the direct transcription into a parser would not work. Why not? Because the resulting parser is left recursive: the parser *parens* will start by calling itself, and this will lead to a non-terminating parsing process. Despite the elegance of the parsers introduced thus far, this is a serious shortcoming of the approach taken. Often, one has to change the grammar considerably to get rid of the left-recursion. Also, one might write left-recursive grammars without being aware of it, and it will take time to debug the constructed parser. Since we do not have an off-line grammar analysis, extra care has to be taken by the programmer since the system just does not work as expected, without giving a proper warning; it may just fail to produce a result at all, or it may terminate prematurely with a stack-overflow.

Special Versions of Basic Combinators

As we see in the *parens* program the values witnessing the recognition of the parentheses themselves are not used in the computation of the result. As this situation is very common we introduce specialised versions of <$> and <*>: in the new operators <$, <* and *>, the missing bracket indicates which witness value is not to be included in the result:

```
infixl 3 `opt`
infixl 5 <*  ,  *>
infixl 7 <$

f <$  p = const <$> pReturn f <*> p
p <*  q = const <$> p         <*> q
p  *> q = id      <$  p       <*> q
```

We use this opportunity to introduce two further useful functions, *opt* and *pParens* and reformulate the *parens* function:

```
pParens :: Parser s a → Parser s a
pParens p = id <$  pSym '(' <*> p <*  pSym ')'

opt :: Parser s a → a → Parser s a
p `opt` v  = p <|> pReturn v

parens     = (max ∘ (1+)) <$> pParens parens <*> parens `opt` 0
```

As a final combinator, which we will use in the next section, we introduce a combinator which creates the parser for a specific keyword given as its parameter:

```
pSyms [] = pReturn []
pSyms (x : xs) = (:) <$> pSym x <*> pSyms xs
```

```
module BasicLibrary where
import Prelude hiding (flip, negate)
import Data.Char
newtype Parser s a = P ([s] → [(a, [s])])
unP (P p) = p

pLettera :: Parser Char Char
pLettera =      P (λinp → case inp of
                            (x : xs) | x ≡ 'a' → [('x', xs)]
                            otherwise →          []
                  )

pSym ::    Eq s ⇒ s → Parser s s
pSym a = P      (λinp → case inp of
                            (x : xs) | x ≡ a → [(x, xs)]
                            otherwise        → []
                  )

pSyms [] = pReturn []
pSyms (x : xs) = (:) <$> pSym x <*> pSyms xs

data Token =  Identifier       -- used in parser
            | Ident String     -- constructed by scanner
            | Number Int
            | If_Symbol
            | Then_Symbol

instance Eq Token where
  (Ident _) ≡ Identifier = True

pIdent = pSym Identifier

pReturn :: a → Parser s a
pReturn a = P (λinp → [(a, inp)])

pFail = P (const [])

(<*>)                 :: Parser s (b → a) → Parser s b → Parser s a
P p1 <*> P p2         = P   (λinp → [    (v1 v2      , ss2)        | (v1, ss1) ← p1 inp
                                                                   , (v2, ss2) ← p2 ss1
                                    ]
                            )

(<|>)                 :: Parser s a → Parser s a → Parser s a
P p1 <|> P p2         = P (λinp →   p1 inp ++ p2 inp)

(<$>) :: (b → a) → (Parser s b) → Parser s a
f <$> p = pReturn f <*> p

instance Functor (Parser s) where
  fmap  = (<$>)

infixl 5   <*>
infixr 3   <|>
infix  7   <$>

parens :: Parser Char Int
parens =              (λ_ b _ d → (1 + b) `max` d)
                      <$> pSym '(' <*> parens <*> pSym ')' <*> parens
          <|>         pReturn 0

infixl 3 `opt`
infixl 5   <* , *>
infixl 7   <$
```



```
f <$  p       = const <$> pReturn f <*> p
p <*  q       = const <$> p         <*> q
p  *> q       = id    <$  p         <*> q
```

Case Study: Pocket Calculators

In this section, we develop –starting from the basic combinators introduced in the previous section– a series of pocket calculators, each being an extension of its predecessor. In doing so we gradually build up a small collection of useful combinators, which extend the basic library.

To be able to run all the different versions we provide a small driver function $run :: (Show\ t) \Rightarrow Parser\ Char\ t \to String \to IO\ ()$ in appendix **??**. The first argument of the function *run* is the actual pocket calculator to be used, whereas the second argument is a string prompting the user with the kind of expressions that can be handled. Furthermore we perform a little bit of preprocessing by removing all spaces occurring in the input.

Recognising a Digit

Our first calculator is extremely simple; it requires a digit as input and returns this digit. As a generalisation of the combinator *pSym* we introduce the combinator *pSatisfy*: it checks whether the current input token satisfies a specific predicate instead of comparing it with the expected symbol:

$$
\begin{aligned}
&pDigit = pSatisfy\ (\lambda x \to \texttt{'0'} \leqslant x \wedge x \leqslant \texttt{'9'})\\
&pSatisfy :: (s \to Bool) \to Parser\ s\ s\\
&pSatisfy\ p = P\ (\lambda inp \to \textbf{case}\ inp\ \textbf{of}\\
&\qquad\qquad\qquad\qquad (x : xs) \mid p\ x \to [(x, xs)]\\
&\qquad\qquad\qquad\qquad otherwise\quad\ \to [\,]\\
&\qquad\qquad\qquad)\\
&pSym\ a = pSatisfy\ (\equiv a)
\end{aligned}
$$

A demo run now reads:

```
*Calcs> run pDigit "5"
Give an expression like: 5 or (q) to quit
3
Result is: '3'
Give an expression like: 5 or (q) to quit
a
Incorrect input
Give an expression like: 5 or (q) to quit
q
*Calcs>
```

In the next version we slightly change the type of the parser such that it returns an *Int* instead of a *Char*, using the combinator <$>:

$$
\begin{aligned}
&pDigitAsInt :: Parser\ Char\ Int\\
&pDigitAsInt = (\lambda c \to fromEnum\ c - fromEnum\ \texttt{'0'})\ \texttt{<\$>}\ pDigit
\end{aligned}
$$

Integers: *pMany* and *pMany1*

Since single digits are very boring, let's change our parser into one which recognises a natural number, i.e. a (non-empty) sequence of digits. For this purpose we introduce two new combinators, both converting a parser for an element to a parser for a sequence of such elements. The first one also accepts the empty sequence, whereas the second one requires at least one element to be present:

$$
\begin{aligned}
&pMany, pMany1 :: Parser\ s\ a \to Parser\ s\ [a] \\
&pMany\ p\ \ = (:) <\$> p <\!\!*\!\!> pMany\ p\ `opt`\ [\,] \\
&pMany1\ p = (:) <\$> p <\!\!*\!\!> pMany\ p
\end{aligned}
$$

The second combinator forms the basis for our natural number recognition process, in which we store the recognised digits in a list, before converting this list into the *Int* value:

$$
\begin{aligned}
&pNatural :: Parser\ Char\ Int \\
&pNatural = foldl\ (\lambda a\ b \to a * 10 + b)\ 0 <\$> pMany1\ pDigitAsInt
\end{aligned}
$$

From here it is only a small step to recognising signed numbers. A $-$ sign in front of the digits is mapped onto the function *negate*, and if it is absent we use the function *id*:

$$
\begin{aligned}
&pInteger :: Parser\ Char\ Int \\
&pInteger = (negate <\$\ \ (pSyms\ \texttt{"-"})\ `opt`\ id) <\!\!*\!\!> pNatural
\end{aligned}
$$

More Sequencing: *pChainL*

In our next version, we will show how to parse expressions with infix operators of various precedence levels and various association directions. We start by parsing an expression containing a single + operator, e.g. `"2+55"`. Note again that the result of recognising the + token is discarded, and the operator (+) is only applied to the two recognised integers:

$$
pPlus = (+) <\$> pInteger <\!\!*\ \ pSyms\ \texttt{"+"} <\!\!*\!\!> pInteger
$$

We extend this parser to a parser which handles any number of operands separated by +-tokens. It demonstrates how we can make the result of a parser to differ completely from its "natural" abstract syntax tree.

$$
\begin{aligned}
&pPlus' = applyAll <\$> pInteger <\!\!*\!\!> pMany\ ((+) <\$\ \ pSyms\ \texttt{"+"} <\!\!*\!\!> pInteger) \\
&applyAll :: a \to [\,a \to a\,] \to a \\
&applyAll\ x\ (f : fs)\ \ = \ \ applyAll\ (f\ x)\ fs \\
&applyAll\ x\ [\,]\ \ \ \ \ \ \ = \ \ x
\end{aligned}
$$

Unfortunately, this approach is a bit too simple, since we are relying on the commutativity of + for this approach to work, as each integer recognized in the call to *pMany* becomes the first argument of the (+) operator. If we want to do the same for expressions with $-$ operators, we have to make sure that we *flip* the operator associated with the recognised operator token, in order to make the value which is recognised as second operand to become the right hand side operand:

$$
\begin{aligned}
&pMinus' = applyAll <\$> pInteger <\!\!*\!\!> pMany\ (flip\ (-) <\$\ \ pSyms\ \texttt{"-"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad <\!\!*\!\!> pInteger \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)
\end{aligned}
$$

$$
flip\ f\ x\ y = f\ y\ x
$$

From here it is only a small step to the recognition of expressions which contain both + and $-$ operators:

$$
\begin{aligned}
&pPlusMinus = applyAll <\$> pInteger \\
&\qquad\qquad\qquad\qquad <\!\!*\!\!> pMany\ (\ \ (\ \ flip\ (-) <\$\ \ pSyms\ \texttt{"-"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad <|> \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ flip\ (+) <\$\ \ pSyms\ \texttt{"+"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad)\ \ \ <\!\!*\!\!> pInteger \\
&\qquad\qquad\qquad\qquad\qquad\qquad)
\end{aligned}
$$

Since we will use this pattern often we abstract from it and introduce a parser combinator *pChainL*, which takes two arguments:

(i) the parser for the separator, which returns a value of type $a \to a \to a$

(ii) the parser for the operands, which returns a value of type a

Using this operator, we redefine the function *pPlusMinus*:

$$\begin{aligned}
&pChainL :: Parser\ s\ (a \to a \to a) \to Parser\ s\ a \to Parser\ s\ a \\
&pChainL\ (\oplus)\ p = applyAll\ \texttt{<\$>}\ p\ \texttt{<*>}\ pMany\ (flip\ \texttt{<\$>}\ (\oplus)\ \texttt{<*>}\ p) \\
&pPlusMinus' = ((-)\ \texttt{<\$}\ \ pSyms\ \texttt{"-"}\ \texttt{<|>}\ (+)\ \texttt{<\$}\ \ pSyms\ \texttt{"+"}) \\
&\qquad\qquad \text{`}pChainL\text{`} \\
&\qquad\qquad pInteger
\end{aligned}$$

Left Factoring: *pChainR*, <**> and <??>

As a natural companion to *pChainL*, we would expect a *pChainR* combinator, which treats the recognised operators right-associatively. Before giving its code, we first introduce two other operators, which play an important role in fighting common sources of inefficiency. When we have the parser p:

$$\begin{aligned}
p = \quad & f\ \texttt{<\$>}\ q\ \texttt{<*>}\ r \\
\texttt{<|>}\ & g\ \texttt{<\$>}\ q\ \texttt{<*>}\ s
\end{aligned}$$

then we see that our backtracking implementation may first recognise the q from the first alternative, subsequently can fail when trying to recognise r, and will then continue with recognising q again before trying to recognise an s. Parser generators recognise such situations and perform a grammar transformation (or equivalent action) in order to share the recognition of q between the two alternatives. Unfortunately, we do not have an explicit representation of the underlying grammar at hand which we can inspect and transform [**?**], and without further help from the programmer there is no way we can identify such a common *left-factor*. Hence, we have to do the left-factoring by hand. Since this situation is quite common, we introduce two operators which assist us in this process. The first one is a modification of <*>, which we have named <**>; it differs from <*> in that it applies the result of its right-hand side operand to the result of its left-hand side operand:

$$\begin{aligned}
&(\texttt{<**>}) :: Parser\ s\ b \to Parser\ s\ (b \to a) \to Parser\ s\ a \\
&p\ \texttt{<**>}\ q = (\lambda a\ f \to f\ a)\ \texttt{<\$>}\ p\ \texttt{<*>}\ q
\end{aligned}$$

With help of this new operator we can now transcribe the above example, introducing calls to *flip* because the functions f and g now get their second arguments first, into:

$$p = q\ \texttt{<**>}\ (flip\ f\ \texttt{<\$>}\ r\ \texttt{<|>}\ flip\ g\ \texttt{<\$>}\ s)$$

In some cases, the element s is missing from the second alternative, and for such situations we have the combinator <??>:

$$\begin{aligned}
&(\texttt{<??>}) :: Parser\ s\ a \to Parser\ s\ (a \to a) \to Parser\ s\ a \\
&p\ \texttt{<??>}\ q = p\ \texttt{<**>}\ (q\ \text{`}opt\text{`}\ id)
\end{aligned}$$

Let us now return to the code for *pChainR*. Our first attempt reads:

$$\begin{aligned}
&pChainR\ (\oplus)\ p = id \qquad \texttt{<\$>}\ p \\
&\quad \texttt{<|>} \qquad\qquad flip\ (\$)\ \texttt{<\$>}\ p\ \texttt{<*>}\ (flip\ \texttt{<\$>}\ (\oplus)\ \texttt{<*>}\ pChainR\ (\oplus)\ p)
\end{aligned}$$

which can, using the refactoring method, be expressed more elegantly by:

$$pChainR\ (\oplus)\ p = r\ \textbf{where}\ r = p\ \texttt{<??>}\ (flip\ \texttt{<\$>}\ (\oplus)\ \texttt{<*>}\ r)$$

Two Precedence Levels

Looking back at the definition of *pPlusMinus*, we see still a recurring pattern, i.e. the recognition of an operator symbol and associating it with its semantics. This is the next thing we are going to abstract from. We start out by defining a function that associates an operator terminal symbol with its semantics:

$$pOp\ (sem, symbol) = sem <\$\ \ pSyms\ symbol$$

Our next library combinator *pChoice* takes a list of parsers and combines them into a single parser:

$$pChoice = foldr\ (<|>)\ pFail$$

Using these two combinators, we now can define the collective additive operator recognition by:

$$\begin{aligned} anyOp &= pChoice \circ map\ pOp \\ addops &= anyOp\ [((+), \texttt{"+"}), ((-), \texttt{"-"})] \end{aligned}$$

Since multiplication has precedence over addition, we can now define a new non-terminal *pTimes*, which can only recognise operands containing multiplicative operators:

$$\begin{aligned} pPlusMinusTimes &= pChainL\ addops\ pTimes \\ pTimes &= pChainL\ mulops\ pInteger \\ mulops &= anyOp\ [((*), \texttt{"*"})] \end{aligned}$$

Any Number of Precedence Levels: *pPack*

Of course, we do not want to restrict ourselves to just two priority levels. On the other hand, we are not looking forward to explicitly introduce a new non-terminal for each precedence level, so we take a look at the code, and try to see a pattern. We start out by substituting the expression for *pTimes* into the definition of *pPlusMinusTimes*:

$$pPlusMinusTimes = pChainL\ addops\ (pChainL\ mulops\ pInteger)$$

in which we recognise a *foldr*:

$$pPlusMinusTimes = foldr\ pChainL\ pInteger\ [addops, mulops]$$

Now it has become straightforward to add new operators: just add the new operator, with its semantics, to the corresponding level of precedence. If its precedence lies between two already existing precedences, then just add a new list between these two levels. To complete the parsing of expressions we add the recognition of parentheses:

$$\begin{aligned} &pPack :: Eq\ s \Rightarrow [s] \to Parser\ s\ a \to [s] \to Parser\ s\ a \\ &pPack\ o\ p\ c = pSyms\ o\ \ *> p <*\ \ pSyms\ c \end{aligned}$$

$$\begin{aligned} pExpr &= foldr\ pChainL\ pFactor\ [addops, mulops] \\ pFactor &= pInteger <|> pPack\ \texttt{"("}\ pExpr\ \texttt{")"} \end{aligned}$$

As a final extension we add recognition of conditional expressions. In order to do so we will need to recognise keywords like **if**, **then**, and **else**. This invites us to add the companion to the *pChoice* combinator:

$$pSeq :: [Parser\ s\ a] \to Parser\ s\ [a]$$

$$\begin{aligned} pSeq\ (p : pp) &= (:) <\$> p <*> pSeq\ pp \\ pSeq\ [\,] &= pReturn\ [\,] \end{aligned}$$

Extending our parser with conditional expressions is now straightforward:

```
pExpr       = foldr pChainL pFactor [addops, mulops] <|> pIfThenElse

pIfThenElse = choose <$  pSyms "if"
                     <*> pBoolExpr
                     <*  pSyms "then"
                     <*> pExpr
                     <*  pSyms "else"
                     <*> pExpr

choose c t e = if c then t else e

pBoolExpr = foldr pChainR pRelExpr [orops, andops]
pRelExpr  =       True <$  pSyms "True"
            <|> False <$  pSyms "False"
            <|> pExpr <**> pRelOp <*> pExpr

andops = anyOp [((∧), "&&")]
orops  = anyOp [((∨), "||")]
pRelOp = anyOp [((⩽), "<="), ((⩾), ">="),
                ((≡), "=="), ((≢), "/="),
                ((<), "<"), ((>) , ">")]
```

Monadic Interface: *Monad* and *pTimes*

The parsers described thus far have the expressive power of context-free grammars. We have introduced extra combinators to capture frequently occurring grammatical patterns such as in the EBNF extensions. Because parsers are normal Haskell values, which are computed at run-time, we can however go beyond the conventional context-free grammar expressiveness by using the result of one parser to construct the next one. An example of this can be found in the recognition of XML-based input. We assume the input be a tree-like structure with tagged nodes, and we want to map our input onto the data type *XML*. To handle situations like this we make our parser type an instance of the class *Monad*:

```
instance Monad (Parser s) where
return = pReturn
P pa >>= a2pb = P (λinput → [b_input'' | (a, input') ← pa input
                                       , b_input''   ← unP (a2pb a) input'])

data XML = Tag String [XML] | Leaf String

pXML =       do t ← pOpenTag
                Tag t <$> pMany pXML <*  pCloseTag t
         <|> Leaf <$> pLeaf

pTagged p   = pPack "<" p ">"
pOpenTag    = pTagged pIdent
pCloseTag t = pTagged (pSym '/'  *> pSyms t)
pLeaf       = ...
pIdent      = pMany1 (pSatisfy (λc → 'a' ⩽ c ∧ c ⩽ 'z'))
```

A second example of the use of monads is in the recognition of the language $\{a^n b^n c^n | n >= 0\}$, which is well known not to be context-free. Here, we use the number of `'a'`'s recognised to build parsers that recognise exactly that number of `'b'`'s and `'c'`'s. For the result, we return the original input, which has now been checked to be an element of the language:

```
pABC = do as ← pMany (pSym 'a')
          let n = length as
          bs ← p_n_Times n (pSym 'b')
          cs ← p_n_Times n (pSym 'c')
          return (as ++ bs ++ cs)

p_n_Times :: Int → Parser s a → Parser s [a]
p_n_Times 0 p = pReturn []
p_n_Times n p = (:) <$> p <*> p_n_Times (n − 1) p
```

Conclusions

We have introduced the idea of a combinator language and have constructed a small library of basic and non-basic combinators. It should be clear by now that there is no end to the number of new combinators that can be defined, each capturing a pattern recurring in some input to be recognised. We finish this section by summing up the advantages of using an EDSL.

full abstraction Most special purpose programming languages have –unlike our host language Haskell– poorly defined abstraction mechanisms, often not going far beyond a simple macro-processing system. Although –with a substantial effort– amazing things can be achieved in this way as we can see from the use of TEX, we do not think this is the right way to go; programs become harder to get correct, and often long detours –which have little to do with the actual problem at hand– have to be taken in order to get things into acceptable shape. Because our embedded language inherits from Haskell –by virtue of being an embedded language– all the abstraction mechanisms and the advanced type system, it takes a head start with respect to all the individual implementation efforts.

type checking Many special purpose programming languages, and especially the so-called scripting languages, only have a weak concept of a type system; simply because the type system was not considered to be important when the design took off and compilers should remain small. Many scripting languages are completely dynamically typed, and some see this even as an advantage since the type system does not get into their way when implementing new abstractions. We feel that this perceived shortcoming is due to the very basic type systems found in most general purpose programming languages. Haskell however has a very powerful type system, which is not easy to surpass, unless one is prepared to enter completely new grounds, as with dependently typed languages such as Agda (see paper in this volume by Bove and Dybjer). One of the huge benefits of working with a strongly typed language is furthermore that the types of the library functions already give a very good insight in the role of the parameters and what a function is computing.

clean semantics One of the ways in which the meaning of a language construct is traditionally defined is by its denotational semantics, i.e. by mapping the language construct onto a mathematical object, usually being a function. This fits very well with the embedding of domain specific languages in Haskell, since functions are the primary class of values in Haskell. As a result, implementing a DSL in Haskell almost boils down to giving its denotational semantics in the conventional way and getting a compiler for free.

lazy evaluation One of the formalisms of choice in implementing the context sensitive aspects of a language is by using attribute grammars. Fortunately, the equivalent of attribute grammars can be implemented straightforwardly in a lazily evaluated functional language; inherited attributes become parameters and synthesized attributes become part of the result of the functions giving the semantics of a construct [**?**, **?**].

Of course there are also downsides to the embedding approach. Although the programmer is thinking he writes a program in the embedded language, he is still programming in the host language. As a result of this, error messages from the type system, which can already be quite challenging in Haskell, are phrased in terms of the host language constructs too, and without further measures the underlying implementation shines through. In the case of our parser combinators, this has as a consequence that the user is not addressed in terms of terminals, non-terminals, keywords, and productions, but in terms of the types implementing these constructs.

There are several ways in which this problem can be alleviated. We can try to hide the internal structure as much as possible by using a lot of **newtype** constructors, and thus defining the parser type by:

$$\textbf{newtype}\ \mathit{Parser}'\ s\ a = \mathit{Parser}'\ ([s] \to [(a, [s])])$$

Hoofdstuk 14

Programmatransformatie

In dit hoofdstuk wordt gesproken over het aantal reducties dat nodig is om een functie uit te rekenen. In Haskell bestaat de notie van reducties niet en de interpreter kan de hoeveelheid dus ook niet tonen. Lees reductie daarom gewoon als een tijdseenheid: hoe meer reducties, hoe langer het duurt.

Efficiëntie

Tijdgebruik

Elke aanroep van een functie die bij het uitrekenen van een expressie wordt gedaan, kost tijd. De gedane functie-aanroepen worden door de interpreter geteld: het zijn de *reductions* die na het antwoord worden vermeld.

Het aantal benodigde reducties kan de tweede keer dat een expressie wordt uitgerekend kleiner zijn dan de eerste keer. Bekijk bijvoorbeeld het volgende geval. De functie f is een ingewikkelde functie, bijvoorbeeld de faculteit-functie. De constante k is gedefinieerd met behulp van een aanroep van f. De functie g gebruikt k:

$$\begin{aligned} f\ x &= \mathit{product}\ [1\,..\,x] \\ k\ \ &= f\ 5 \\ g\ x &= k + 1 \end{aligned}$$

De constante k wordt dankzij lazy evaluatie niet tijdens het analyseren van de file uitgerekend. Hij wordt pas uitgerekend als hij voor het eerst gebruikt wordt, bijvoorbeeld door een aanroep van g:

```
? g 1
121
(57 reductions, 78 cells)
```

Als daarna nog eens g aangeroepen wordt, wordt de waarde van k niet opnieuw uitgerekend. De tweede aanroep van g is dus veel sneller:

```
? g 1
121
(3 reductions, 8 cells)
```

Je kunt in een file dus gerust allerlei constante-definities zetten. Pas als deze voor het eerst gebruikt worden kosten ze tijd. Functies die zijn gedefinieerd door een hogere-orde functie partieel te parametriseren, bijvoorbeeld

$$som = foldr\ (+)\ 0$$

kosten de tweede keer dat ze gebruikt worden één reductie minder:

```
? som [1,2,3]
6
(9 reductions, 19 cells)
? som [1,2,3]
6
(8 reductions, 17 cells)
```

Na twee keer aanroepen blijft het aantal reducties verder hetzelfde.

Complexiteitsanalyse

Bij de analyse van het tijdgebruik van een functie is vaak niet het precieze aantal reducties van belang. Veel interessanter is het om te zien hoe dit aantal reducties verandert als de grootte van de invoer verandert.

Bekijk bijvoorbeeld drie lijsten met 10, 100 en 1000 elementen (de getallen vanaf 1 in omgekeerde volgorde). Het aantal reducties van enkele functies op deze lijsten is als volgt:

elementen	*head*	*last*	*length*	*sum*	$(+\!\!+x)$	*isort*
10	2	11	42	32	52	133
100	2	101	402	302	502	10303
1000	2	1001	4002	3002	5002	1003003
n	2	$n+1$	$4n+2$	$3n+2$	$5n+2$	n^2+3n+3

Het bepalen van de *head* is altijd in twee reducties te doen. De head van een lijst hoeft immers alleen maar van het begin geplukt te worden. De functie *last* kost echter meer tijd naarmate de lijst langer is. Deze lijst moet immers helemaal doorlopen worden, en dat duurt langer naarmate de lijst langer is. Ook bij *length*, *sum* en $+\!\!+$ is het aantal benodigde reducties een lineaire functie van de lengte van de lijst. Bij *isort* is het aantal reducties echter een kwadratische functie van de lengte van de lijst.

Het is niet altijd zo gemakkelijk om te bepalen wat het aantal reducties is als functie van de lengte van de parameter. In de praktijk wordt er daarom mee volstaan om de *grootte-orde* van deze functie aan te geven: de benodigde tijd voor *head* is *constant*, voor *last*, *length*, *sum* en $+\!\!+$ is hij *lineair*, en voor *isort* is hij *kwadratisch* in de lengte van de parameter. De grootte-orde van een functie wordt genoteerd met de letter $\mathcal{O}$. Een kwadratische functie wordt aangegeven met $\mathcal{O}(n^2)$, een lineaire functie met $\mathcal{O}(n)$, en een constante functie met $\mathcal{O}(1)$.

Het aantal benodigde reducties voor zowel *length* als *sum* is $\mathcal{O}(n)$. Als de benodigde tijdsduur voor twee functies dezelfde grootte-orde heeft, kan die dus nog een constante factor schelen.

De volgende tabel geeft enig gevoel voor de snelheid waarmee de verschillende functies stijgen:

orde	$n = 10$	$n = 100$	$n = 1000$	naam
$\mathcal{O}(1)$	1	1	1	constant
$\mathcal{O}(\log n)$	3	7	10	logaritmisch
$\mathcal{O}(\sqrt{n})$	3	10	32	
$\mathcal{O}(n)$	10	100	1000	lineair
$\mathcal{O}(n \log n)$	30	700	10000	
$\mathcal{O}(n\sqrt{n})$	30	1000	31623	
$\mathcal{O}(n^2)$	100	10000	10^6	kwadratisch
$\mathcal{O}(n^3)$	1000	10^6	10^9	kubisch
$\mathcal{O}(2^n)$	1024	10^{30}	10^{300}	exponentieel

In deze tabel is voor de constante factor 1 genomen, en voor het grondtal van de logaritme 2. Voor de grootte-orde is het grondtal niet van belang, omdat wegens ${}^{g}\log n = {}^{2}\log n / {}^{2}\log g$ verandering van grondtal slechts een constante factor scheelt.

Let op dat, bijvoorbeeld, een $\mathcal{O}(n\sqrt{n})$ algoritme niet altijd 'beter' is dan een $\mathcal{O}(n^2)$ algoritme voor hetzelfde probleem. Als de constante factor in het $\mathcal{O}(n\sqrt{n})$ algoritme bijvoorbeeld tien keer zo groot is als die in het $\mathcal{O}(n^2)$ algoritme, dan is het eerste algoritme alleen sneller voor $n > 100$. Soms doen algoritmen met een lage complexiteit veel meer operaties per stap dan algoritmen met een hogere complexiteit, en hebben dus ook een veel grotere constante factor. Echter, als n maar groot genoeg is, zijn algoritmen met een lagere complexiteit sneller.

Aan de definitie van een functie is vaak eenvoudig de grootte-orde van het aantal benodigde reducties af te lezen. Die grootte-orde kan gebruikt worden als maat voor de *complexiteit* van de functie. Hier volgt een aantal voorbeelden. Het eerste voorbeeld gaat over een niet-recursieve functie. Dan volgen drie voorbeelden van recursieve functies waarbij de parameter van de recursieve aanroep iets kleiner is (1 korter of 1 minder), dus functies met een definitie van de vorm

$$\begin{aligned} f\ (x : xs) &= \ldots f\ xs \ldots \\ g\ (n+1) &= \ldots g\ n \ldots \end{aligned}$$

Tenslotte volgen drie voorbeelden van recursieve functies waarbij de parameter van de recursieve aanroep ongeveer halveert, zoals in

$$\begin{aligned} f\ (Tak\ x\ li\ re) &= \ldots f\ li \ldots \\ g\ (2 * n) &= \ldots g\ n \ldots \end{aligned}$$

- De functie is niet recursief, en bevat alleen aanroepen van functies met complexiteit $\mathcal{O}(1)$. Voorbeeld: *head*, *tail*, *abs*. (Let op: de tijd die nodig is om het resultaat van *tail* af te drukken is lineair, maar het bepalen van de *tail* op zich duurt constante tijd). Een aantal malen een constante blijft een constante, dus deze functies hebben zelf ook complexiteit $\mathcal{O}(1)$.

- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter. Het eindantwoord wordt vervolgens in constante tijd bepaald. Voorbeeld: *fac*, *last*, *insert*, *length*, *sum*. Bovendien: *map f* en *foldr f*, waarbij f constante complexiteit heeft. Deze functies hebben complexiteit $\mathcal{O}(n)$.

- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter. Het eindantwoord wordt vervolgens in lineaire tijd bepaald. Voorbeeld: *map f* of *foldr f*, waabij f lineaire complexiteit heeft. Een speciaal geval hiervan is *isort*; deze functie is immers gedefinieerd als *foldr insert* [ ], waarin *insert* lineaire complexiteit heeft. Deze functies hebben complexiteit $\mathcal{O}(n^2)$.

- De functie doet *twee* recursieve aanroepen van zichzelf met iets kleinere parameter. Voorbeeld: de voorlaatste versie van *subs* in paragraaf 7. Dit soort functies zijn heel vreselijk: bij elke stap in de recursie verdubbelt het aantal reducties. Deze functies hebben complexiteit $\mathcal{O}(2^n)$. blz. 95

- De functie doet één recursieve aanroep van zichzelf met een parameter die ongeveer twee keer zo klein geworden is. Het eindantwoord wordt vervolgens in constante tijd berekend. Voorbeeld: *elemBoom* in paragraaf 8. Deze functies hebben complexiteit $\mathcal{O}(\log n)$. blz. 122

- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op *beide* delen. Het eindantwoord wordt vervolgens in constante tijd berekend. Voorbeeld: *omvang* in paragraaf 8. Deze functies hebben complexiteit $\mathcal{O}(n)$. blz. 120

- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op beide delen. Het eindantwoord wordt vervolgens in lineaire tijd berekend. Voorbeeld: *labels* in paragraaf 8, *msort* in paragraaf 6. Deze functies hebben complexiteit $\mathcal{O}(n \log n)$. blz. 123 blz. 75

Samengevat in een tabel:

parameter	aant.rec.aanr.	vervolgens	voorbeeld	complexiteit
geen	0	constant	*head*	$\mathcal{O}(1)$
kleiner	1	constant	*sum*	$\mathcal{O}(n)$
kleiner	1	lineair	*isort*	$\mathcal{O}(n^2)$
kleiner	2	will.	*subs*	$\mathcal{O}(2^n)$
helft	1	constant	*elemBoom*	$\mathcal{O}(\log n)$
helft	1	lineair		$\mathcal{O}(n)$
helft	2	constant	*omvang*	$\mathcal{O}(n)$
helft	2	lineair	*msort*	$\mathcal{O}(n \log n)$

Verbeteren van efficiëntie

Hoe kleiner de parameter van een recursieve functie is, des te korter duurt het om de functie uit te rekenen. Uit het voorgaande volgt dat er daarnaast nog drie manieren zijn om de efficiëntie van een functie te verbeteren:

- doe de recursieve aanroep liever op gehalveerde parameters dan op een met één verminderde parameter;
- doe liever één dan twee recursieve aanroepen;
- houd het vervolgwerk na de recursieve aanroep(en) liever constant dan lineair.

Bij sommige functies kunnen deze methoden inderdaad gebruikt worden om een snellere functie te krijgen. We bekijken van elke methode een voorbeeld.

a. Kleine parameters

De operator $+\!\!+$ is gedefinieerd met inductie naar de linker parameter:

$$\begin{aligned} [\,] \quad\quad +\!\!+\, ys &= ys \\ (x : xs) +\!\!+\, ys &= x : (xs +\!\!+\, ys) \end{aligned}$$

De benodigde tijd voor het uitrekenen van $+\!\!+$ is daarom lineair in de lengte van de linker parameter. De lengte van de rechter parameter beïnvloedt de tijdsduur niet. Als twee lijsten samengevoegd moeten worden, maar de volgorde van de elementen doet er niet toe, dan is het dus efficiënter om de kortste van de twee voorop te zetten.

De operator is associatief, dat wil zeggen dat de waarde van $(xs +\!\!+\, ys) +\!\!+\, zs$ gelijk is aan die van $xs +\!\!+\, (ys +\!\!+\, zs)$. Voor de complexiteit maakt het echter wèl uit welke van de twee expressies berekend wordt.

Stel dat de lengte van xs gelijk is aan x, de lengte van ys y en die van zs z. Dan geldt:

tijd voor...	$(xs +\!\!+\, ys) +\!\!+\, zs$	$xs +\!\!+\, (ys +\!\!+\, zs)$
eerste $+\!\!+$	x	x
tweede $+\!\!+$	$x + y$	y
totaal	$2x + y$	$x + y$

Als $+\!\!+$ naar rechts associeert, wordt hij dus sneller berekend dan als hij naar links associeert.
blz. 38 Daarom wordt $+\!\!+$ in de prelude als rechts-associërende operator gedefinieerd (zie paragraaf 3).

b. Halveren i.p.v. verminderen (lineair vervolgwerk)

Bekijk functies waarbij het vervolg-werk na de recursieve aanroep lineair is. Een recursieve functie die zichzelf twee keer aanroept met een parameter die half zo groot is (bijvoorbeeld een linker- en een rechter deelboom van een gebalanceerde boom) heeft complexiteit $\mathcal{O}(n \log n)$. Dat is dus

beter dan een functie die zichzelf één keer aanroept met een parameter die slechts iets kleiner is geworden (bijvoorbeeld de staart van een lijst), die complexiteit $\mathcal{O}(n^2)$ heeft.

De sorteerfunctie *msort*

$$
\begin{aligned}
&\mathit{msort}\ \mathit{xs} \mid \mathit{lengte} \leqslant 1 = \mathit{xs}\\
&\qquad\quad\ \mid \mathit{otherwise}\ = \mathit{merge}\ (\mathit{msort}\ \mathit{ys})\ (\mathit{msort}\ \mathit{zs})\\
&\ \ \mathbf{where}\\
&\quad \mathit{ys} \quad\ \ = \mathit{take}\ \mathit{half}\ \mathit{xs}\\
&\quad \mathit{zs} \quad\ \ = \mathit{drop}\ \mathit{half}\ \mathit{xs}\\
&\quad \mathit{half} \quad = \mathit{lengte}\ /\ 2\\
&\quad \mathit{lengte} = \mathit{length}\ \mathit{xs}
\end{aligned}
$$

is dus efficiënter dan de functie *isort*

$$
\begin{aligned}
&\mathit{isort}\ [\,] \qquad\quad = [\,]\\
&\mathit{isort}\ (x : \mathit{xs}) = \mathit{insert}\ x\ (\mathit{isort}\ \mathit{xs})
\end{aligned}
$$

Bij veel functies op lijsten ligt een algoritme op de *isort*-manier het meest voor de hand. Het loont de moeite om altijd even na te gaan of er ook een *msort*-achtig algoritme mogelijk is. Denk daarbij aan de slagzin

> *Verdeel en Heers*

die in dit geval betekent: ‘verdeel de parameter in twee ongeveer even grote lijsten; pas de functie recursief toe op de helften, en voeg de deel-oplossingen vervolgens samen’. In de functie *msort* gebeurt het verdelen door de functies *take* en *drop*; het heersen gebeurt door de functie *merge*.

c. Halveren i.p.v. verminderen (constant vervolgwerk)

Bekijk functies die zichzelf één keer aanroepen met constant vervolgwerk. Een voorbeeld daarvan is de machtsverhef-functie:

$$
\begin{aligned}
&x \uparrow 0 \qquad\quad\ = 1\\
&x \uparrow (n+1) = x * x \uparrow n
\end{aligned}
$$

Deze functie heeft complexiteit $\mathcal{O}(n)$. Zou de parameter gehalveerd worden bij de recursieve aanroep, dan bedroeg de complexiteit slechts $\mathcal{O}(\log n)$. In het geval van machtsverheffen is dat mogelijk:

$$
\begin{aligned}
&x \uparrow 0 \qquad\qquad\ \ = 1\\
&x \uparrow (2*n) \qquad\ = \mathit{kwadraat}\ (x \uparrow n)\\
&x \uparrow (2*n+1) = x * \mathit{kwadraat}\ (x \uparrow n)\\
&\mathit{kwadraat}\ x \qquad = x * x
\end{aligned}
$$

In deze definitie wordt de bekende rekenregel $x^{2n} = (x^n)^2$ uitgebuit. In Haskell kunnen de gevallen ‘even parameter’ en ‘oneven parameter’ onderscheiden worden met behulp van patronen. In talen waarin dat niet mogelijk is kan het altijd nog met behulp van de functie *even*.

d. Weinig recursieve aanroepen (halverende parameter)

Als je het voor het kiezen hebt, is het beter om één recursieve aanroep te doen dan twee. Daarom is het in een zoekboom mogelijk om in $\mathcal{O}(\log n)$ tijd het gezochte element te vinden:

$$
\begin{aligned}
&\mathit{zoek}\ e\ \mathit{Blad} \qquad\qquad\qquad\ \ = \mathit{False}\\
&\mathit{zoek}\ e\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) \mid e \equiv x = \mathit{True}\\
&\qquad\qquad\qquad\qquad\quad\ \mid e < x = \mathit{zoek}\ e\ \mathit{li}\\
&\qquad\qquad\qquad\qquad\quad\ \mid e > x = \mathit{zoek}\ e\ \mathit{re}
\end{aligned}
$$

de functie wordt immers maar recursief aangeroepen op één helft: de linker deelboom als $e < x$, of de rechter deelboom als $e > x$. In een boom waarin de elementen niet geordend zijn, kost zoeken in

het algemeen $\mathcal{O}(n)$ tijd, omdat een recursieve aanroep op de linker èn de rechter deelboom nodig is:

$$\begin{array}{ll} \mathit{zoek}\ e\ \mathit{Blad} & = \mathit{False} \\ \mathit{zoek}\ e\ (\mathit{Tak}\ x\ \mathit{li}\ \mathit{re}) \mid e \equiv x & = \mathit{True} \\ \qquad\qquad\qquad\qquad \mid \mathit{otherwise} & = \mathit{zoek}\ e\ \mathit{li} \vee \mathit{zoek}\ e\ \mathit{re} \end{array}$$

als het element in de linkerboom wordt aangetroffen heb je geluk, maar anders zal toch *zoek* nog eens worden aangeroepen op de rechterboom.

e. Weinig recursieve aanroepen (verminderende parameter)

Het uitvoeren van één recursieve aanroep in plaats van twee is helemaal spectaculair als de parameter niet halveert maar slechts iets kleiner wordt: het maakt van een exponentieel algoritme een lineair algoritme! Het is dus uit den boze om twee recursieve aanroepen te doen met dezelfde parameter, zoals gebeurt in *subs*:

$$\begin{array}{ll} \mathit{subs}\ [\,] & = [[\,]] \\ \mathit{subs}\ (x : xs) & = \mathit{map}\ (x{:})\ (\mathit{subs}\ xs) +\!\!+ \mathit{subs}\ xs \end{array}$$

blz. 95

Zoals in paragraaf 7 is aangegeven, kan de recursieve aanroep in een **where**-constructie worden gezet, waardoor hij nog maar één keer wordt uitgevoerd:

$$\begin{array}{l} \mathit{subs}\ [\,] = [[\,]] \\ \mathit{subs}\ (x : xs) = \mathit{map}\ (x{:})\ (\mathit{subsxs}) +\!\!+ \mathit{subsxs} \\ \quad \textbf{where} \\ \qquad \mathit{subsxs} = \mathit{subs}\ xs \end{array}$$

Weliswaar wordt *subs* hiermee niet lineair, omdat de $+\!\!+$ meer dan constante tijd kost, maar toch wordt er tijd bespaard.

Als de twee recursieve aanroepen niet dezelfde parameter hebben, gaat deze methode niet op. Er is soms echter toch een verbetering mogelijk. Bekijk bijvoorbeeld de functie van Fibonacci:

$$\begin{array}{ll} \mathit{fib}\ 0 & = 0 \\ \mathit{fib}\ 1 & = 1 \\ \mathit{fib}\ (n+2) & = \mathit{fib}\ n + \mathit{fib}\ (n+1) \end{array}$$

De functie wordt tweemaal recursief aangeroepen met parameter die niet veel kleiner is, in ieder geval niet halveert. De complexiteit van *fib* is dus $\mathcal{O}(2^n)$.

Een verbetering hiervan is mogelijk door een extra functie te schrijven, die meer doet dan het gevraagde. In dit geval maken we een functie $\mathit{fib}'\ n$, die behalve de waarde van $\mathit{fib}\ n$ ook $\mathit{fib}\ (n-1)$ berekent. Deze functie levert dus een 2-tupel op. De functie *fib* kan dan geschreven worden door gebruik te maken van fib':

$$\begin{array}{l} \mathit{fib}\ 0 = 0 \\ \mathit{fib}\ n = \mathit{snd}\ (\mathit{fib}'\ n) \\ \quad \textbf{where} \\ \qquad \begin{array}{ll} \mathit{fib}'\ 1 & = (0,1) \\ \mathit{fib}'\ (n+1) & = (b, a+b)\ \textbf{where}\ (a,b) = \mathit{fib}'\ n \end{array} \end{array}$$

Deze functie bevat nog maar één recursieve aanroep, en heeft dus complexiteit $\mathcal{O}(n)$.

f. Vervolgwerk beperkt houden

Bij 'verdeel en heers'-functies, die het resultaat van twee recursieve aanroepen combineren tot een eindantwoord is het van belang hoeveel werk het combineren kost. Kost het combineren constante tijd, dan is de complexiteit $\mathcal{O}(n)$; kost het combineren lineaire tijd, dan is de complexiteit $\mathcal{O}(n \log n)$.

De functie *labels*, die de elementen van een boom in een lijst zet, heeft als de boom gebalanceerd is complexiteit $\mathcal{O}(n \log n)$:

$$\begin{aligned} &labels\ Blad &&= [\,] \\ &labels\ (Tak\ x\ li\ re) &&= labels\ li +\!\!+ [x] +\!\!+ labels\ re \end{aligned}$$

Voor het 'heersen' wordt immers de operator $+\!\!+$ gebruikt, die lineaire tijd kost.

Ook nu is verbetering mogelijk, door een extra functie te maken, die eigenlijk te veel doet:

$$\begin{aligned} &labelsVoor &&:: Boom\ a \to [a] \to [a] \\ &labelsVoor\ boom\ xs &&= labels\ boom +\!\!+ xs \end{aligned}$$

Gegeven deze functie kan *labels* geschreven worden als

$$labels\ boom = labelsVoor\ boom\ [\,]$$

Op deze manier levert dat natuurlijk nog geen tijdwinst op. Maar er is een andere definitie van *labelsVoor* mogelijk:

$$\begin{aligned} &labelsVoor\ Blad\ xs &&= xs \\ &labelsVoor\ (Tak\ x\ li\ re)\ xs &&= labelsVoor\ li\ (x : labelsVoor\ re\ xs) \end{aligned}$$

Deze functie doet twee recursieve aanroepen op de deelbomen (net als *labels*), maar het 'heersen' kost nu nog maar constante tijd (voor de operator :). De complexiteit van het algoritme is daarom $\mathcal{O}(n)$, een verbetering ten opzichte van de $\mathcal{O}(n \log n)$ van het oorspronkelijke algoritme.

Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte. Het geheugen wordt bij berekeningen voor drie doeleinden gebruikt:

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;
- het bijhouden van nog uit te rekenen deel-expressies.

Het geheugen voor de datastructuren wordt de *heap* ('hoop') genoemd; het geheugen voor de administratie van de berekening de *stack* ('stapel').

De heap

Op de heap worden datastructuren (lijsten, tupels, bomen enz.) opgebouwd door constructorfuncties te gebruiken. De heap bestaat uit *cellen*, waarin de informatie wordt opgeslagen.[1] Bij elke aanroep van de constructor-operator : worden twee nieuwe cellen gebruikt. Ook bij gebruik van zelf-gedefinieerde constructorfuncties worden cellen gebruikt (voor elke parameter van een constructorfunctie één).

Vroeg of laat is de gehele heap verbruikt. Op dat moment treedt de *garbage collector* ('vuilnisophaler') in werking. Alle cellen die niet meer nodig zijn worden daarbij voor hergebruik geschikt gemaakt. Bij het uitrekenen van de expressie *length* (*subs* $[1\,..\,8]$) worden de deelrijen van $[1\,..\,8]$ bepaald en geteld. Zodra een deelrij geteld is, is de deelrij zelf niet meer belangrijk. Bij garbage collection zal de betreffende heap-ruimte voor hergebruik worden vrijgegeven.

Als er tijdens het uitrekenen van een expressie garbage collection(s) hebben plaatsgevonden, wordt dit na afloop door de interpreter vermeld:

```
? length (subs [1..8])
```

[1] Afhankelijk van de gebruikte computer kost elke cell 4 of 8 bytes in het geheugen.

```
256
(4141 reductions, 9286 cells, 1 garbage collection)
? length (subs [1..10])
1024
(18487 reductions, 43093 cells, 6 garbage collections)
?
```

Het is ook mogelijk om elke afzonderlijke garbage-collection vermeld te krijgen. Daartoe moet de optie `+g` gezet worden:

$$?:set + g$$

Bij het begin van elke garbage collection wordt dan `{{Gc:` op het scherm gezet.Na afloop van elke garbage collection wordt daarachter het aantal opnieuw te gebruiken cellen vermeld:

```
? length (subs [1..10])
{{Gc:7987}}{{Gc:7998}}{{Gc:7994}}{{Gc:7991}}{{Gc:7997}}{{Gc:8004}}1024
(18487 reductions, 43093 cells, 6 garbage collections)
```

Als na garbage collection nog steeds niet genoeg geheugen beschikbaar is om een nieuwe cell aan te maken, wordt de berekening afgebroken met de melding

```
ERROR: Garbage collection fails to reclaim sufficient space
```

Het enige wat er dan nog opzit is om Haskell te starten met een grotere heap. Op het gebruik van de heap valt niet veel te bezuinigen. De opgebouwde datastructuren zijn meestal gewoon nodig. De enige manier om het gebruik van constructorfuncties, en daarmee het gebruik van de heap, te
blz. 97 vermijden is het zo veel mogelijk gebruiken van @-patronen, zoals beschreven in paragraaf 7. Daar werd op drie manieren de functie *tails* gedefinieerd:

- zonder patronen

$$\begin{aligned} tails\ [\,] &= [[\,]] \\ tails\ xs &= xs : tails\ (tail\ xs) \end{aligned}$$

- met gewone patronen

$$\begin{aligned} tails\ [\,] &= [[\,]] \\ tails\ (x : xs) &= (x : xs) : tails\ xs \end{aligned}$$

- met @-patronen

$$\begin{aligned} tails\ [\,] &= [[\,]] \\ tails\ lyst@(x : xs) &= lyst : tails\ xs \end{aligned}$$

Bij toepassing van deze functies op de lijst $[1\,..\,10]$ is het tijd- en geheugengebruik als volgt:

zonder patronen	210 reductions	569 cells
met gewone patronen	200 reductions	579 cells
met @-patronen	200 reductions	559 cells

Hieruit blijkt nogmaals dat de definitie met @-patronen de voordelen van de andere twee definities combineert.

De stack

De stack wordt door de interpreter gebruikt om de expressie op te slaan die uitgerekend moet worden. Ook worden de tussenresultaten op de stack opgeslagen. Elke 'reduction' levert een nieuw tussenresultaat op.

De stackruimte die nodig is om de tussenresultaten op te slaan kan groter zijn dan de uit te rekenen expressie en het resultaat. Bekijk bijvoorbeeld de tussenresultaten in de berekening *foldr* (+) 0 [1 . . 5], waarin de som van de getallen 1 t/m 5 wordt uitgerekend:

$$
\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ [1\,..\,5]\\
&1 + \mathit{foldr}\ (+)\ 0\ [2\,..\,5]\\
&1 + (2 + \mathit{foldr}\ (+)\ 0\ [3\,..\,5])\\
&1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [4\,..\,5]))\\
&1 + (2 + (3 + (4 + \mathit{foldr}\ (+)\ 0\ [5])))\\
&1 + (2 + (3 + (4 + (5 + \mathit{foldr}\ (+)\ 0\ [])))\\
&1 + (2 + (3 + (4 + (5 + 0))))\\
&1 + (2 + (3 + (4 + 5)))\\
&1 + (2 + (3 + 9))\\
&1 + (2 + 12)\\
&1 + 14\\
&15
\end{aligned}
$$

Het feitelijke uitrekenen van de +-operatoren begint pas nadat op de stack de complete expressie $1 + (2 + (3 + (4 + (5 + 0))))$ is opgebouwd. In de meeste gevallen heb je daar geen last van, maar het wordt vervelend als je erg lange lijsten gaat optellen:

```
? foldr (+) 0 [1..5000]
12502500
? foldr (+) 0 [1..6000]
ERROR: Control stack overflow
```

Als je in plaats van *foldr* de functie *foldl* gebruikt worden de getallen op een andere manier opgeteld. Ook hier is echter bij het optellen van lange lijsten veel stackruimte nodig:

$$
\begin{aligned}
&\mathit{foldl}\ (+)\quad 0 && [1\,..\,5]\\
&\mathit{foldl}\ (+)\ (\quad 0 + \ 1) && [2\,..\,5]\\
&\mathit{foldl}\ (+)\ ((\ \ 0 + \ 1) + 2) && [3\,..\,5]\\
&\mathit{foldl}\ (+)\ (((\ 0 + \ 1) + 2) + 3) && [4\,..\,5]\\
&\mathit{foldl}\ (+)\ ((((0 + \ 1) + 2) + 3) + 4) && [5]\\
&\mathit{foldl}\ (+)\ (((((0 + 1) + 2) + 3) + 4) + 5) && []\\
&\qquad\quad ((((0 + \ 1) + 2) + 3) + 4) + 5\\
&\qquad\quad (((\qquad 1 \ + 2) + 3) + 4) + 5\\
&\qquad\quad ((\qquad\ \ 3 \quad\ + 3) + 4) + 5\\
&\qquad\quad (\qquad\qquad 6 \quad\ + 4) + 5\\
&\qquad\qquad\qquad\qquad\ \ 10 \quad + 5\\
&\qquad\qquad\qquad\qquad\qquad\ \ 15
\end{aligned}
$$

Vanwege het lazy-evaluatie principe wordt het uitrekenen van parameters zo lang mogelijk uitgesteld. Dat geldt ook voor de tweede parameter van *foldl*. In dit geval is dat een beetje vervelend, want uiteindelijk wordt de expressie tòch uitgerekend. Het uitstellen is dus nergens goed voor geweest, en kost alleen maar een boel stackruimte.

Zonder lazy evaluatie was de berekening als volgt verlopen; zodra dat mogelijk is wordt de tweede parameter van *foldl* uitgerekend:

$$
\begin{array}{ll}
\mathit{foldl}\ (+)\ 0 & [1\,..\,5] \\
\mathit{foldl}\ (+)\ (0+1) & [2\,..\,5] \\
\mathit{foldl}\ (+)\ 1 & [2\,..\,5] \\
\mathit{foldl}\ (+)\ (1+2) & [3\,..\,5] \\
\mathit{foldl}\ (+)\ 3 & [3\,..\,5] \\
\mathit{foldl}\ (+)\ (3+3) & [4\,..\,5] \\
\mathit{foldl}\ (+)\ 6 & [4\,..\,5] \\
\mathit{foldl}\ (+)\ (6+4) & [5] \\
\mathit{foldl}\ (+)\ 10 & [5] \\
\mathit{foldl}\ (+)\ (10+5) & [\,] \\
\mathit{foldl}\ (+)\ 15 & [\,] \\
15 &
\end{array}
$$

De benodigde stackruimte is bij deze evaluatievolgorde gedurende de berekening constant.

Hoe krijg je de interpreter nu zo ver dat hij een functie (in dit geval de partieel geparametriseerde functie $\mathit{foldl}\ (+)$) niet-lazy uitrekent? Speciaal voor dit doel definieren we een functie , *strict*, gebruik makend van een ingebouwde functie $\mathit{seq} :: a \to b \to b$. Wat *seq* doet is eerst zijn linker argument uitrekenen en vervolgens zijn rechter argument opleveren.

$$\mathit{strict}\ f\ x = x\ `\mathit{seq}`\ f\ x$$

De functie krijgt dus een functie en een waarde als parameter, en past de functie toe op de waarde, maar dwingt eerste wel even evaluatie van die waarde af.

Met behulp van de functie *strict* kun je elke gewenste functie niet-lazy doen uitrekenen. In het voorbeeld hierboven zou het handig zijn als *foldl* zijn *tweede* parameter niet-lazy uitrekent. Dat wil zeggen dat de partieel geparametriseerde functie $\mathit{foldl}\ (+)$ niet-lazy moet zijn in zijn eerste parameter. Dat kan, door de functie *strict* erop toe te passen; we schrijven dus $\mathit{strict}\ (\mathit{foldl}\ (+))$. Let op het extra paar haakjes: $\mathit{foldl}\ (+)$ moet als geheel aan *strict* worden meegegeven.

In de prelude wordt een functie foldl' gedefinieerd, die hetzelfde doet als *foldl*, maar dan met niet-lazy tweede parameter. Ter vergelijking geven we eerst nog eens de definitie van *foldl*:

$$
\begin{array}{l}
\mathit{foldl}\ f\ e\ [\,] = e \\
\mathit{foldl}\ f\ e\ (x:xs) = \mathit{foldl}\ f\ (f\ e\ x)\ xs
\end{array}
$$

In de functie foldl' wordt bij de recursieve aanroep de functie *strict* gebruikt zoals hiervoor beschreven:

$$
\begin{array}{l}
\mathit{foldl}'\ f\ e\ [\,] = e \\
\mathit{foldl}'\ f\ e\ (x:xs) = \mathit{strict}\ (\mathit{foldl}'\ f)\ (f\ e\ x)\ xs
\end{array}
$$

Gebruikmakend van foldl' is de stackruimte constant, zodat rustig de som van hele lange lijsten berekend kan worden:

```
? foldl (+) 0 [1..6000]
ERROR: Control stack overflow
? foldl' (+) 0 [1..6000]
18003000
```

Voor operatoren zoals + en $*$, die de waarde van beide parameters nodig hebben om hun resultaat te berekenen, is het aan te bevelen om foldl' te gebruiken in plaats van *foldr* of *foldl*. Er zal daardoor nooit tekort aan stackruimte optreden. In de prelude is dan ook gedefinieerd:

$$
\begin{array}{ll}
\mathit{sum} & = \mathit{foldl}'\ (+)\ 0 \\
\mathit{product} & = \mathit{foldl}'\ (*)\ 1
\end{array}
$$

Voor de operator $\wedge$ wordt echter wel *foldr* gebruikt:

$$and = foldr\ (\wedge)\ True$$

Bij deze operator zorgt lazy evaluatie er immers voor dat de tweede parameter niet wordt uitgerekend als dat niet nodig is. Dat betekent dat *and* stopt met rekenen zodra er een waarde *False* in de lijst staat. Bij gebruik van *foldl′* zou hij altijd de hele lijst verwerken.

Wetten

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:

$$
\begin{array}{ll}
& \neg((a \vee b) \vee c) \to \neg d \\
= & \quad \text{(wet van de Morgan)} \\
& (\neg(a \vee b) \wedge \neg c) \to \neg d \\
= & \quad \text{(wet van de Morgan)} \\
& ((\neg a \wedge \neg b) \wedge \neg c) \to \neg d \\
= & \quad \text{(wet van Howard)} \\
& (\neg a \wedge \neg b) \to (\neg c \to \neg d) \\
= & \quad \text{(wet van Howard)} \\
& \neg a \to (\neg b \to (\neg c \to \neg d))
\end{array}
$$

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 uitvinden welke wet gebruikt is. De aanwezigheid van hints draagt in belangrijke mate bij aan de 'leesbaarheid' van een afleiding.

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 deel-expressie vervangt door een andere deel-expressie 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:

$$(map\ f \circ map\ g)\ xs = map\ (f \circ g)\ xs$$

Dit is geen *definitie* van *map* of $\circ$ (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 determinant-functie op
blz. 106 matrices (zie paragraaf 7) komt bijvoorbeeld de volgende expressie voor:

$$(altsum \circ zipWith\ (*.)\ ry \circ map\ det \circ map\ Mat \circ gaps \circ transpose)\ rys$$

Door bovenstaande wet hierop toe te passen blijkt dat dit ook geschreven kan worden als

$$(altsum \circ zipWith\ (*.)\ ry \circ map\ (det \circ Mat) \circ gaps \circ transpose)\ rys$$

Door gebruik te maken van wetten kan met complete programma's 'gerekend' worden. Programma's worden op die manier getransformeerd tot andere programma's. 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 begin-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 begin-programma.

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

$$f \circ (g \circ h) = (f \circ g) \circ h$$

- *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:

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

- *map* distribueert over samenstelling:

$$map\ (f \circ g) = map\ f \circ map\ g$$

- 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 singleton-lijst 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:

$$map\ f \circ (x:) = (f\ x:) \circ map\ f$$

- Elk gebruik van *map* kan ook geschreven worden als aanroep van *foldr*:

$$map\ f\ xs = foldr\ g\ [\,]\ xs\ \textbf{where}\ g\ x\ ys = f\ x : ys$$

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.

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 14 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. blz. 191

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 singleton-lijst*
Als e het neutrale element is van de operator f, dan geldt

$$\mathit{foldr}\ f\ e\ [x] = x$$

Bewijs:

```
foldr f e [x]
=   (notatie lijst-opsomming)
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:

$$f \circ (g \circ h) = (f \circ g) \circ h$$

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`
	= (def. `(.)`)
	`f ((g.h) x)`
	= (def. `(.)`)
	`f (g (h x))`
		`((f.g) . h) x`
		= (def. `(.)`)
		`(f.g) (h x)`
		= (def. `(.)`)
		`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 bewijs-methode 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:

$$map\ f \circ (x:) = ((f\ x):) \circ map\ f$$

Bewijs:

	`map f . (x:)`	`((f x):) . map f`

`xs`	`(map f . (x:)) xs`
	= (def. `(.)`)
	`map f ((x:) xs)`
	= (section-notatie)
	`map f (x:xs)`
	= (def. `map`)
	`f x : map f xs`
		`( ((f x):) . map f) xs`
		= (def. `(.)`)
		`((f x):) (map f xs)`
		= (section-notatie)
		`f x : map f xs`

Net als het vorige bewijs had dit bewijs natuurlijk ook als één lange afleiding geschreven kunnen worden. In deze tweekoloms-notatie 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.

Inductieve bewijzen

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:

$$map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$$

Bewijs met inductie naar xs:

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)`
		`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*. 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:

$$map\ (f \circ g) = map\ f \circ map\ g$$

Bewijs met inductie naar xs:

	`map (f.g)`	`map f . map g`

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`
		`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:

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

Dit is een generalisatie van de distributiewet van *map* over $+\!\!+$.

Bewijs met inductie naar xss:

	`map f . concat`	`concat . map (map f)`

`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)`
	= (distributie-wet)
	`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)`

Ook nu weer zijn de expressies in de onderste regels gelijk vanwege de aanname van de inductiehypothese. 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 singleton-lijst, en een lijst met minstens twee elementen $(x1 : x2 : xs)$.

Wet *dualiteitswet*
Als f een associatieve operator is (dus $x \text{ `}f\text{` } (y \text{ `}f\text{` } z) = (x \text{ `}f\text{` } y) \text{ `}f\text{` } z$), en e is het neutrale element van f (dus $f\ x\ e = f\ e\ x = x$ voor alle x), dan geldt:

$$foldr\ f\ e = foldl\ f\ e$$

Bewijs met inductie naar xs:

	`foldr f e`	`foldl f e`
`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)`
		`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)`

De laatste twee regels zijn gelijk omdat de lijst $(x1 \text{ `}f\text{` } x2) : xs$ korter is dan $x1 : x2 : xs$. De wet mag voor deze lijst dus al aangenomen worden.

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*:

$$foldr\ f\ e\ (reverse\ xs) = foldl\ (flip\ f)\ e\ xs$$

- 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 hulp-wet, 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`

`xs`	`foldr (:) [] xs`	`id xs`
		= (def. `id`)
		`xs`

`[]`	`foldr (:) [] []`
	= (def. `foldr`)
	`[]`
		`[]`

`x:xs`	`foldr (:) [] (x:xs)`
	= (def. `foldr`)
	`x : foldr (:) [] xs`
		`x : xs`

> **Wet** *hulpwet voor de volgende wet*
>
> $$foldr\ f\ e\ (\mathbf{as} +\!\!+ [b]) = foldr\ f\ (f\ b\ e)\ \mathbf{as}$$

Bewijs met inductie naar **as**:

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)`

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:

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`

De laatste expressies in dit bewijs zijn gelijk vanwege de inductiehypothese. 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 6 gedefinieerd als blz. 67

$$\begin{aligned} reverse\ [\,] &= [\,] \\ reverse\ (x : xs) &= reverse\ xs +\!\!+ [x] \end{aligned}$$

of het daaraan equivalente

$$\begin{aligned} &reverse = foldr\ post\ [\,] \\ &\quad \textbf{where} \\ &\qquad post\ x\ xs = xs +\!\!+ [x] \end{aligned}$$

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

$$\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]$$

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:

$$\begin{aligned} \mathit{fib}\ 0 &= 0 \\ \mathit{fib}\ 1 &= 1 \\ \mathit{fib}\ (n+2) &= \mathit{fib}\ n + \mathit{fib}\ (n+1) \end{aligned}$$

blz. 186

De benodigde tijd voor het berekenen van *fib* n is $\mathcal{O}(2^n)$. In paragraaf 14 werd al een verbetering tot $\mathcal{O}(n)$ bereikt, 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:

$$\mathit{fib}\ n = c * (p \uparrow n - q \uparrow n)$$

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.

Bewijs van 'Fibonacci door machtsverheffen' met inductie naar n:

n	`fib` n	$c \times (p^n - q^n)$
0	`fib` 0	$c \times (p^0 - q^0)$
	= (def. `fib`)	= (def. machtsverheffen)
	0	$c \times (1 - 1)$
	= (def. $\times$)	= (eigenschap $-$)
	$c \times 0$	$c \times 0$
1	`fib` 1	$c \times (p^1 - q^1)$
	= (def. `fib`)	= (def. machtsverheffen)
	1	$c \times (p - q)$
	= (def. /)	= (eigenschap c)
	$c \times (1/c)$	$c \times (1/c)$
n+2	`fib` (n+2)	$c \times (p^{n+2} - q^{n+2})$
	= (def. `fib`)	= (eigenschap machtsverheffen)
	`fib` n `+` `fib` (n+1)	$c \times (p^n p^2 - q^n q^2)$
		= (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 $\times$)
		$c \times (p^n - q^n) + c \times (p^{n+1} - q^{n+1})$

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 halverings-methode. Door *fib* te definiëren door

$$
\begin{array}{l}
\mathit{fib}\ n = c * (p\ \hat{}.\ n - q\ \hat{}.\ n) \\
\quad \textbf{where} \\
\quad\quad c \quad\quad = 1.0\ /\ \mathit{wortel5} \\
\quad\quad p \quad\quad = 0.5 * (1.0 + \mathit{wortel5}) \\
\quad\quad q \quad\quad = 0.5 * (1.0 - \mathit{wortel5}) \\
\quad\quad \mathit{wortel5} = \mathit{sqrt}\ 5.0
\end{array}
$$

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.

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 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 resultaat-lijst 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:

$$\mathit{len} \circ (x:) = (1+) \circ \mathit{len}$$

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:

$$len \circ map\ f = len$$

Het bewijs verloopt met inductie naar xs:

	`len . map f`	`len`

`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`

Wet *lengte na* $+\!\!+$
De lengte van de concatenatie van twee lijsten is de som van de lengtes van die lijsten:

$$len\ (xs +\!\!+ ys) = len\ xs + len\ ys$$

Het bewijs verloopt met inductie naar xs:

`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)`

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:

$$len \circ concat = sum \circ map\ len$$

Het bewijs verloopt met inductie naar *xss*:

	`len . concat`	`sum . map len`
`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)`

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:
>
> $$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys$$

Bewijs met inductie naar *xs*:

`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)`

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:
>
> $$sum \circ concat = sum \circ map\ sum$$

Het bewijs verloopt met inductie naar *xss*:

	`sum . concat`	`sum . map sum`
`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)`

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 ge*map*te 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:

$$sum\ (map\ (1+)\ xs) = len\ xs + sum\ xs$$

Bewijs met inductie naar *xs*:

`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)`

In opgave 14.4 wordt deze wet gegeneraliseerd naar een willekeurige lineaire functie.
blz. 213

Wetten over combinatorische functies

Met behulp van een aantal hierboven genoemde wetten zijn wetten te bewijzen over combinatorische functies uit sectie 7. We bewijzen voor *inits*, *segs* en *combs* een wet die aangeeft hoeveel elementen het resultaat heeft (zie ook opgave 7.3):
blz. 93
blz. 113

$$\begin{aligned} len \circ inits &= (1+) \circ len \\ len \circ segs &= f \circ len \textbf{ where } f\ n = 1 + (n * n + n)\ /\ 2 \\ len \circ combs\ k &= (`boven` k) \circ len \end{aligned}$$

Wet *aantal beginsegmenten*
Het aantal beginsegmenten van een lijst is één meer dan het aantal elementen van de lijst:

$$len \circ inits = (1+) \circ len$$

Het bewijs verloopt met inductie naar *xs*:

	`len . inits`	`(1+) . len`
`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)`

Wet *aantal segmenten*
Het aantal segmenten van een lijst is een kwadratische functie van het aantal elementen van de lijst:

$$len \circ segs = f \circ len \textbf{ where } f\ n = 1 + (n * n + n)\ /\ 2$$

Het bewijs verloopt met inductie naar xs. We schrijven n voor $len\ xs$.

	`len . segs`	`f . len where f` n `=` $1+(n^2+n)/2$
`xs`	`len (segs xs)`	`f (len xs)`
`[]`	`len (segs [])`	`f (len [])`
	= (def. `segs`)	= (def. `len`)
	`len [[]]`	`f 0`
	= (def. `len`)	= (def. `f`)
	`1 + len []`	$1+(0^2+0)/2$
	= (def. `len`)	= (uitrekenen)
	`1 + 0`	`1 + 0`
`x:xs`	`len (segs (x:xs))`	`f (len (x:xs))`
	= (def. `segs`)	= (def. `len`)
	`len ( segs xs`	`f (1 +` n`)`
	`++ map (x:)(inits xs))`	= (def. `f`)
	= (lengte na ++)	$1+((1+n)^2+(1+n))\ /\ 2$
	`len (segs xs) +`	= (merkwaardig product)
	`len (map (x:)(inits xs))`	$1+((1+2n+n^2)+(1+n))\ /\ 2$
	= (lengte na map)	= (+ associatief en commutatief)
	`len (segs xs) + len (inits xs)`	$1+((n^2+n)+(2+2n))\ /\ 2$
	= (aantal beginsegmenten)	= (distributie /)
	`len (segs xs) + 1 + len xs`	$1+(n^2+n)/2+1+n$

De definitie van *boven* in paragraaf 2 was niet volledig. De complete definitie luidt: blz. 16

$$\begin{aligned} boven\ n\ k \mid n \geqslant k &= fac\ n\ /\ (fac\ k * fac\ (n-k)) \\ \mid n < k &= 0 \end{aligned}$$

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':

$$len \circ combs\ k = (`boven`k) \circ len$$

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`
`k` `xs`	`len (combs k xs)`	$\binom{len\ xs}{k}$

`0`
`xs`
	`len (combs 0 xs)`
	= (def. `combs`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
	= (def. `len`)
	`1 + 0`
	= (eigenschap +)
	`1`
		$\binom{len\ xs}{0}$
		= (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`
		$\binom{len\ [\,]}{k+1}$
		= (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)`
		$\binom{len\ (x:xs)}{k+1}$
		= (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 inductiestap-bewijs 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:

$$\binom{n+1}{k+1} \;=\; 0 \;=\; 0+0 \;=\; \binom{n}{k} + \binom{n}{k+1}$$

Polymorfie

De volgende wetten zijn geldig voor alle functies f:

$$
\begin{aligned}
\mathit{inits} \circ \mathit{map}\ f &= \mathit{map}\ (\mathit{map}\ f) \circ \mathit{inits} \\
\mathit{segs} \circ \mathit{map}\ f &= \mathit{map}\ (\mathit{map}\ f) \circ \mathit{segs} \\
\mathit{subs} \circ \mathit{map}\ f &= \mathit{map}\ (\mathit{map}\ f) \circ \mathit{subs} \\
\mathit{perms} \circ \mathit{map}\ f &= \mathit{map}\ (\mathit{map}\ f) \circ \mathit{perms}
\end{aligned}
$$

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:

$$\mathit{inits} \circ \mathit{map}\ f = \mathit{map}\ (\mathit{map}\ f) \circ \mathit{inits}$$

Het bewijs verloopt met inductie naar *xs*:

	`inits . map f`	`map (map f) . inits`
`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))`
		`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

$$\mathit{combinat} \circ \mathit{map}\ f = \mathit{map}\ (\mathit{map}\ f) \circ \mathit{combinat}$$

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

$$\mathit{combinat} :: [a] \to [[a]]$$

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:

$$combinat \circ map\ f = map\ (map\ f) \circ combinat$$

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
blz. 93 eigenschappen van elementen', die in sectie 7 werd gebruikt.

blz. 195 Er zijn wetten die lijken op deze 'wet van de combinatorische functies'. In paragraaf 14 werd bijvoorbeeld de wet 'map na concat' bewezen. Die wet stelt dat voor alle functies f geldt:

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

De wet kun je natuurlijk ook andersom lezen. Dan staat er:

$$concat \circ map\ (map\ f) = map\ f \circ concat$$

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:

$$concat :: [[a]] \to [a]$$

Neemt bij gebruik van combinatorische functies het aantal lijst-nivo'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
blz. 31 daarvoor geldende wet voldoet. Wel is de functie een *polymorfe* functie. In paragraaf 2 werd een polymorfe functie gedefinieerd als 'een functie met een type waar type-variabelen in voorkomen'. Net als het begrip 'combinatorische functie' is het begrip 'polymorfe functie' met een wet minder vaag te definiëren. Dat gaat als volgt:

> Een functie *poly* tussen lijsten heet een *polymorfe* functie als voor alle functies f geldt:
>
> $$poly \circ \underbrace{map\ (\{\lambda rm \lambda dots}_{nmap\ \{\lambda rm's} = \underbrace{map\ (\{\lambda rm \lambda dots}_{kmap\ \{\lambda rm's} \circ poly$$
>
> Deze functie heeft dan het type:
>
> $$poly\tilde{}\tilde{} :: \tilde{}\tilde{}\ [\underbrace{\{\lambda rm \lambda dots}_{n \text{ dimensionale lijst}}\ \tilde{}->\ \tilde{}\ [\underbrace{\{\lambda rm \lambda dots}_{k \text{ dimensionale lijst}}$$

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 tupel, 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*.[2]

Bewijzen van rekenkundige wetten

blz. 191 In paragraaf 14 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

[2] 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.

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.

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 data-declaratie. 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 data-declaratie 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 data-declaratie 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:

$$\begin{aligned}\textbf{data}\ \mathit{Nat} = {} & \mathit{Nul} \\ \mid {} & \mathit{Volg}\ \mathit{Nat}\end{aligned}$$

Een natuurlijk getal is dus òf het getal *Nul*, of het wordt opgebouwd door de constructor-functie *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:

$$\begin{aligned}\mathbf{1} &= \mathit{Volg}\ \mathit{Nul} \\ \mathit{twee} &= \mathit{Volg}\ (\mathit{Volg}\ \mathit{Nul}) \\ \mathit{drie} &= \mathit{Volg}\ (\mathit{Volg}\ (\mathit{Volg}\ \mathit{Nul})) \\ \mathit{vier} &= \mathit{Volg}\ (\mathit{Volg}\ (\mathit{Volg}\ (\mathit{Volg}\ \mathit{Nul})))\end{aligned}$$

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:

$$\begin{aligned}\mathit{Nul} \quad + y &= y \\ \mathit{Volg}\ x + y &= \mathit{Volg}\ (x + y)\end{aligned}$$

In de tweede regel wordt de te definiëren functie recursief aangeroepen. Dit is toegestaan, omdat x een kleinere datastructuur is dan $\mathit{Volg}\ x$. Met behulp van de plus-functie kan, ook weer met inductie, een vermenigvuldig-functie gedefinieerd worden:

$$\begin{aligned}\mathit{Nul} \quad * y &= \mathit{Nul} \\ \mathit{Volg}\ x * y &= y + (x * y)\end{aligned}$$

Ook hier wordt de te definiëren functie recursief aangeroepen met een kleinere parameter. Met behulp van deze functie kan de machtsverhef-functie worden gedefinieerd, ditmaal met inductie naar de tweede parameter:

$$\begin{aligned}x \uparrow \mathit{Nul} &= \mathit{Volg}\ \mathit{Nul} \\ x \uparrow \mathit{Volg}\ y &= x * (x \uparrow y)\end{aligned}$$

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.	$x + Nul$	$=$	x	
2.	$x + Volg\ y$	$=$	$Volg\ (x + y)$	
3.	$x + y$	$=$	$y + x$	$+$ is commutatief
4.	$(x + y) + z$	$=$	$x + (y + z)$	$+$ is associatief
5.	$x * Nul$	$=$	Nul	
6.	$x * Volg\ y$	$=$	$x + (x * y)$	
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.	$x \uparrow (y + z)$	$=$	$x \uparrow y * x \uparrow z$	
12.	$(x * y) \uparrow z$	$=$	$x \uparrow z * y \uparrow z$	
13.	$(x \uparrow y) \uparrow z$	$=$	$x \uparrow (y * z)$	herhaald machtsverheffen

Bewijs van wet 1, met inductie naar x:

`x`	`x + Nul`	`x`

`Nul`	`Nul + Nul`
	= (def. `+`)
	`Nul`
		`Nul`

`Volg x`	`Volg x + Nul`
	= (def. `+`)
	`Volg (x+Nul)`
		`Volg x`

Bewijs van wet 2, met inductie naar x:

`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))`

Bewijs van wet 3 (plus is commutatief), met inductie naar x:

`x`	`x + y`	`y + x`

`Nul`	`Nul + y`
	= (def. `+`)
	`y`
		`y + Nul`
		= (wet 1)
		`y`

`Volg x`	`Volg x + y`
	= (def. `+`)
	`Volg (x+y)`
		`y + Volg x`
		= (wet 2)
		`Volg (y+x)`

Bewijs van wet 4 (plus is associatief), met inductie naar x:

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))

Bewijs van wet 5, met inductie naar x:

x	x * Nul	Nul
Nul	Nul * Nul = (def. *) Nul	Nul
Volg x	Volg x * Nul = (def. *) Nul + (x*Nul)	Nul = (def. +) Nul+Nul

Bewijs van wet 6, met inductie naar x:

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))	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:

x	x * y	y * x
Nul	Nul * y = (def. *) Nul	y * Nul = (wet 5) Nul
Volg x	Volg x * y = (def. *) y + (x*y)	y * Volg x = (wet 6) y + (y*x)

Bewijs van wet 8 ($*$ distribueert links over $+$), met inductie naar x:

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))	(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:

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))

Bewijs van wet 11, met inductie naar y:

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)	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

14.1 Welke versie van *segs* is efficiënter: die uit paragraaf 7 of die uit opgave 7.1? blz. 95 blz. 113

14.2 Bekijk het (derde) schema in paragraaf 14. Geef bij de optimalisatie-methodes **b** t/m **f** in paragraaf 14 aan in welke regel van het schema de algoritmen voor en na optimalisatie vallen. blz. 184 blz. 184

14.3 De functie *concat* kan gedefinieerd worden door

$$concat\ xss = fold\ (+\!\!+)\ [\,]\ xss$$

Daarbij kan voor *fold* één van de drie functies *foldr*, *foldl* of *foldl′* gebruikt worden. Bespreek het effect van deze keuze voor de efficiëntie, zowel voor het geval dat *xss* uitsluitend eindige lijsten bevat, als voor het geval dat er ook oneindige lijsten in zitten.

14.4 In paragraaf 14 wordt de wet blz. 204

$$sum\ (map\ (1+)\ xs) = len\ xs + sum\ xs$$

bewezen. Formuleer een dergelijke wet voor een willekeurige lineaire functie in plaats van (1+), dus

$$sum\ (map\ ((k+) \circ (n*))\ xs) = \ldots$$

Bewijs de geformuleerde wet.

14.5 Bewijs de volgende

Wet *fold na concatenatie*
Als $(\oplus)$ een associatieve operator is, dan geldt:

$$foldr\ (\oplus)\ e \circ concat = foldr\ (\oplus)\ e \circ map\ (foldr\ (\oplus)\ e)$$

14.6 Bepaal een functie g en een waarde e waarvoor geldt:

$$map\ f = foldr\ g\ e$$

Bewijs de gelijkheid voor de gevonden g en e.

14.7 Bewijs de volgende wet:

$$len \circ subs = (2\uparrow) \circ len$$

14.8 Bewijs dat *subs* een combinatorische functie is.

14.9 Bewijs wet 12 en wet 13 uit paragraaf 14. blz. 208

Hoofdstuk 15

Prolog in Haskell

In this chapter[1] we describe an interpreter for the programming language *Prolog*, which is the canonical representative of the class of so-called *logic languages*. It was inspired by a similar presentation by Cohen [**?**]. As we will see the language is based on the predicate calculus. In this chapter we will see the *list-of-successes* method coming back, just as we will recognise elements from type inferencing. The chapter furthermore serves as a demonstration of the use of the parser combinators from chapter **??**.

Prolog without variables; the propositional calculus

Just like a Haskell interpreter reads a collection of functions definitions, and subsequently evaluates given expressions by rewriting function application, the Prolog interpreter we are about to develop takes a set of *rules* and tries to *satify* given *goals* by applying these rules.

In the simplest form of Prolog, which we introduce in this section, the goal is just a *name*. Each *rule* states that a goal can be reached if a couple of sub-goals can be reached. An example of a collection of rules is:

$$
\begin{aligned}
&a \leftarrow b, d.\\
&b \leftarrow .\\
&c \leftarrow b, e.\\
&d \leftarrow b.\\
&e \leftarrow a.
\end{aligned}
$$

If we now ask whether the goal, say a, can be satisfied then a *proof* or *derivation* is constructed by the interpreter in the course of trying to answer the question, and if such a proof can be constructed the final answer is affirmative. In the case of goal a the second rule states that b can be reached without further assumptions, the fourth line states that d can be reached if b can be reached, and finally the first rule states that a can be reached if b and d can be reached. So we interpret the commas separating the elements in a right hand sides of a rule as *and*-operators. Notice that there is nothing which prevents us from giving several rules with the same left-hand side name.

Also notice that the problem we try to solve here can be phrased in grammatical terms; if we associate with name a non-terminal and interpret the rules as grammar productions, then the question whether a goal g can be reached is equivalent to asking whether the non-terminal g can derive the empty string.

[1] based on a paper by Jeroen Fokker: *Prolog in Haskell*, 1989

We start out by defining some data types which can be used to represent the program (i.e. the collection of rules):

$$\begin{array}{l}\textbf{type}\ \mathit{Goal} = \mathit{Char}\\ \textbf{data}\ \mathit{Rule} = \mathit{Goal} :\Leftarrow: [\mathit{Goal}]\end{array}$$

The next step is to construct the function *solve*, which takes a list of goals and tries to solve them with the given rule set:

$$\mathit{solve} :: [\mathit{Rule}] \to [\mathit{Goal}] \to \mathit{Bool}$$

If we now want to see whether `'a'` and `'b'`can both be solved, we call the function solve as:[2]

```
:{
let rules = [ 'a' :<-: "bd", 'b' :<-: ""
            , 'c' :<-: "be", 'd' :<-: "b", 'e' :<-: "a"]
:}
print (solve rules "ab")
```

The function *solve* succeeds if there are no goals left to be solved, i.e. when its second parameter is the empty list [ ]; if the list of goals is nonempty we look for those rules which can be used to solve one of the goals, i.e. which has the goal as its left-hand side symbol. For each rule which applies we:

(i) remove the left-hand side symbol from the set of goals still to be solved,

(ii) add the symbols in right-hand side of that rule to the remaining set of goals, and

(iii) call the function *solve* recursively with this new set of goals.

Once the collection of goals gets exhausted we have found a proof.

$$\begin{array}{l}\mathit{solve}\ \mathit{rules}\ [\,] = \quad \mathit{True}\\ \mathit{solve}\ \mathit{rules}\ (g : \mathit{gs}) = \mathit{or}\ [\mathit{solve}\ \mathit{rules}\ (\mathit{rhs} +\!\!+\ \mathit{gs}) \mid g' :\Leftarrow: \mathit{rhs} \leftarrow \mathit{rules}, g \equiv g']\end{array}$$

Note that in order to make this a viable solution we should keep track of the goals we are currently dealing with, in order to prevent the function *solve* calling itself recursively with goals being dealt with showing up in the list of goals again; if this happens the recursion never stops.

Prolog with variables: Predicate calculus

A very short introduction to Prolog

In this paragraph we extend the Prolog interpreter such that it can handle parametrised goals. As an example of such a set of more powerful rules we describe a family relationship:

$$\begin{array}{ll}\mathit{mother}\ (\mathit{wilhelmina}, \mathit{juliana}) & \leftarrow .\\ \mathit{mother}\ (\mathit{juliana}, \mathit{beatrix}) & \leftarrow .\\ \mathit{mother}\ (\mathit{beatrix}, \mathit{alexander}) & \leftarrow .\\ \mathit{father}\ (\mathit{alexander}, \mathit{amalia}) & \leftarrow .\\ \mathit{parent}\ (X, Y) & \leftarrow \mathit{mother}\ (X, Y).\\ \mathit{parent}\ (X, Y) & \leftarrow \mathit{father}\ (X, Y).\\ \mathit{grandmother}\ (X, Y) & \leftarrow \mathit{mother}\ (X, Z), \mathit{parent}\ (Z, Y).\end{array}$$

[2] The `:\{` and `:\}` tokens make it possible to distribute the **let**-definition over a couple of lines, without GHCi interpreting each line separately.

The first four rules state simple facts about the Dutch royal family[3], and hence have an empty right-hand side. The next two lines encode the predicates $\forall X, Y.mother(X, Y) \rightarrow parent(X, Y)$ and $\forall X, Y.father(X, Y) \rightarrow parent(X, Y)$. The last line encodes the rule $\forall X, Y.(\exists Z.mother(X, Z) \wedge parent(Z, Y)) \rightarrow grandmother(X, Y)$. Note that a variable which occurs both in the left-hand side and the right hand side of a rule is universally quantified in the corresponding predicate, whereas a variable which occurs only in the right-hand side is quantified existentially.

We can now ask the interpreter whether the goal *mother* (*juliana*, *beatrix*) can be proven, which is indeed the case.

More interesting is that we can also ask for which values a predicate which contains free variables can be satisfied; if we pose the question *grandmother* (X, Y), we get all possible grandmother relations which can be inferred from our set of rules:

```
term? grandmother(X,Y)
X= wilhelmina
Y= beatrix

X= juliana
Y= alexander

X= beatrix
Y= amalia
term?
```

For a term we distinguish three kinds of parameters:

- constants, starting with a lower case letter, such as in *juliana*
- variables, starting with an upper case letter, such as in X and Y
- function symbols, applied to a sequence of parameters, such as (*grandmother* (*beatrix*, Y))

The first alternative actually is eauivalent to a function symbol taking an empty list of parameters. Note that function symbols are not interpreted in any way! They get their meaning from how they are used in rules and patterns.

As another example of how to use function symbols in Prolog we define the *append* relation for lists. A list is either the object *nil*, or the term consisting of the function symbol *cons* applied to an object and a list. Since Prolog is not a typed language there is however nothing which prevents us from applying the *cons* function symbol to one or three parameters, nor is there a way to enforce that the second parameter is a list (i.e something produced by a *nil* or a *cons*)!

Using the function symbols *cons* and *nil* we can now express what is means to *append* two lists:

$$append\ (cons\ (A, X),\ Y, cons\ (A, Z)) :- append\ (X,\ Y, Z).$$
$$append\ (nil, X, X)$$

What distinguishes Prolog from Haskell is that although we call *append* a function symbol it actually defines a ternary relation, and we can run the program "backwards" by provinding the second and third parameter, i.e. by asking for which lists it holds that if we append the single element 3 it holds that the result is the list $1, 2, 3$:

[3] `http://www.koninklijkhuis.nl/english`

```
term? append(X, cons(3, nil), cons (1, cons (2, cons (3, nil))))
goal                 : append(X, cons(3, nil), cons(1, cons(2, cons(3, nil))))
unifies with head of : append(cons(A0, X0), Y0, cons(A0, Z0)):-append(X0, Y0, Z0).
new environment      : [("X",cons(A0, X0)),("Y0",cons(3, nil)),("A0",1),
                        ("Z0",cons(2, cons(3, nil)))]
new goals            : [append(X0, Y0, Z0)]

goal                 : append(X0, Y0, Z0)
unifies with head of : append(cons(A1, X1), Y1, cons(A1, Z1)):-append(X1, Y1, Z1).
new environment      : [("X0",cons(A1, X1)),("Y1",cons(3, nil)),("A1",2),
                        ("Z1",cons(3, nil)),("X",cons(A0, X0)),
                        ("Y0",cons(3, nil)),("A0",1),("Z0",cons(2, cons(3, nil)))]
new goals            : [append(X1, Y1, Z1)]

goal                 : append(X1, Y1, Z1)
unifies with head of : append(nil, X2, X2):-.
new environment      : [("X1",nil),("X2",cons(3, nil)),("X0",cons(A1, X1)),
                        ("Y1",cons(3, nil)),("A1",2),("Z1",cons(3, nil)),
                        ("X",cons(A0, X0)),("Y0",cons(3, nil)), ("A0",1),
                        ("Z0",cons(2, cons(3, nil)))]
new goals            : []

X = cons(1, cons(2, nil))
term?
```

Figuur 15.1: A trace of *append* $(X, cons\ (3, nil), cons\ (1, cons\ (2, cons\ (3, nil))))$

```
term? append(X, cons(3,nil), cons(1, cons(2, cons(3, nil))))
X = cons(1,cons(2,nil))
```

Before we develop the interpreter we give a trace of the resolution process leading to the above result in figure 15.1.

Although there is a relation flavour in some Prolog programs it is actually the case that most of the Prolog programs, and at least large parts of most Prolog programs, do not make use of this relational behaviour. Most parameters will be either used as input- or output parameters. In the famous Turbo Prolog system one is even required to indicate whether a parameter is an input or an output parameter, thus reducing the Prolog program effectively to a functional program which is implicitly using the list-of-successes method.

The Interpreter Code

Since this chapter is a literal script we start out by importing some necessary Haskell modules:

```
module PrologInterpreter where
import Char hiding (isSpace)
import Text.ParserCombinators.UU.Parsing
import Data.List
import System.IO
```

In the program we are about to develop we will use a single type to represent goals and and elements from rules; we also represent single names as functions without arguments:

```
type Ident = String
data Term = Con Int
          | Var Ident
          | Fun Ident [Term]
          deriving Eq

data Rule  = Term :<=: [Term]

instance Show Term where
  show (Con i)    = show i
  show (Var i)    =      i
  show (Fun i [])  =      i
  show (Fun i ts) = i ++ "(" ++ showCommas ts ++ ")"

instance Show Rule where
  show (t :<=: ts) = show t ++ ":-" ++ showCommas ts ++ "."

showCommas l = concat (intersperse ", " (map show l))
```

If we use such a term as a goal it should be a *Fun*, since it does not make sense to ask for the solution of a single variable. As we have seen the answer to the question whether a goal can be satisfied is no longer a truth value, but a list of bindings for the variable occurring free in the original goal. We represent suc bindings as a list of (*Ident*, *Term*) pairs. In our implementation we will not use environments which are *idempotent*, i.e. if we look for value of a variable in the environment, the result may be another variable which is bound elsewhere in the environment; as a consequence we have to repeatedly look up variables, until we find either a proper *Fun*-alternative bound to it, or until the identifier we have at hand is not bound in the environment at all.

```
type Env = [(Ident, Term)]

lookUp (Var x) e = case lookup x e of
                     Nothing -> Var x
                     Just res -> lookUp res e
lookUp t _       = t
```

The function *solve* becomes a bit more complicated: it does not return a simple *Bool*-ean value indicating whether the goal can be solved, but returns instead a list of all environments for which the goal can be satisfied. If we fail to find any derivation this list will be empty. Hence we use again the *list-of-successes* method we have already seen used in the combinator based parsing.

Besides taking the list of rules and a list of goals, the function *solve* also takes an environment containing all assumptions made about variables occurring in either the original goal or any of the encountered sub-goals thus far.

Since we may use the same rulemore than once in a derivation we need a way to keep all the instances of the rules apart. We do so by tagging all the identifiers occurring in a rule instance with an integer, such that each ule instance gets its own unique identification. As a consequence the function *solve* takes one more parameter: an *Int* value which is used to tag the identifiers in new rule instantiations. The function *tag* creates such a new rule instance:

```
solve :: [Rule] → [Term] → Env → Int → [(Env, Trace)]
type Trace = [String]
class Taggable a where
  tag :: Int → a → a

instance Taggable Term where
  tag n (Con x)    = Con x
  tag n (Var x)    = Var (x ++ show n)
  tag n (Fun x xs) = Fun x (map (tag n) xs)

instance Taggable Rule where
  tag n (c :⇐: cs) = (tag n c) :⇐: (map (tag n) cs)
```

The decision whether we can use a given rule to solve a given goal now becomes a bit more involved: instead of comparing the goal and the rule head, we have to see whether we can *unify* these two terms, i.e. whether we can find a substitution (i.e. an *Env* value) which makes the two terms equal. This unification process we have seen before when we dealt with type inferencing: the variables in the terms correspond to polymorphic type positions, and the constants with monomorphic types.

We proceed as follows:

- two constants can be unified if they are the same
- if one of the two terms is a variable then we extend the environment with a single binding, in which the variable is bound to the other term (actually we should also check whether the variable can somehow be reached from that term, since in that case we would represent an infinite term, which is forbidden in Prolog)
- if both terms are function applications we require that:
  (i) the function symbols are the same
  (ii) the number of parameters are the same
  (iii) the arguments can be pairwise unified by the same substitution

Since unification may fail we let it return a *Maybe Env*, and since we will be folding with *unify* over lists of arguments, we also pass it a *Maybe Env*. If the latter equals *Nothing* this indicates that somewhere before we concluded already that unification of the root term we strated with is not possible;

```
unify :: (Term, Term) → Maybe Env → Maybe Env
unify _ Nothing              = Nothing
unify (t, u) env@(Just e) = uni (lookUp t e) (lookUp u e)
  where uni (Var x) y          = Just ((x, y) : e)
        uni x (Var y)          = Just ((y, x) : e)
        uni (Con x) (Con y) = if x ≡ y then env else Nothing
        uni (Fun x xs) (Fun y ys)
          | x ≡ y ∧ length xs ≡ length ys
                               = foldr unify env (zip xs ys)
          | otherwise          = Nothing
        uni _ _                = Nothing
```

The next function to deal with is *solve* itself. In the list comprehension it starts by making fresh copies of all the rules. For each of the rules we next check whether its head unifies with the current goal. If this succeeds we get a new environment r. In this case we can now recursively call *solve* with the updated set of goals, and the just found environment r:

```
solve rules [] e _ = [(e, [])]
solve rules lt@(t : ts) e n =
  [(sol, msg : trace) | (c :<=: cs) <- map (tag n) rules
                      , Just r      <- [unify (t, c) (Just e)]
                      , let msg = display "goal                    : " t          ++
                                  display "unifies with head of    : " (c :<=: cs) ++
                                  display "new environment         : " r          ++
                                  display "new goals               : " (cs ++ ts) ++ "\n"
                      , (sol, trace) <- solve rules (cs ++ ts) r (n + 1)
                      ]

display string value = string ++ show value ++ "\n"
```

The *main* program

For the sake of completeness we present the *main* program, which reads a collection of rules form a file, and subsequently asks for goals to be answered.

```
main :: IO ()
main =
do hSetBuffering stdin LineBuffering
   putStr "File with rules? "
   fn <- getLine
   s <- readFile fn
   let (rules, errors) = start (pList pRule) s
   if Prelude.null errors
     then do mapM_ (putStrLn ∘ show) rules
             loop rules
     else  do putStrLn "No rules parsed"
             mapM_ (putStrLn ∘ show) errors
             main

loop rules = do putStr "term? "
                s <- getLine
                if (s ≡ "stop") then return ()
                  else do let (goal, errors) = start pFun s
                          if null errors then printsolutions (solve rules [goal] [] 0)
                            else do putStrLn "A term was expected:"
                                    mapM_ (putStrLn ∘ show) errors
                          loop rules
```

Parsing

Using the function from the *uu − parsinglib* package it is straightforward to read the file with rules and the terms given as goals:

```
type Pars a = P (Str Char) a

pRule :: Pars Rule
pRule  = (:⇐:) <$> pFun
                <*> (pToken ":-"  *> pListSep pComma pTerm `opt` [])
                <*   pSym '.'

pTerm, pCon, pVar, pFun :: Pars Term
pTerm = pCon <|> pVar <|> pFun
pCon  = Con  <$> pNatural
pVar  = Var  <$> pList1 (pSym ('A', 'Z'))
pFun  = Fun  <$> pIdentifier <*> (pParens (pListSep pComma pTerm) `opt` [])

isSpace x = x ≡ ' ' ∨ x ≡ '\n'
start p inp = parse ((,) <$> p <*> pEnd) $ listToStr ∘ filter (¬ ∘ isSpace) $ inp

pComma    = pSym ','
pParens p = pSym '(' *> p <* pSym ')'
pIdentifier = (:) <$> pSym ('a', 'z') <*> pList (pLower <|> pUpper <|> pDigit)

pDigit  = pSym ('0', '9')
pLower = pSym ('a', 'z')
pUpper = pSym ('A', 'Z')

pDigitAsInt = digit2Int <$> pDigit
pNatural    = foldl (λa b → a * 10 + b) 0 <$> pList1 pDigitAsInt
digit2Int a = ord a − ord '0'
```

Presenting the solutions

Since the Prolog interpreter can return many different bindings for the free variables in original goal we have decided to present them one by one. This is taken care of by the function *printSolutions*, which was called in the function *loop* which was presented before. Note that only the values finally bound to the free variables in the original goal are printed.

```
printsolutions :: [(Env, Trace)] → IO ()
printsolutions sols = sequence_ [do {printsolution bs; getLine} | bs ← sols]

printsolution (bs, trace) = do mapM_ putStr trace
  putStr (concat ∘ map showBdg $ bs)
where showBdg (x, t) | isUpper (head x) ∧ length x ≡ 1 = x ++ " = " ++ showTerm t ++ "\n"
                     | otherwise = ""
      showTerm (Con n) = show n
      showTerm t@(Var _) = showTerm (lookUp t bs)
      showTerm (Fun f []) = f
      showTerm (Fun f ts) = f ++ "(" ++ concat (intersperse ", " (map showTerm ts)) ++ ")"
```

Hoofdstuk 16

Graphical User Interfaces

wxHaskell

A Portable and Concise GUI Library for Haskell

Daan Leijen
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
daan@cs.uu.nl

Abstract

wxHaskell is a graphical user interface (GUI) library for Haskell that is built on wxWidgets: a free industrial strength GUI library for C++ that has been ported to all major platforms, including Windows, Gtk, and MacOS X. In contrast with many other libraries, wxWidgets retains the native look-and-feel of each particular platform. We show how distinctive features of Haskell, like parametric polymorphism, higher-order functions, and first-class computations, can be used to present a concise and elegant monadic interface for portable GUI programs.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (Functional) Programming*; D.2.2 [**Design Tools and Techniques**]: User interfaces.

General Terms

Design, Languages.

Keywords

Graphical user interface, combinator library, layout, wxWidgets, Haskell, C++.

1 Introduction

The ideal graphical user interface (GUI) library is efficient, portable across platforms, retains a native look-and-feel, and provides a lot of standard functionality. A Haskell programmer also expects good abstraction facilities and a strong type discipline. wxHaskell is a free GUI library for Haskell that aims to satisfy these criteria [25].

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

There is no intrinsic difficulty in implementing a GUI library that provides the above features. However, the amount of work and maintainance associated with such project should not be underestimated; many GUI libraries had a promising start, but failed to be maintained when new features or platforms arose. With wxHaskell, we try to avoid this pitfall by building on an existing cross-platform framework named wxWidgets: a free industrial strength GUI library for C++ [43].

wxWidgets provides a common interface to native widgets on all major GUI platforms, including Windows, Gtk, and Mac OS X. It has been in development since 1992 and has a very active development community. The library also has strong support from the industry and has been used for large commercial applications, for example, AOL communicator and AVG anti-virus.

wxHaskell consists of two libraries, *WXCore* and *WX*. The *WXCore* library provides the core interface to wxWidgets functionality. It exposes about 2800 methods and more than 500 classes of wxWidgets. Using this library is just like programming wxWidgets in C++ and provides the raw functionality of wxWidgets. The extensive interface made it possible to already develop substantial GUI programs in wxHaskell, including a Bayesian belief network editor and a generic structure editor, called Proxima [41]. The *WXCore* library is fully Haskell'98 compliant and uses the standard foreign function interface [11] to link with the wxWidgets library.

The *WX* library is implemented on top of *WXCore* and provides many useful functional abstractions to make the raw wxWidgets interface easier to use. This is where Haskell shines, and we use type class overloading, higher-order functions, and polymorphism to capture common programming patterns. In particular, in Section 6 we show how *attribute abstractions* can be used to model widget settings and event handlers. Furthermore, *WX* contains a rich combinator library to specify layout. In section 7, we use the layout combinator library as a particular example of a general technique for declarative abstraction over imperative interfaces. As described later in this article, the *WX* library does use some extensions to Haskell'98, like existential types. Most of this article is devoted to programming wxHaskell with the *WX* library, and we start with two examples that should give a good impression of the functionality of the library.

2 Examples

Figure 1 is a small program in wxHaskell that shows a frame with a centered label above two buttons. Pressing the *ok* button closes the frame, pressing the *cancel* button changes the text of the label.

```
main = start gui
gui :: IO ()
gui =
  do f   ← frame    [text := "Example"]
     lab ← label  f [text := "Hello wxHaskell"]
     ok  ← button f [text := "Ok"]
     can ← button f [text := "Cancel"]
     set ok  [on command := close f]
     set can [on command := set lab [text := "Goodbye?"]]
     set f [layout := column 5 [floatCenter (widget lab)
                               ,floatCenter $
                                row 5 [widget ok, widget can]]]
```

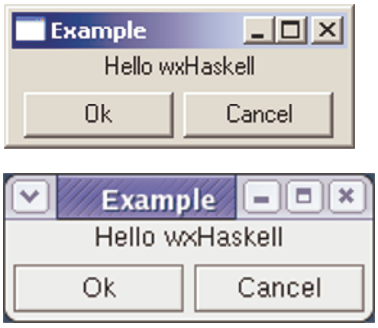


Figure 1. A first program in wxHaskell, with screenshots on Windows XP and Linux (Gtk)

A graphical wxHaskell program[1] is initialized with the *start* function, that registers the application with the graphical subsystem and starts an event loop. The argument of *start* is an *IO* value that is invoked at the initialization event. This computation should create the initial interface and install further event handlers. While the event loop is active, Haskell is only invoked via these event handlers.

The *gui* value creates the initial interface. The *frame* creates a top level window frame, with the text `"Example"` in the title bar. Inside the frame, we create a text label and two buttons. The expression *on command* designates the event handler that is called when a button is pressed. The *ok* button closes the frame, which terminates the application, and the *cancel* button changes the content of the text label.

Finally, the *layout* of the frame is specified. wxHaskell has a rich layout combinator library that is discussed in more detail in Section 7. The text label and buttons float to the center of its display area, and the buttons are placed next to each other. In contrast to many dialogs and windows in contemporary applications, this layout is fully resizeable.

3 Asteroids

We now discuss a somewhat more realistic example of programming with wxHaskell. In particular we look at a minimal version of the well known *asteroids* game, where a spaceship tries to fly through an asteroid field. Figure 2 shows a screenshot. Even though we use a game as an example here, we stress that wxHaskell is a library designed foremost for user interfaces, not games. Nevertheless, simple games like asteroids show many interesting aspects of wxHaskell.

The game consists of a spaceship that can move to the left and right using the arrow keys. There is an infinite supply of random rocks (asteroids) that move vertically downwards. Whenever the spaceship hits a rock, the rock becomes a flaming ball. In a more realistic version, this would destroy the ship, but we choose a more peaceful variant here. We start by defining some constants:

```
height   = 300
width    = 300
diameter = 24
chance   = 0.1 :: Double
```

[1] One can access wxHaskell functionality, like the portable database binding, without using the GUI functionality.

For simplicity, we use fixed dimensions for the game field, given by *width* and *height*. The *diameter* is the diameter of the rocks, and the *chance* is the chance that a new rock appears in a given time frame. The main function of our game is *asteroids* that creates the user interface:

```
asteroids :: IO ()
asteroids =
  do g      ← getStdGen
     vrocks ← variable [value := randomRocks g]
     vship  ← variable [value := div width 2]
     f      ← frame    [resizeable := False]
     t      ← timer f  [interval := 50
                       ,on command := advance vrocks f]
     set f [text       := "Asteroids"
           ,bgcolor    := white
           ,layout     := space width height
           ,on paint   := draw vrocks vship
           ,on leftKey  := set vship [value :~ \x → x-5]
           ,on rightKey := set vship [value :~ \x → x+5]
           ]
```

First a random number generator *g* is created that is used to randomly create rocks. We create two mutable variables: *vrocks* holds an infinite list that contains the positions of all future rock positions, *vship* contains the current *x* position of the ship.

Next, we create the main window frame *f*. The frame is not resizeable, and we can see in the screenshot that the maximize box is greyed out. We also attach an (invisible) timer *t* to this frame that ticks every 50 milliseconds. On each tick, it calls the function *advance* that advances all rocks to their next position and updates the screen.

Finally, we set a host of attributes on the frame *f*. Note that we could have set all of these immediately when creating the frame but the author liked this layout better. The text *Asteroids* is displayed in the title bar, and the background color of the frame is white. As there are no child widgets, the *layout* just consists of empty space of a fixed size. The attributes prefixed with *on* designate event handlers. The *paint* event handler is called when a redraw of the frame is necessary, and it invokes the *draw* function that we define later in this section. Pressing the left arrow key or right arrow key changes the x position of the spaceship. In contrast to the $(:=)$ operator, the $(:\sim)$ operator does not assign a new value, but applies a function to the attribute value, in this case, a function that increases or

decreases the x position by 5 pixels. A somewhat better definition would respect the bounds of the game too, for example:

```
on leftKey := set vship [value :~ \x → max 0 (x − 5)]
```

The *vrocks* variable holds an infinite list of all future rock positions. This infinite list is generated by the *randomRocks* function that takes a random number generator *g* as its argument:

```
randomRocks :: RandomGen g ⇒ g → [[Point]]
randomRocks g =
  flatten [] (map fresh (randoms g))
flatten rocks (t : ts) =
  let now  = map head rocks
      later = filter (not ∘ null) (map tail rocks)
  in now : flatten (t ++ later) ts
fresh r
  | r > chance = []
  | otherwise  = [track (floor (fromIntegral width ∗ r / chance))]
track x =
  [point x (y − diameter) | y ← [0,6 .. height + 2 ∗ diameter]]
```

The standard *randoms* function generates an infinite list of random numbers in the range $[0,1)$. The *fresh* function compares each number agains the *chance*, and if a new rock should appear, it generates a finite list of positions that move the rock from the top to the bottom of the game field. The expression *map fresh* (*randoms g*) denotes an infinite list, where each element contains either an empty list, or a list of positions for a new rock. Finally, we *flatten* this list into a list of time frames, where each element contains the position of every rock in that particular time frame.

The *advance* function is the driving force behind the game, and it is called on every timer tick.

```
advance vrocks f =
  do set vrocks [value :~ tail]
     repaint f
```

The *advance* function advances to the next time frame by taking the *tail* of the list. It then forces the frame *f* to repaint itself. The *paint* event handler of the frame calls the *draw* function that repaints the game:

```
draw vrocks vship dc view =
  do rocks ← get vrocks value
     x     ← get vship value
     let ship       = point x (height − 2 ∗ diameter)
         positions  = head rocks
         collisions = map (collide ship) positions
     drawShip dc ship
     mapM_ (drawRock dc) (zip positions collisions)
     when (or collisions) (play explode)
```

The *draw* function was partially parameterised with the *vrocks* and *vship* variables. The last two parameters are supplied by the paint event handler: the current *device context* (*dc*) and view area (*view*). The device context is in this case the window area on the screen, but it could also be a printer or bitmap for example.

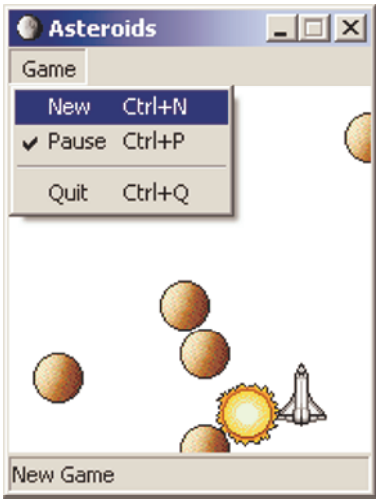


Figure 2. The asteroids game.

First, we retrieve the current rocks and x position of the spaceship. The position of the spaceship, *ship*, is at a fixed y-position. The current rock *positions* are simply the head of the rocks list. The *collisions* list tells for each rock position whether it collides with the ship. Finally, we draw the ship and all the rocks. As a final touch, we also play a sound fragment of an explosion when a collision has happened. The *collide* function just checks if two positions are too close for comfort using standard vector functions from the wxHaskell library:

```
collide pos0 pos1 =
  let distance = vecLength (vecBetween pos0 pos1)
  in distance ⩽ fromIntegral diameter
```

A ship can be drawn using standard drawing primitives, for example, we could draw the ship as a solid red circle:

```
drawShip dc pos =
  circle dc pos (div diameter 2) [brush := brushSolid red]
```

The *circle* function takes a device context, a position, a radius, and a list of properties as arguments. The *brush* attribute determines how the circle is filled. wxHaskell comes with an extensive array of drawing primitives, for example polygons, rounded rectangles, and elliptic arcs. But for a spaceship, it is nicer of course to use bitmaps instead:

```
drawShip dc pos =
  drawBitmap dc ship pos True []
drawRock dc (pos, collides) =
  let picture = if collides then burning else rock
  in drawBitmap dc picture pos True []
```

The *drawBitmap* function takes a device context, a bitmap, a position, the transparency mode, and a list of properties as arguments. The bitmap for a rock is changed to a *burning* ball when it collides with the spaceship. To finish the program, we define the resources that we used:

```
rock    = bitmap "rock.ico"
burning = bitmap "burning.ico"
```

```
ship    = bitmap "ship.ico"
explode = sound "explode.wav"
```

And that is all we need – asteroids in 55 lines of code.

3.1 Extensions

Extending the game with new features is straightforward. For example, to change the speed of the spaceship by pressing the plus or minus key, we just add more event handlers to the frame *f*:

```
on (charKey '-') := set t [interval :~ \i -> i*2]
on (charKey '+') := set t [interval :~ \i -> max 10 (div i 2)]
```

The minus key increments the timer interval, while the plus key decrements it, effectively making the game run slower or faster. The screenshot in Figure 2 also shows a menu and status bar. Here is the code for creating the menu pane:

```
game  <- menuPane       [text  := "&Game"]
new   <- menuItem game  [text  := "&New\tCtrl+N"
                        ,help  := "New game"]
pause <- menuItem game  [text  := "&Pause\tCtrl+P"
                        ,help  := "Pause game"
                        ,checkable := True]
menuLine game
quit  <- menuQuit game  [help  := "Quit the game"]
```

The "&" notation in menu texts signifies the hotkey for that item when the menu has the focus. Behind a tab character we can also specify a menu shortcut key. There is also a structured interface to such accelerator keys, but specifying those keys as part of the menu text proves very convenient in practice. Note that the *pause* menu is a checkable menu item. For the *quit* menu, we use the special *menuQuit* function instead of *menuItem*, as this item is sometimes handled specially on certain platforms, in particular on Mac OS X.

To each new menu item, we attach an appropiate event handler:

```
set new   [on command := asteroids]
set pause [on command := set t [enabled :~ not]]
set quit  [on command := close f]
```

The *quit* menu simply closes the frame. The *pause* menu toggles the enabled state of the timer by applying the *not* function. Turning off the timer effectively pauses the game.[2] The *new* menu is interesting as it starts a completely new asteroids game in another frame. As we don't use any global variables, the new game functions completely independent from any other asteroids game. Finally, we show the menu by specifying the menu bar of the frame:

```
set f [menubar := [game]]
```

Our final extension is a status bar. A status bar consists of status fields that contain text or bitmaps. For our game, a single status field suffices.

```
status <- statusField [text := "Welcome to asteroids"]
set f [statusbar := [status]]
```

[2] Although one can cheat now by changing the x position of the ship while in pause mode.

The *status* is passed to the *advance* function, which updates the status field with the count of rocks that are currently visible:

```
advance status vrocks f =
  do (r:rs) <- get vrocks value
     set vrocks [value := rs]
     set status [text  := "rocks: " ++ show (length r)]
     repaint f
```

4 Design

In the previous section, we have seen how graphical user interfaces in wxHaskell are defined using the imperative *IO* monad. Despite the use of this monad, the examples have a declarative flavour and are much more concise than their imperative counterparts in C++. We believe that the ability to treat *IO* computations as first class values allows us to reach this high level of abstraction: using the ability to defer, modify and combine computations, we can for example use attribute lists to set properties of widgets.

The use of mutable variables to communicate across event handlers is very imperative, though. There has been much research into avoiding mutable state and providing a declarative model for GUI programming. We discuss many of these approaches in the related work section. However, this is still an active research area and we felt it was better to provide a standard monadic interface first. As shown in [13], it is relatively easy to implement a declarative interface on top of a standard monadic interface, and others have already started working on a Fruit [14] interface on top of wxHaskell [35].

4.1 Safety

The wxHaskell library imposes a strong typing discipline on the wxWidgets library. This means that the type checker will reject programs with illegal operations on widgets. Also, the memory management is fully automatic, with the provision that programmers are able to manually manage certain external resources like font descriptors or large bitmaps. The library also checks for *NULL* pointers, raising a Haskell exception instead of triggering a segmentation fault.

Common to many other GUI libraries, wxHaskell still suffers from the *hierarchy problem*: the library imposes a strict hierarchical relation on the created widgets. For example, the program in Figure 1 shows how the buttons and the label all take the parent frame *f* as their first argument. It would be more natural to just create buttons and labels:

```
f   <- frame  [text := "Example"]
lab <- label  [text := "Hello wxHaskell"]
ok  <- button [text := "Ok"]
can <- button [text := "Cancel"]
```

The layout now determines a relation between widgets. We believe that the hierarchical relation between widgets is mostly an artifact of libraries where memory management is explicit: by imposing a strict hierarchical order, a container can automatically discard its child widgets.

Even with the parent argument removed, there are still many ways to make errors in the layout specification. Worse, these errors are

not caught by the type checker but occur at runtime. There are three kind of errors: 'forgetting' widgets, duplication of widgets, and violating the hierarchical order. Here are examples of the last two error kinds.

```
set f  [layout := row 5 [widget ok, widget ok]]  -- duplication
set ok [layout := widget can]                    -- order
```

A potential solution to the *hierarchy problem* is the use of a *linear type* system [7, 45] to express the appropiate constraints. Another solution is to let the layout specification construct the components. One can implement a set of layout combinators that return a nested cartesian product of widget identifiers. The nested cartesian product is used to represent a heterogenous list of identifiers, and combinators that generate those can be implemented along the lines of Baars *et al* [6]. Here is a concrete example of this approach:

```
do (f,(lab,(ok,(can,())))) ← frame (above label
                                    (beside button button))
```

The returned identifiers can now be used to set various properties of all widgets. Using *fixIO* and the recursive *mdo* notation of Erkök and Launchbury [17], we can even arrange things so that widgets can refer to each other at creation time.

We have not adopted this solution for wxHaskell though. First, the syntax of the nested cartesian product is inconvenient for widgets with many components. Furthermore, the order of the identifiers is directly determined by layout; it is very easy to make a small mistake and get a type error in another part of the program. Due to type constraints, the layout combinators can no longer use convenient list syntax to present rows and columns, but fixed arity combinators have to be used. Further research is needed to solve these problems, and maybe record calculi or syntax macros may provide solutions. For now, we feel that the slight chance of invalid layout is acceptable with the given alternatives.

5 Inheritance

Since wxHaskell is based on an object-oriented framework, we need to model the inheritance relationship between different widgets. This relation is encoded using *phantom* types [27, 26]. In essence, wxHaskell widgets are just foreign pointers to C++ objects. For convenience, we use a type synonym to distinguish these object pointers from other pointers:

```
type Object a = Ptr a
```

The type argument *a* is a phantom type: no value of this type is ever present as pointers are just plain machine adresses. The phantom type *a* is only used to encode the inheritance relation of the objects in Haskell. For each C++ class we have a corresponding *phantom data type* to represent this class, for example:

```
data CWindow a
data CFrame a
data CControl a
data CButton a
```

We call this a phantom data type as the type is only used in phantom type arguments. As no values of phantom types are ever created, no constructor definition is needed. Currently, only GHC supports phantom data type declarations, and in the library we just supply dummy constructor definitons. Next, we define type synonyms that encode the full inheritance path of a certain class:

```
type Window a = Object  (CWindow a)
type Frame a  = Window  (CFrame a)
type Control a = Window (CControl a)
type Button a  = Control (CButton a)
```

Using these types, we can impose a strong type discipline on the different kinds of widgets, making it impossible to perform illegal operations on the object pointers. For example, here are the types for the widget creation functions of Figure 1:

```
frame  ::                [Prop (Frame ())] → IO (Frame ())
button :: Window a →     [Prop (Button ())] → IO (Button ())
label  :: Window a →     [Prop (Label ())]  → IO (Label ())
```

For now, we can ignore the type of the property lists which are described in more detail in the Section 6. We see how each function creates an object of the appropiate type. A type *C* () denotes an object of exactly class *C*; a type *C a* denotes an object that is at least an instance of class *C*. In the creation functions, the co(ntra) variance is encoded nicely in these types: the function *button* creates an object of exactly class *Button*, but it can be placed in any object that is an instance of the *Window* class. For example:

```
do f ← frame []
   b ← button f []
```

The frame *f* has type *Frame* (). We can use *f* as an argument to *button* since a *Frame* () is an instance of *Window a* – just by expanding the type synonyms we have:

```
Frame () = Window (CFrame ()) ≅ Window a
```

The encoding of (single interface) inheritance using polymorphism and phantom types is simple and effective. Furthermore, type errors from the compiler are usually quite good – especially in comparison with an encoding using Haskell type classes.

6 Attributes and properties

In this section we discuss how we type and implement the *attributes* of widgets. Attributes first appeared in Haskell/DB [27] in the context of databases but proved useful for GUI's too. In Figure 1 we see some examples of widget attributes, like *text* and *layout*. The type of an attribute reflects both the type of the object it belongs to, and the type of the values it can hold. An attribute of type *Attr w a* applies to objects of type *w* that can hold values of type *a*. For example, the *text* attribute for buttons has type:

```
text :: Attr (Button a) String
```

The current value of an attribute can be retrieved using *get*:

```
get :: w → Attr w a → IO a
```

The type of *get* reflects the simple use of polymorphism to connect the type of an attribute to both the widgets it applies to (*w*), and the type of the result (*a*).

Using the $(:=)$ operator, we can combine a value with an attribute. The combination of an attribute with a value is called a *property*. Properties first appeared in Koen Claessen's (unreleased) Yahu library [12], and prove very convenient in practice. In wxHaskell, we use a refined version of the Yahu combinators. Since the value is given, the type of properties is only associated with the type of objects it belongs to. This allows us to combine properties of a certain object into a single homogenous list.

$(:=) :: Attr\ w\ a \to a \to Prop\ w$

Finally, the *set* function assigns a list of properties to an object:

$set :: w \to [Prop\ w] \to IO\ ()$

As properties still carry their object parameter, polymorphism ensures that only properties belonging to an object of type *w* can be used. Here is a short example that attaches an exclamation mark to the text label of a button:

```
exclamation :: Button a → IO ()
exclamation b =
  do s ← get b text
     set b [text := s ++ "!"]
```

The update of an attribute is a common operation. The update operator $(:\sim)$ applies a function to an attribute value:

$(:\sim) :: Attr\ w\ a \to (a \to a) \to Prop\ w$

Using this operator in combination with the Haskell section syntax, we can write the previous example as a single concise expression:

$exclamation\ b = set\ b\ [text :\sim\ (++"!")]$

6.1 Shared attributes

Many attributes are shared among different objects. For example, in Figure 1, the *text* attribute is used for frames, buttons, and labels. Since the wxWidgets *Window* class provides for a *text* attribute, we could use inheritance to define the *text* attribute for any kind of window:

$text :: Attr\ (Window\ a)\ String$

However, this is not such a good definition for a library, as user defined widgets can no longer support this attribute. In wxHaskell, the *text* attribute is therefore defined in a type class, together with an instance for windows:

```
class Textual w where
  text :: Attr w String
instance Textual (Window a) where
  text = ...
```

Here, we mix object inheritance with *ad hoc* overloading: any object that derives from the *Window* class, like buttons and labels, are also an instance of the *Textual* class and support the *text* attribute. This is also very convenient from an implementation perspective – we can implement the *text* attribute in terms of wxWidgets primitives in a single location. If the inheritance was not encoded in the type parameter, we would have to define the *text* attribute for every widget kind separately, i.e. an instance for buttons, another instance for labels, etc. Given that a realistic GUI library like wxWidgets supports at least fifty separate widget kinds, this would quickly become a burden.

The price of this convenience is that we do not adhere to the Haskell98 standard (in the *WX* library). When we expand the type synonym of *Window a*, we get the following instance declaration:

instance *Textual* $(Ptr\ (CObject\ (CWindow\ a)))$

This instance declaration is illegal in Haskell 98 since an instance type must be of the form $(T\ a_1 \ldots\ a_n)$. This restriction is imposed to prevent someone from defining an overlapping instance, for example:

instance *Textual* $(Ptr\ a)$

In a sense, the Haskell98 restriction on instance types is too strict: the first instance declaration is safe and unambiguous. Only new instances that possibly overlap with this instance should be rejected. The GHC compiler lifts this restriction and we use the freedom to good effect in the *WX* library.

6.2 Implementation of attributes

Internally, the attribute data type stores the primitive set and get functions. Note that this single definition shows that polymorphism, higher-order functions, and first class computations are very convenient for proper abstraction.

data $Attr\ w\ a = Attr\ (w \to IO\ a)\ (w \to a \to IO\ ())$

As an example, we give the full definition of the *text* attribute that uses the primitive *windowGetLabel* and *windowSetLabel* functions of the *WXCore* library:

```
instance Textual (Window a) where
  text = Attr windowGetLabel windowSetLabel
```

The *get* function has a trivial implementation that just extracts the corresponding function from the attribute and applies it:

```
get :: w → Attr w a → IO a
get w (Attr getter setter) = getter w
```

The attentive reader will have noticed already that the assignment operators, $(:=)$ and $(:\sim)$, are really constructors since they start with a colon. In particular, they are the constructors of the property data type:

```
data Prop w = ∀a. (Attr w a) := a
            | ∀a. (Attr w a) :~ (a → a)
```

We use local quantification [24] to hide the value type *a*, which allows us to put properties in homogenous lists. This is again an extension to Haskell98, but it is supported by all major Haskell compilers. The *set* function opens the existentially quantified type through pattern matching:

```
set :: w → [Prop w] → IO ()
set w props
  = mapM_ setone props
  where
    setone (Attr getter setter := x) = setter w x
    setone (Attr getter setter :~ f) = do x ← getter w
                                          setter w (f x)
```

It is well known that an explicit representation of function application requires an existential type. We could have avoided the use of existential types, by defining the assignment and update operators directly as functions. Here is a possible implementation:

```
type Prop w = w → IO ()
(:=) :: Attr w a → a → Prop w
(:=) (Attr getter setter) x = \w → setter w x
set :: w → [Prop w] → IO ()
set w props = mapM_ (\f → f w) props
```

This is the approach taken by the Yahu library. However, this does not allow reflection over the property list, which is used in wxHaskell to implement *creation* attributes (which are beyond the scope of this article).

7 Layout

This section discusses the design of the layout combinators of wxHaskell. The visual layout of widgets inside a parent frame is specified with the *layout* attribute that holds values of the abstract data type *Layout*. Here are some primitive layouts:

```
caption :: String → Layout
space   :: Int → Int → Layout
rule    :: Int → Int → Layout
boxed   :: String → Layout → Layout
```

The *caption* layout creates a static text label, *space* creates an empty area of a certain width and height, and *rule* creates a black area. The *boxed* layout container adds a labeled border around a layout.

Using the *widget* combinator, we can layout any created widget that derives from the *Window* class.The *container* combinator is used for widgets that contain other widgets, like scrolled windows or panels:

```
widget    :: Window a → Layout
container :: Window a → Layout → Layout
```

To allow for user defined widgets, the *widget* combinator is actually part of the *Widget* class, where *Window a* is an instance of *Widget*.

Basic layouts can be combined using the powerful *grid* combinator:

```
grid :: Int → Int → [[Layout]] → Layout
```

The first two arguments determine the amount of space that should be added between the columns and rows of the grid. The last argument is a list of rows, where each row is a list of layouts. The *grid* combinator will lay these elements out as a table where all columns and rows are aligned.

We can already define useful abstractions with these primitives:

```
empty      = space 0 0
hrule w    = rule w 1
vrule h    = rule 1 h
row w xs   = grid w 0 [xs]
column h xs = grid 0 h [[x] | x ← xs]
```

Here is an example of a layout that displays two text entries for retrieving an x- and y-coordinate. The *grid* combinator is used to align the labels and text entries, with 5 pixels between the components.

```
grid 5 5 [[caption "x:", widget xinput]
         ,[caption "y:", widget yinput]]
```

7.1 Alignment, expansion, and stretch

We can see that with the current set of primitive combinators, we can always calculate the *minimum size* of a layout. However, the area in which a layout is displayed can be larger than its minimum size, due to alignment constraints imposed by a *grid*, or due to user interaction when the display area is resized. How a layout is displayed in a larger area is determined by three attributes of a layout: the *alignment*, the *expansion*, and the *stretch*.

The *alignment* of a layout determines where a layout is positioned in the display area. The alignment consists of horizontal and vertical aligment. Ideally, each component can be specified continuously between the edges, but unfortunately, the wxWidgets library only allows us to align centered or towards the edges. There are thus six primitives to specify the alignment of a layout:

```
halignLeft   :: Layout → Layout   -- default
halignRight  :: Layout → Layout
halignCenter :: Layout → Layout

valignTop    :: Layout → Layout   -- default
valignBottom :: Layout → Layout
valignCenter :: Layout → Layout
```

The *expansion* of a layout determines how a layout expands into the display area. There are four possible expansions. By default, a layout is *rigid*, meaning that it won't resize itself to fit the display area. A layout is *shaped* when it will proportionately expand to fill the display area, i.e. it maintaints its aspect ratio. For a *shaped* layout, the alignment is only visible in one direction, depending on the display area.

The other two modes are *hexpand* and *vexpand*, where a layout expands only horizontally or vertically to fit the display area. Again, wxWidgets does not allow the last two modes separately, and we only provide an *expand* combinator that expands in both directions. For such layout, alignment is ignored completely.

```
rigid  :: Layout → Layout   -- default
shaped :: Layout → Layout
expand :: Layout → Layout
```

The *stretch* of a layout determines if a layout demands a larger display area in the horizontal or vertical direction. The previous two attributes, *alignment* and *expansion*, determine how a layout is rendered when the display area is larger than the minimum. In contrast,

the *stretch* determines whether the layout actually gets a larger display area assigned in the first place! By giving a layout stretch, it is assigned all extra space left in the parents' display area.

```
static  :: Layout → Layout   -- default
hstretch :: Layout → Layout
vstretch :: Layout → Layout
stretch  = hstretch ∘ vstretch
```

As a rule, *stretch* is automatically applied to the top layout, which ensures that this layout gets at least all available space assigned to it. For example, the following layout centers an *ok* button horizontally in a frame *f*:

```
set f [layout := halignCenter (widget ok)]
```

Due to the implicit *stretch* this example works as it stands. If this stretch had not been applied, the layout would only be assigned its minimim size as its diplay area, and the centered alignment would have no visible effect. So stretch is not very useful for layouts consisting of a single widget; it only becomes useful in combination with grids.

7.2 Stretch and expansion for grids

Layout containers like *boxed* and *container* automatically inherit the stretch and expansion mode of their children. Furthermore, a *grid* has a special set of rules that determines the stretch of its rows and columns. A column of a grid is horizontally stretchable when all elements of that columnn have horizontal stretch. Dually, a row is vertically stretchable when all elements of that row have vertical stretch. Furthermore, when any row or column is stretchable, the grid will stretch in that direction too and the grid will *expand* to fill assigned area.

This still leaves the question of how extra space is divided amongst stretchable rows and columns. The *weight* attribute is used to proportionally divide space amongst rows and columns. A layout can have a horizontal and vertical (positive) weight:

```
hweight :: Int → Layout → Layout
vweight :: Int → Layout → Layout
```

The default weight of a layout is one. The weight of a row or column is the maximum weight of its elements. The weight of the rows and columns is not propagated to the grid layout itself, which has its own weight.

There are two rules for dividing space amongst rows and columns: first, if all weights of stretchable elements are equal, the space is divided equally amongst those elements. If the weights are differing, the space is divided proportionally according to the weight of the element – i.e. a layout with weight two gets twice as much space as a layout with weight one. The first rule is useful for attaching zero weights to elements, that will cancel out as soon as another element becomes stretchable (with a weight larger than zero). Alas, the current wxWidgets implementation does not provide proportional stretching yet, and wxHaskell disregards all weight attributes at the moment of writing.

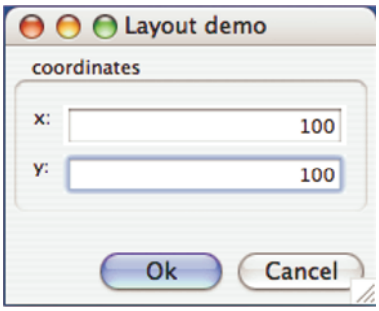


Figure 3. Layout on MacOS X.

7.3 Common layout transformers

With the given set of primitive combinators, we can construct a set of combinators that capture common layout patterns. For example, alignment in the horizontal and vertical direction can be combined:

```
alignCenter       = halignCenter ∘ valignCenter
alignBottomRight  = halignRight  ∘ valignBottom
```

By combining stretch with alignment, we can float a layout in its display area:

```
floatCenter       = stretch ∘ alignCenter
floatBottomRight  = stretch ∘ alignBottomRight
```

Dually, by combining stretch and expansion, layouts will fill the assigned display area:

```
hfill = hstretch ∘ expand
vfill = vstretch ∘ expand
fill  = hfill ∘ vfill
```

Using stretchable empty space, we can emulate much of the behaviour of TEX boxes, as stretchable empty space can be imagined as glue between layouts.

```
hglue = hstretch empty
vglue = vstretch empty
glue  = stretch  empty
```

Using the *glue* combinators in combination with *weight*, it is possible to define the 'primitive' alignment combinators in terms of glue. For example:

```
halignCenter l = row 0 [hweight 0 hglue, l, hweight 0 hglue]
```

Note that we set the horizontal weight of the *hglue* to zero. When the layout *l* stretches horizontally and expands, the entire display area should be assigned to the *l* in order to expand over all the available space. Since the default weight of *l* is one, a proportional division of the available space indeed assigns everything to *l*, mimicking the behaviour of its primitive counterpart.

7.4 Example

Here is a complete example that demonstrates a complicated layout and the convenience of the grid propagation rules. We layout a

frame that displays a form for entering an x and y coordinate, as shown in Figure 3.

```
layoutDemo
  = do f   ← frame    [text := "Layout demo"]
       p   ← panel  f []
       x   ← entry  p [text := "100"]
       y   ← entry  p [text := "100"]
       ok  ← button p [text := "Ok"]
       can ← button p [text := "Cancel"]
       set f [layout :=
         container p $ margin 5 $
         column 5 [boxed "coordinates" $
                     grid 5 5 [[caption "x:", hfill (widget x)]
                              ,[caption "y:", hfill (widget y)]]
                  ,floatBottomRight $
                     row 5 [widget ok, widget can]
                  ]]
```

The *panel* creates an empty widget that manages keyboard navigation control for child widgets[3]. When this frame is resized, the text entries fill the available space horizontally (due to *hfill*), while the ok and cancel buttons float to the bottom right. Due to the propagation rules, the *grid* stretches horizontally and expands, just like the *boxed* layout. Furthermore, the *column* stretches in both directions and expands, and thus the entire layout is resizeable. When the *floatBottomRight* is replaced by an *alignBottomRight*, there is no stretch anymore, and the horizontal stretch of the *boxed* layout is not propagated. In this case, the top layout is no longer resizeable.

We can express the same layout using a TEX approach with *glue*:

```
container p $ margin 5 $
column 0 [boxed "coordinates" $
            grid 5 5 [[caption "x:", hfill (widget x)]
                     ,[caption "y:", hfill (widget y)]]
         ,stretch (vspace 5)
         ,row 0 [hglue, widget ok, hspace 5, widget can]
         ]]
```

Note that we need to be more explicit about the space between elements in a row and column.

7.5 Implementing layout

The implementation of layout combinator library is interesting in the sense that the techniques are generally applicable for declarative abstractions over imperative interfaces [27, 26]. In the case of the wxWidgets library, the imperative interface consists of creating *Sizer* objects that encode the layout constraints imposed by the wxHaskell layout combinators.

Instead of directly creating *Sizer* objects, we first generate an intermediate data structure that represents a canonical encoding of the layout. Besides leading to clearer code, it also enables analysis and transformation of the resulting data structure. For example, we can implement the propagation rules as a separate transformation. Only when the layout is assigned, the data structure is translated into an *IO* value that creates proper *Sizer* objects that implement the layout.

[3] wxHaskell *panel*'s have nothing to do with Java panels that are used for layout.

The *Layout* data type contains a constructor for each primitive layout. Each constructor contains all information to render the layout:

```
data Layout
  = Grid   {attrs :: Attrs, gap :: Size, rows :: [[Layout]]}
  | Widget{attrs :: Attrs, win :: Window ()}
  | Space  {attrs :: Attrs, area :: Size}
  | Label  {attrs :: Attrs, txt  :: String}
  ...
```

All primitive layouts contain an *attrs* field that contains all common layout attributes, like alignment and stretch:

```
data Attrs      = Attrs{stretch_h   :: Bool
                       , stretch_v  :: Bool
                       , align_h    :: Align_h
                       , align_v    :: Align_v
                       , expansion  :: Expansion
                       ...}
data Expansion  = Rigid | Shaped | Expand
data Align_h    = AlignLeft | AlignRight | AlignCenter_h
data Align_v    = AlignTop | AlignBottom | AlignCenter_v
```

The implementation of the basic layout combinators is straightforward:

```
space w h = Space  defaultAttrs (size w h)
widget w  = Widget defaultAttrs (downcastWindow w)
...
```

The implementation of the layout transformers is straightforward too, but somewhat cumbersome due to the lack of syntax for record updates:

```
rigid l    = l{attrs = (attrs l){expansion = Rigid}}
hstretch l = l{attrs = (attrs l){stretch_h  = True}}
...
```

The *grid* combinator is more interesting as we have to apply the propagation rules for stretch and expansion. These rules have to be applied immediately to the attributes to implement the layout transformers faithfully. A separate pass algorithm is also possible, but that would require a more elaborate *Layout* data type with an explicit representation of layout transformers.

```
grid w h rows = Grid gridAttrs (size w h) rows
  where
    gridAttrs = defaultAttrs{
      stretch_v  = any (all (stretch_v ∘ attrs)) rows,
      stretch_h  = any (all (stretch_h ∘ attrs)) (transpose rows),
      expansion = if (stretch_v gridAttrs ∨ stretch_h gridAttrs)
                    then Expand else Static}
```

We can elegantly view rows as columns using *transpose*. Note also that the use of laziness in the definition of *expansion* is not essential.

Now that we made the layout explicit in the *Layout* data structure, we can write a function that interprets a *Layout* structure and generates the appropiate wxWidgets' *Sizer* objects:

```
sizerFromLayout :: Window a → Layout → IO (Sizer ())
```

We will not discuss this function in detail as the interface to *Sizer* objects is beyond the scope of this article. However, with an explicit representation of layout, it is fairly straightforward to create the corresponding *Sizer* objects. The ability to freely combine and manipulate *IO* values as first-class entities during the interpretation of the *Layout* description proves very useful here.

8 Communication with C++

Even though Haskell has an extensive foreign function interface [11], it was still a significant challenge to create the Haskell binding to the wxWidgets C++ library. This section describes the technical difficulties and solutions.

8.1 The calling convention

No current Haskell compiler supports the C++ calling convention, and we do not expect that this situation will change in the near future. The solution adapted by wxHaskell is to expose every C++ function as a C function. Here is an example of a wrapper for the *SetLabel* method of the *Window* class:

```
extern "C"
void wxWindowSetLabel( wxWindow* self, const char* text) {
  self → SetLabel(text);
}
```

We also create a C header file that contains the signature of our wrapper function:

```
extern void wxWindowSetLabel( wxWindow*, const char*);
```

The *wxWindowSetLabel* function has the C calling convention and can readily be called from Haskell using the foreign function interface. We also add some minimal marshalling to make the function callable using Haskell types instead of C types.

```
windowSetLabel :: Window a → String → IO ()
windowSetLabel self text =
  whenValid self (withCString text (wxWindowSetLabel self))
foreign import ccall "wxWindowSetLabel"
  :: Ptr a → Ptr CChar → IO ()
```

To avoid accidental mistakes in the foreign definition, we include the C header file when compiling this Haskell module. GHC can be instructed to include a C header file using the *−#include* flag.

Unfortunately, this is not the entire story. The C++ library is linked with the C++ runtime library, while the Haskell program is linked using the C runtime library – resulting in link errors on most platforms. wxHaskell avoids these problems by compiling the C++ code into a dynamic link library. A drawback of this approach is that the linker can not perform dead code elimination and the entire wxWidgets library is included in the resulting dynamic link library. Of course, it can also save space, as this library is shared among all wxHaskell applications.

8.2 wxDirect

If there were only few functions in the wxWidgets library, we could write these C wrappers by hand. However, wxWidgets contains more than 500 classes with about 4000 methods. Ideally, we would have a special tool that could read C++ header files and generate the needed wrappers for us. The SWIG toolkit [8] tries to do exactly this, but writing a SWIG binding for Haskell and the corresponding binding specification for wxWidgets is still a lot of work. Another option is to use a tool like SWIG to generate IDL from the C++ headers and to use H/Direct [18, 19, 26] to generate the binding.

For wxHaskell, we opted for a more pragmatic solution. The wxEiffel library [39] contains already thousands of hand written C wrappers for wxWidgets together with a header file containing the signatures. wxHaskell uses the same C wrappers for the Haskell binding. The Haskell wrappers and foreign import declarations are generated using a custom tool called wxDirect. This tool uses Parsec [28] to parse the signatures in the C header and generates appropiate Haskell wrappers. As the data types in wxWidgets are limited to basic C types and C++ objects, the marshalling translation much simpler than that of a general tool like H/Direct.

As argued in [18, 19, 26], a plain C signature has not enough information to generate proper marshalling code. Using C macros, we annotated the C signatures with extra information. The signature of *wxWindowSetLabel* is for example:

```
void wxWindowSetLabel( Self(wxWindow) self, String text);
```

Macros like *Self* provide wxDirect with enough information to generate proper marshalling code and corresponding Haskell type signatures. When used by the C compiler, the macros expand to the previous plain C signature. This approach means that changes in the interface of wxWidgets require manual correction of the C wrappers, but fortunately, this interface has been stable for years now.

8.3 Performance

The performance of wxHaskell applications with regard to GUI operations is very good, and wxHaskell applications are generally indistinguishable from “native” applications written with MFC or GTK for example. This is hardly surprising, as all the hard work is done by the underlying C++ library – Haskell is just used as the glue language for the proper wxWidget calls. For the same reason, the memory consumption with regard to GUI operations is also about the same as that of native applications.

One of the largest wxHaskell programs is NetEdit: a Bayesian belief network editor that consists of about 4000 lines of wxHaskell specific code. On Windows XP, NetEdit uses about 12mb of memory for large belief networks of more than 50 nodes. The performance of the drawing routines is so good that NetEdit can use a naïve redraw algorithm without any noticable delays for the user.

The binaries generated with wxHaskell tend to be rather large though – GHC generates a 3mb binary for NetEdit. The use of a compressor like UPX can reduce the size of the executable to about 600kb. The shared library for wxHaskell generated by GCC is also about 3mb. On Windows platforms, we use Visual C++ to generate the shared library which approximately reduces the size to 2mb, which becomes 700kb after UPX compression.

9 Related work

There has been a lot of research on functional GUI libraries. Many of these libraries have a monadic interface. Haggis [20] is build on X Windows and uses concurrency to achieve a high level of composition between widgets. The Gtk2Hs and Gtk+Hs [42] libraries use the Gtk library and, like wxHaskell, provide an extensive array of widgets. Many libraries use the portable Tk framework as their GUI platform. HTk [23] is an extensive library that provides a sophisticated concurrent event model [36]. TkGofer [44, 13] is an elegant library for the Gofer interpreter that pioneered the use of type classes to model inheritance relations. Yahu [12] is an improved (but unreleased) version of TkGofer for Haskell that first used property lists to set attributes. The HToolkit [5] library has similar goals as wxHaskell but implements its own C wrapper around the win32 and Gtk interface.

Besides monadic libraries, there has been a lot of research into more declarative GUI libraries. Functional reactive animations (Fran) [16] elegantly described declarative animations as continuous functions from time to pictures. This idea was used in FranTk [38, 37] to model graphical user interfaces. Functional reactive programming [46] is a development where arrows [22, 32] are used to fix space-time leaks in Fran. The Fruit library [14, 15] uses these techniques in the context of GUI programming. In contrast to Fran, imperative streams [40] use discrete time streams instead of continuous functions to model animation and GUI's.

One of the most well known functional models for GUI programming is Fudgets [9, 10, 33]. The extensive Fudget library uses X windows and is supported by many Haskell compilers. The Fudget combinators give a rigid structure to the data flow in a program, and the Gadgets framework [29] introduces the concept of *wire*s to present a more flexible interface.

Many GUI libraries are implemented for other functional languages. A well known library is the ObjectIO library for Clean [2, 4, 3] that has partly been ported to Haskell [1]. This library uses uniqueness types [7] to safely encapsulate side effects. LablGtk [21] is a binding for O'Caml to the Gtk library and uses a label calculus to model property lists. Exene [34] is a concurrent GUI library for ML that uses X Windows.

H/Direct [18, 19, 26] described phantom types to model single interface inheritance. Another technique to model inheritance, that relies on multiple parameter type classes and functional dependencies, was described by Pang and Chakravarty [31, 30]. Phantom types were discussed as a general technique to impose a strong type discipline on untyped interfaces by Leijen [27, 26].

10 Conclusion

We have learned an important lesson from wxHaskell: do not write your own GUI library! By using the portable and well-maintained wxWidgets C++ library, we were able to create an industrial strength GUI library for Haskell in a relatively short time frame. Furthermore, we have shown how distinctive features of Haskell, like parameteric polymorphism, higher-order functions, and first-class computations, can be used to present a concise and elegant monadic interface to program GUI's. The resulting programs tend to be much shorter, and more concise, than their counterparts in C++.

In the future, we hope to extend the *WX* library with more abstractions and more widgets. Furthermore, we hope that wxHaskell can become a platform for research into more declarative models for programming GUI's.

11 Acknowledgements

wxHaskell could not have existed without the effort of many developers on wxWidgets and wxEiffel, in particular Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, Uwe Sanders, and many others. Koen Claessen's Yahu library provided the inspiration for property lists in wxHaskell.

12 References

[1] P. Achten and S. Peyton Jones. Porting the Clean object I/O library to Haskell. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages (2000)*, pages 194–213, 2000.

[2] P. Achten and M. Plasmeijer. The beauty and the beast. Technical Report 93–03, Research Inst. for Declarative Systems, Dept. of Informatics, University of Nijmegen, Mar. 1993.

[3] P. Achten and M. J. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[4] P. Achten, J. van Groningen, and M. Plasmeijer. High level specification of I/O in functional languages. In J. Launchbury and P. Sansom, editors, *Workshop Notes in Computer Science*, pages 1–17. Springer-Verlag, 1993. Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 June 1992.

[5] K. A. Angelov. The HToolkit project. `http://htoolkit.sourceforge.net`.

[6] A. Baars, A. Löh, and D. Swierstra. Parsing permutation phrases. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 171–182. Elsevier, 2001.

[7] E. Barendsen and S. Smetsers. Uniqueness Type Inference. In M. Hermenegildo and S. Swierstra, editors, *7th International Symposium on Programming Language Implementation and Logic Programming (PLILP'95), Utrecht, The Netherlands*, volume 982 of *LNCS*, pages 189–206. Springer-Verlag, 1995.

[8] D. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++ . In *4th annual Tcl/Tk workshop*, Monterey, CA, July 1996.

[9] M. Carlsson and T. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Functional Programming and Computer Architectures (FPCA)*, pages 321–330. ACM press, June 1993.

[10] M. Carlsson and T. Hallgren. *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Gothenburg University, 1998.

[11] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton-Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi`, Dec. 2003.

[12] K. Claessen. The Yahu library. `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/afp/yahu.html`.

[13] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *2nd International Conference on Functional programming (ICFP)*, pages 251–262, 1997. Also appeared in ACM SIGPLAN Notices 32, 8, (Aug. 1997).

[14] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *ACM Sigplan 2001 Haskell Workshop*, Sept. 2001.

[15] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM Press, 2003.

[16] C. Elliott and P. Hudak. Functional reactive animation. In *The proceedings of the 1997 ACM Sigplan International Conference on Functional Programming (ICFP97)*, pages 263–273. ACM press, 1997.

[17] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, Sept. 2000.

[18] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A Binary Foreign Language Interface to Haskell. In *The International Conference on Functional Programming (ICFP)*, Baltimore, USA, 1998. Also appeared in ACM SIGPLAN Notices 34, 1, (Jan. 1999).

[19] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling hell from heaven and heaven from hell. In *The International Conference on Functional Programming (ICFP)*, Paris, France, 1999. Also appeared in ACM SIGPLAN Notices 34, 9, (Sep. 1999).

[20] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, 1995.

[21] J. Garrigue. The LablGtk library. `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[22] J. Hughes. Generalising monads to arrows. In *Science of Computer Programming*, volume 37, pages 67–111, May 2000.

[23] E. Karlsen, G. Russell, A. Lüdtke, and C. Lüth. The HTk library. `http://www.informatik.uni-bremen.de/htk`.

[24] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[25] D. Leijen. The wxHaskell library. `http://wxhaskell.sourceforge.net`.

[26] D. Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2003.

[27] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Second USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[28] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[29] R. Noble and C. Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. In M. Hermenegildo and S. D. Swierstra, editors, *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 321–340. Springer-Verlag, Sept. 1995.

[30] A. T. H. Pang. Binding Haskell to object-oriented component systems via reflection. Master's thesis, The University of New South Wales, School of Computer Science and Engineering, June 2003. `http://www.algorithm.com.au/files/reflection/reflection.pdf`.

[31] A. T. H. Pang and M. M. T. Chakravarty. Interfacing Haskell with object-oriented languages. In G. Michaelson and P. Trinder, editors, *15th International Workshop on the Implementation of Functional Languages (IFL'03)*, LNCS. Springer-Verlag, 2004.

[32] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[33] A. Reid and S. Singh. Implementing fudgets with standard widget sets. In *Glasgow Functional Programming workshop*, pages 222–235. Springer-Verlag, 1993.

[34] J. H. Reppy. *Higher Order Concurrency*. PhD thesis, Cornell University, 1992.

[35] B. Robinson. wxFruit: A practical GUI toolkit for functional reactive programming. `http://zoo.cs.yale.edu/classes/cs490/03-04b/bartholomew.robinson`.

[36] G. Russell. Events in Haskell, and how to implement them. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 157–168, 2001.

[37] M. Sage. The FranTk library. `http://www.haskell.org/FranTk`.

[38] M. Sage. FranTk – a declarative GUI language for Haskell. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 106–117. ACM Press, 2000.

[39] U. Sander et al. The wxEiffel library. `http://wxeiffel.sourceforge.net`.

[40] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 261–272. ACM Press, 1998.

[41] M. Schrage. *Proxima: a generic presentation oriented XML editor*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2004.

[42] A. Simons and M. Chakravarty. The Gtk2Hs library. `http://gtk2hs.sourceforge.net`.

[43] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, et al. The wxWidgets library. `http://www.wxwidgets.org`.

[44] T. Vullinghs, D. Tuinman, and W. Schulte. Lightweight GUIs for functional programming. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 341–356. Springer-Verlag, 1995.

[45] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

[46] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 146–156. ACM Press, 2001.

Hoofdstuk 17

Case Study: Finger Trees

As a case study of a more complicated data type we will take a look at:

> Ralf Hinze and Ross Paterson. *Finger trees: a simple general-purpose data structure*, Journal of Functional Programming, 2005. To appear.

For copyright reasons you will have to download this paper yourself from: `http://www.informatik.uni-bonn.de/~ralf/publications/FingerTrees.pdf`

Bijlage A

ISO/ASCII tabel

	0*16+...		1*16+...		2*16+...		3*16+...		4*16+...		5*16+...		6*16+...		7*16+...	
...+0	0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
...+1	1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
...+2	2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
...+3	3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
...+4	4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
...+5	5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
...+6	6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
...+7	7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
...+8	8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
...+9	9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
...+10	10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
...+11	11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
...+12	12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
...+13	13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
...+14	14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
...+15	15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

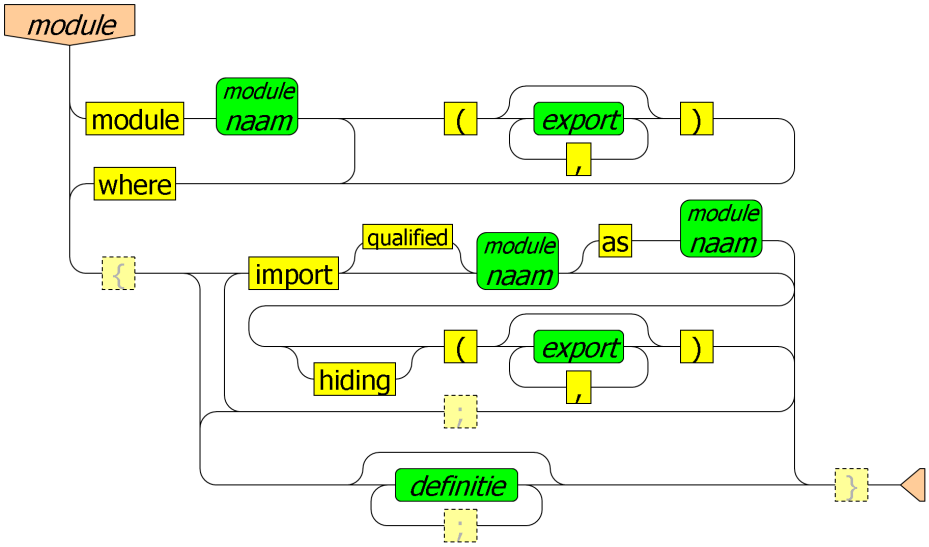
Haskell-notatie voor speciale tekens in een string gaat met behulp van een code achter een backslash:

- de *naam* van het speciale teken, bijvoorbeeld `"\ESC"` voor het escape-teken;
- het *nummer* van het speciale teken, bijvoorbeeld `"\27"` voor het escape-teken;
- het nummer in het achttallig stelsel (*octaal*), bijvoorbeeld `"\o33"` voor het escape-teken;
- het nummer in het zestientallig stelsel (*hexadecimaal*), bijvoorbeeld `"\x1B"` voor het escape-teken;
- door de overeenkomstige letter vier kolommen verder naar rechts, bijvoorbeeld `"\^["` voor het escape-teken;
- een van de volgende codes: `"\n"` (newline), `"\b"` (backspace), `"\t"` (tab), `"\a"` (alarm), `"\f"` (formfeed), `"\""` ("-symbool), `"\’"` (’-symbool), en `"\\"` (\-symbool)

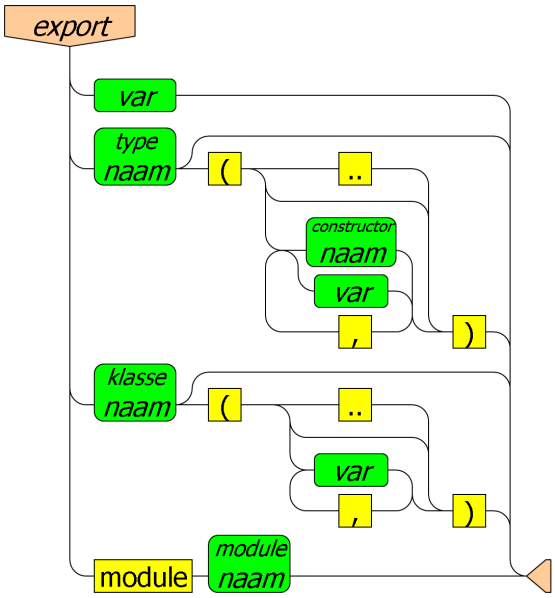
Bijlage B

Haskell syntaxdiagrammen

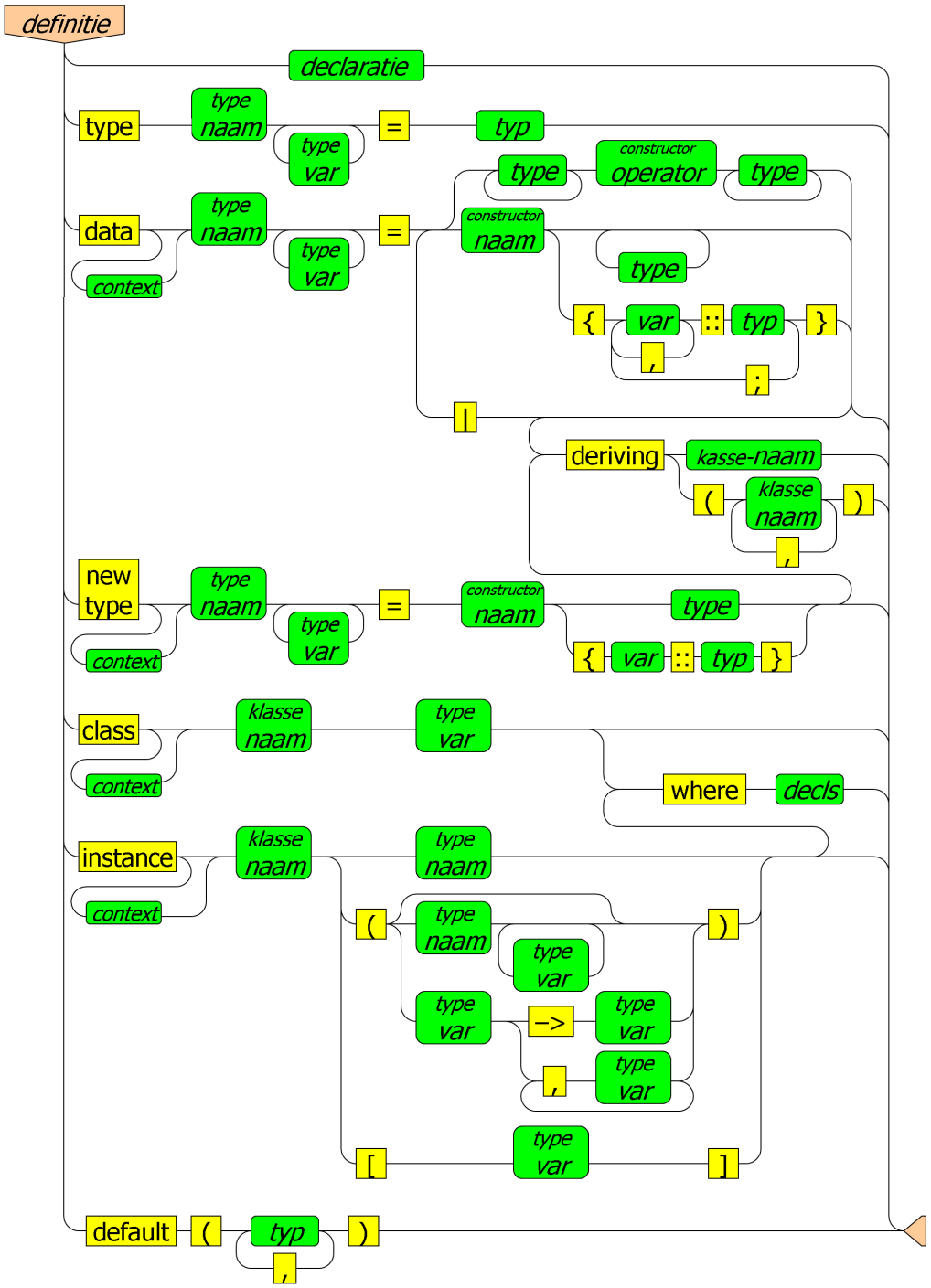
Elke Haskell-module bestaat uit een serie *definities*. Daaraan vooraf gaan nog een optionele module-header, en optionele import-specificaties.



In de module-header kan worden aangegeven welke variabele- (en functie-)namen, typenamen en klasse-namen in andere modules mogen worden gebruikt. Bij een type en een klasse kan desgewenst slechts een deel van de constructor-, respectievelijk memberfuncties worden geëxporteerd.

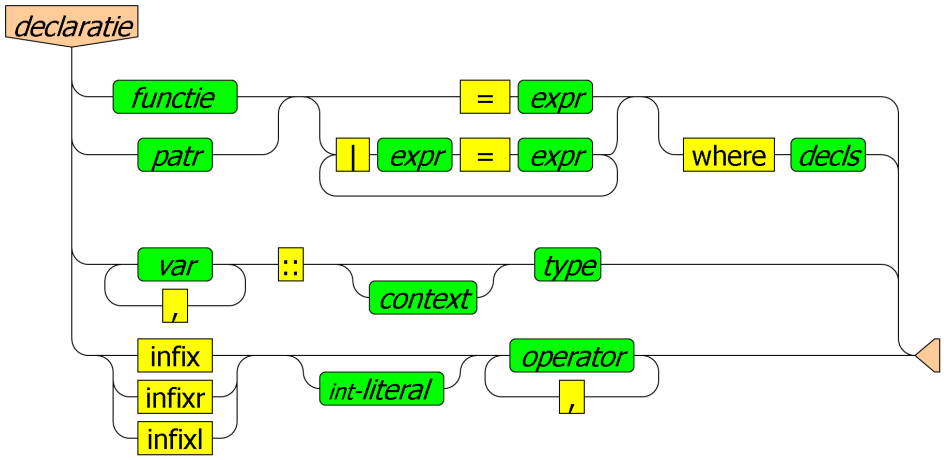


Een definitie kan alleen op het top-level van een module staan. Een definitie kan de vorm hebben van een *declaratie*, waaraan een apart schema is gewijd. Met definities voor type, data en newtype introduceer je een nieuw type. Met class introduceer je een nieuwe klasse, en met instance maak je een type tot lid van zo'n klasse.

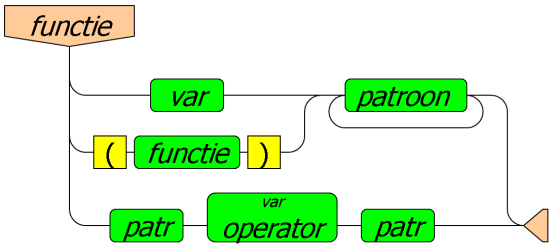


Een declaratie is één van de mogelijke toplevel definitie-vormen. Maar een declaratie kan ook onderdeel uitmaken van een groep *decls*, die kunnen voorkomen:

- als lokale declaraties achter where in één van de functie-declaraties
- als member-declaratie achter where in klasse- en instance-definities
- als lokale declaraties achter let in een do-expressie en in een lijstcomprehensie-expressie

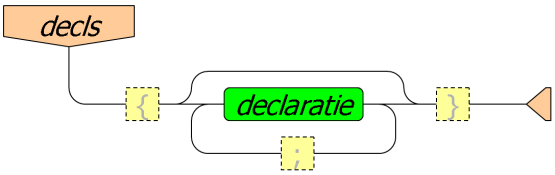


Links van het = teken van de declaratie kan een *functie* staan, waarmee je een variabelenaam voor een functie kunt introduceren, of een nieuwe operator kunt definiëren.



Links van een het = teken mag ook een *patr*oon staan, waarmee je een variabele of een patroon van variabelen een waarde kunt geven. In de *decls* die deel uitmaken van een class- of instance-definitie mag dit *patr* uitsluitend een *var* zijn.

Met de andere declaratie-vormen (met :: en infix) specificeer je het type van een functie en de associatie van een operator. Dit soort declaraties mogen niet voorkomen in een instance-definitie.



In het schema voor *decls* zijn de accolades en puntkomma's gedimd weergegeven. Ze mogen worden weggelaten als je de *layout-regel* hanteert: declaraties die tot dezelfde groep behoren worden precies even ver ingesprongen. Dit geldt ook voor de toplevel declaraties in de *module*, en voor de case- en do-constructies in het schema voor *expr*.

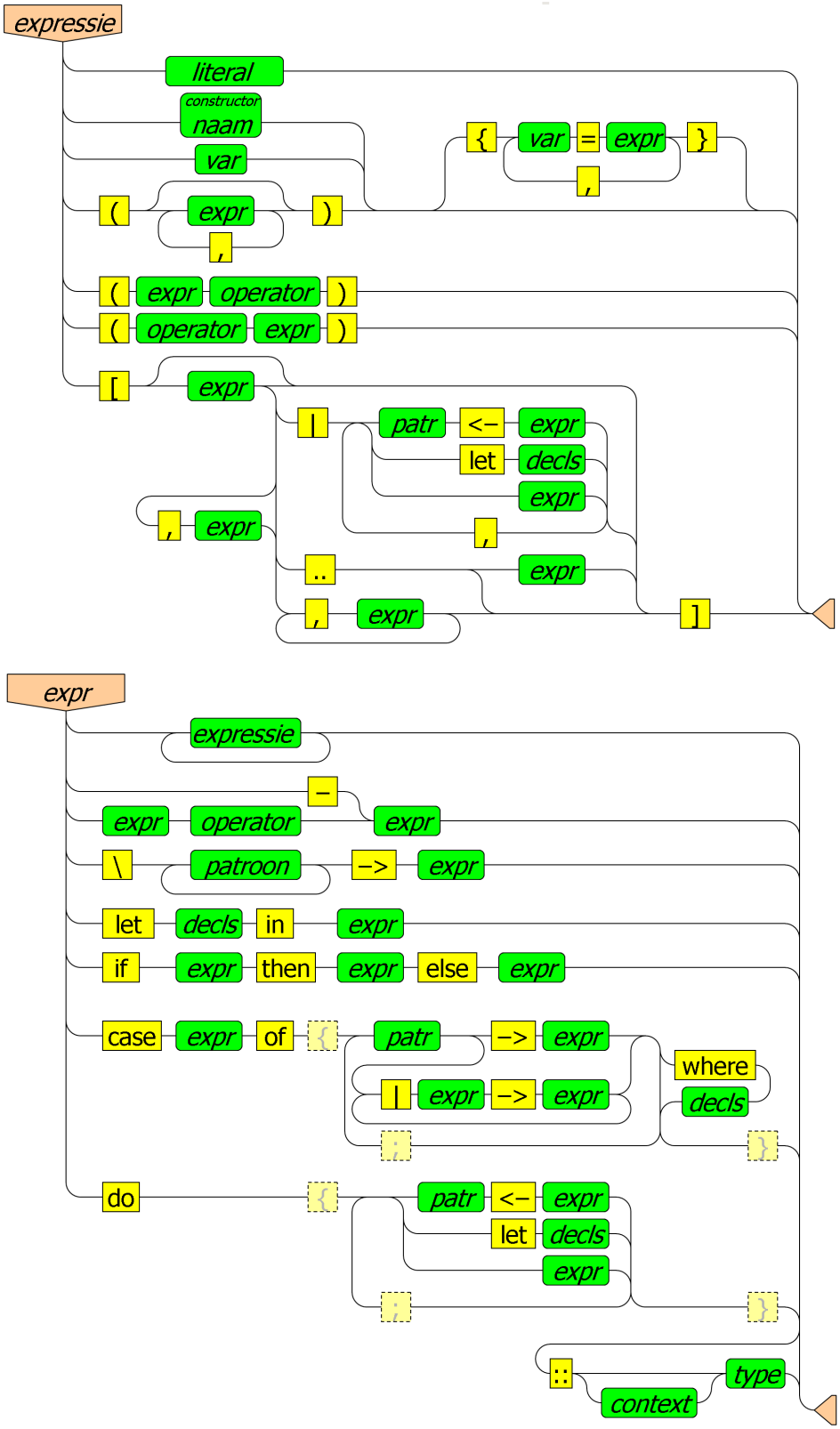
De rechterkant van een functie-declaratie bestaat uit een *expressie*. Er zijn twee wederzijds afhankelijke vormen:

- Een *expressie* is een eenvoudige vorm (literal, naam, of variabele), of een ingewikkeldere vorm, die dan wel tussen (ronde of vierkante) haakjes moet staan.
- De ingewikkelde vorm is beschreven als *expr*. Dat mag een simpele *expressie* zijn, een serie opeenvolgende expressies (dat is een aanroep van een functie met parameters), of een van een zestal andere vormen.

Expressies worden gebruikt:

- aan de rechterkant van een declaratie
- als boolean voorwaarde in een declaratie, en in een if- en een case-expressie

- op vele plaatsen als onderdeel van een grotere *expressie* of *expr*.



Niet in het schema is aangegeven dat het laaste niet-lege onderdeel van een do-constructie een *expr*

moet zijn, en dus niet een generator-met-pijltje of een let.

Ook niet in het schema is aangegeven dat de operatoren 10 nivo's van prioriteit kennen, en naar links of naar rechts kunnen associëren.

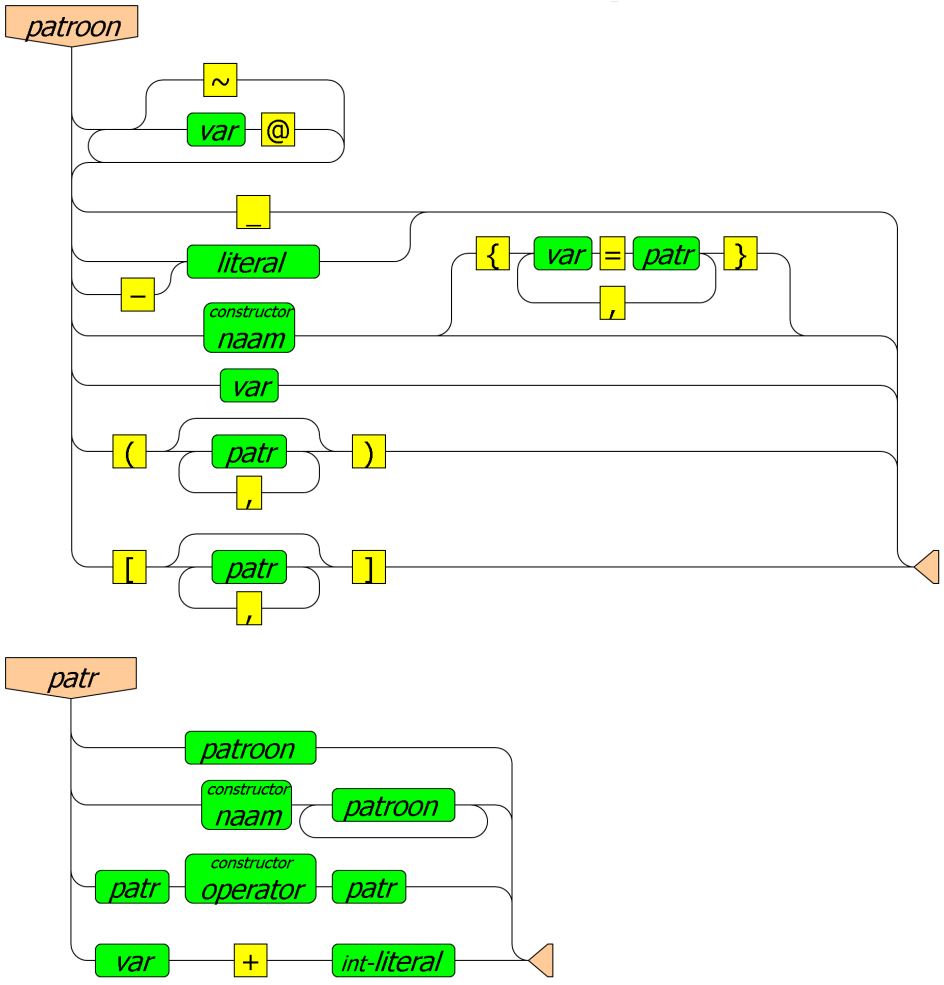
Er is één unaire operator -, met prioriteit 6. Er bestaan in Haskell geen andere unaire operatoren, en je kunt die ook niet zelf maken. De not-functie is geen operator maar een functie.

Aan de linkerkant van een functie-declaratie speelt een *patroon* een rol: als formele parameter van de te declareren functie of operator, of direct als doel van een declaratie. Net als van expressies zijn er twee wederzijds afhankelijke vormen:

- Een *patroon* is een eenvoudige vorm (literal, naam, of variabele), of een ingewikkeldere vorm, die dan wel tussen (ronde of vierkante) haakjes moet staan.
- De ingewikkelde vorm is beschreven als *patr*. Dat mag een simpel *patroon* zijn, of een van een drietal andere vormen.

Patronen worden gebruikt:

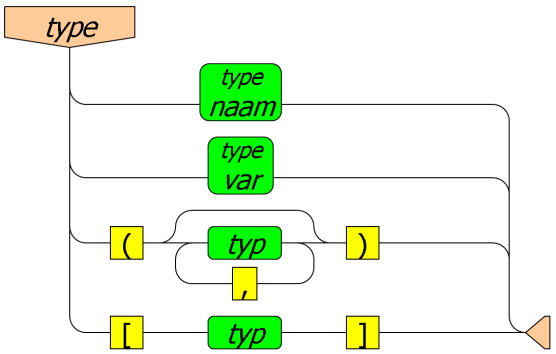
- aan de linkerkant van een declaratie
- als generator in een do-expressie en een lijstcomprehensie-expressie
- op diverse plaatsen als onderdeel van een groter *patroon* of *patr*.



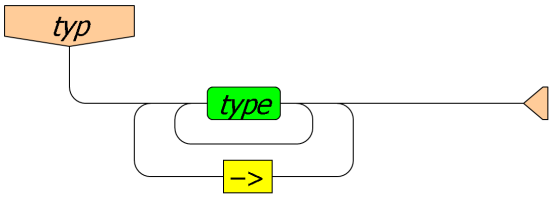
Behalve functie- en patroon-declaraties is er een declaratievorm waarmee je het type van een variabele specificeert, in een declaratie *var* :: *type*.

Het type kan een simpel *type* zijn, aangeduid door een typenaam, een typevariabele, of een inge-

wikkelder *typ* tussen ronde of vierkante haakjes.



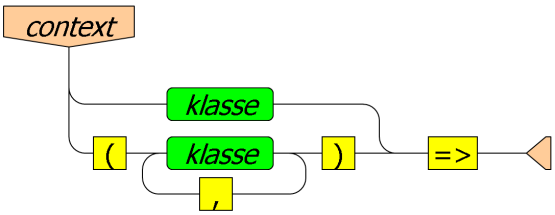
Zo'n ingewikkelder *typ* bestaat uit een *type*, al dan niet toegepast op andere types, waarvan er dan weer meerdere gescheiden mogen worden door functie-pijltjes.



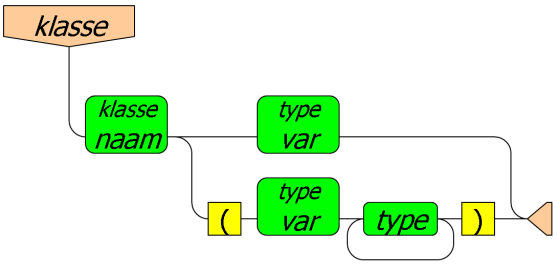
Een type wordt gebruikt:

- aan de rechterkant van een typespecificatie-*declaratie*
- in een optionele typespecificatie aan het eind van elke gewenste *expr*
- als onderdeel van een *definitie*: aan de rechterkant van een type-definitie, en als parameter van constructorfuncties in een data- en newtype-definitie.

Bij de specificatie van een type kan soms een *context* worden aangeduid. Zo'n context beschrijft één voorwaarde (zonder haakjes) of meerdere voorwaarden (met haakjes).



Met elke voorwaarde geef je aan dat een bepaalde type-variabele niet een willekeurig type aanduidt, maar een type dat behoort tot een bepaalde klasse. Die type-variabele kan eventueel weer geparametriseerd zijn met andere types, maar dan moet het geheel tussen haakjes staan.



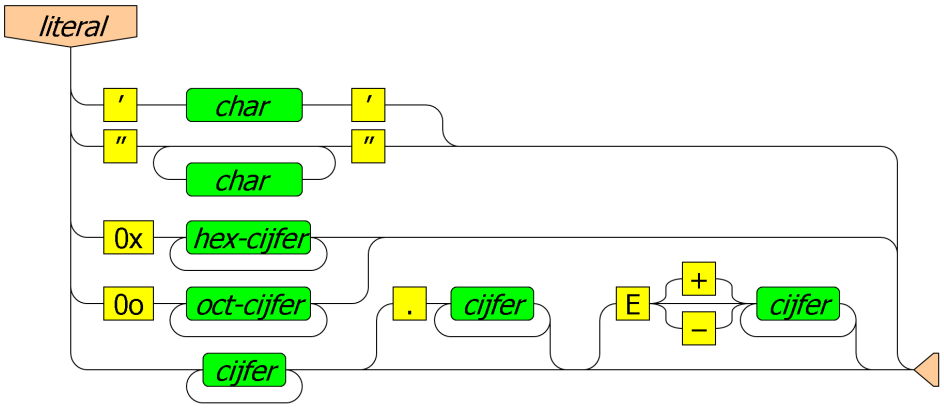
Een context wordt gebruikt:

- als voorwaarde in een typespecificatie (in een *declaratie* of *expr*)

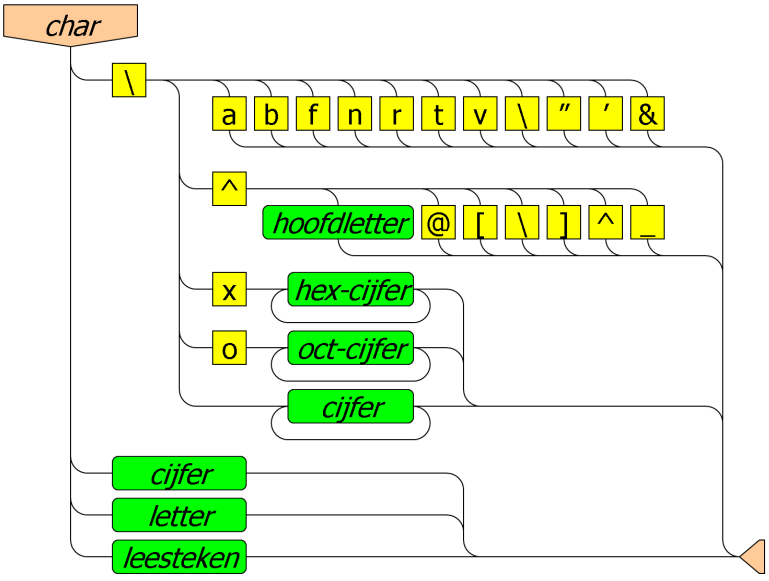
- als voorwaarde in een data- of newtype-definitie
- als voorwaarde in een class- of instance-definitie. In dit geval mag de type-variabele niet geparametriseerd worden met andere types.

Een *literal* is een character-, string-, integer- of float-constante. De char tussen enkele aanhalingstekens mag geen losse backslash of enkel aanhalingsteken zijn. De chars tussen dubbele aanhalingstekens mag geen losse backslash of dubbel aanhalingsteken zijn.

Constante getallen kunnen na een speciale prefix met hexadecimale of octale cijfers worden weergegeven. Zonder prefix is het de gewone decimale notatie. Als het optionele gedeelte achter een punt en/of achter de letter E wordt gebruikt, is het een Float constante. Anders is het een Int constante.



Met een backslash kunnen speciale characters worden aangeduid. Een backslash met een van de letters a, b, f, n, r, t, v staan voor alert, backspace, formfeed, newline, return, tab, vertical tab. Volgt op de backslash een tweede backslash of een enkel of dubbel aanhalingsteken, dan staat dat tweede symbool voor zichzelf. Volgt op de backslash een dakje, dan kunnen de 32 control-tekens direct worden aangeduid. Bijzondere symbolen kunnen ook met (eventueel hexadecimale of octale) cijfers worden aangeduid. De combinatie `\&` genereert 0 symbolen, en kan in bijzondere gevallen worden gebruikt om een backslash-cijferreeks combinatie af te sluiten.



Een variabele moet met een kleine letter beginnen. Variabelen kunnen aanduiden:

- waarden: getallen en andere primitieve waarden, datastructuren zoals (tupels, lijsten en

boomstructuren), en omdat het een functionele taal is ook functies
- types, als parameter van polymorfe types

De naam van een constante moet met een hoofdletter beginnen. Namen kunnen aanduiden:
- constante waarden: we spreken dan van constructoren of (vooral als het om functies gaat) constructor-functies
- types: ingebouwde types zoals Int, of zelfgedefinieerde, al dan niet geparametriseerde types
- klassen
- modules

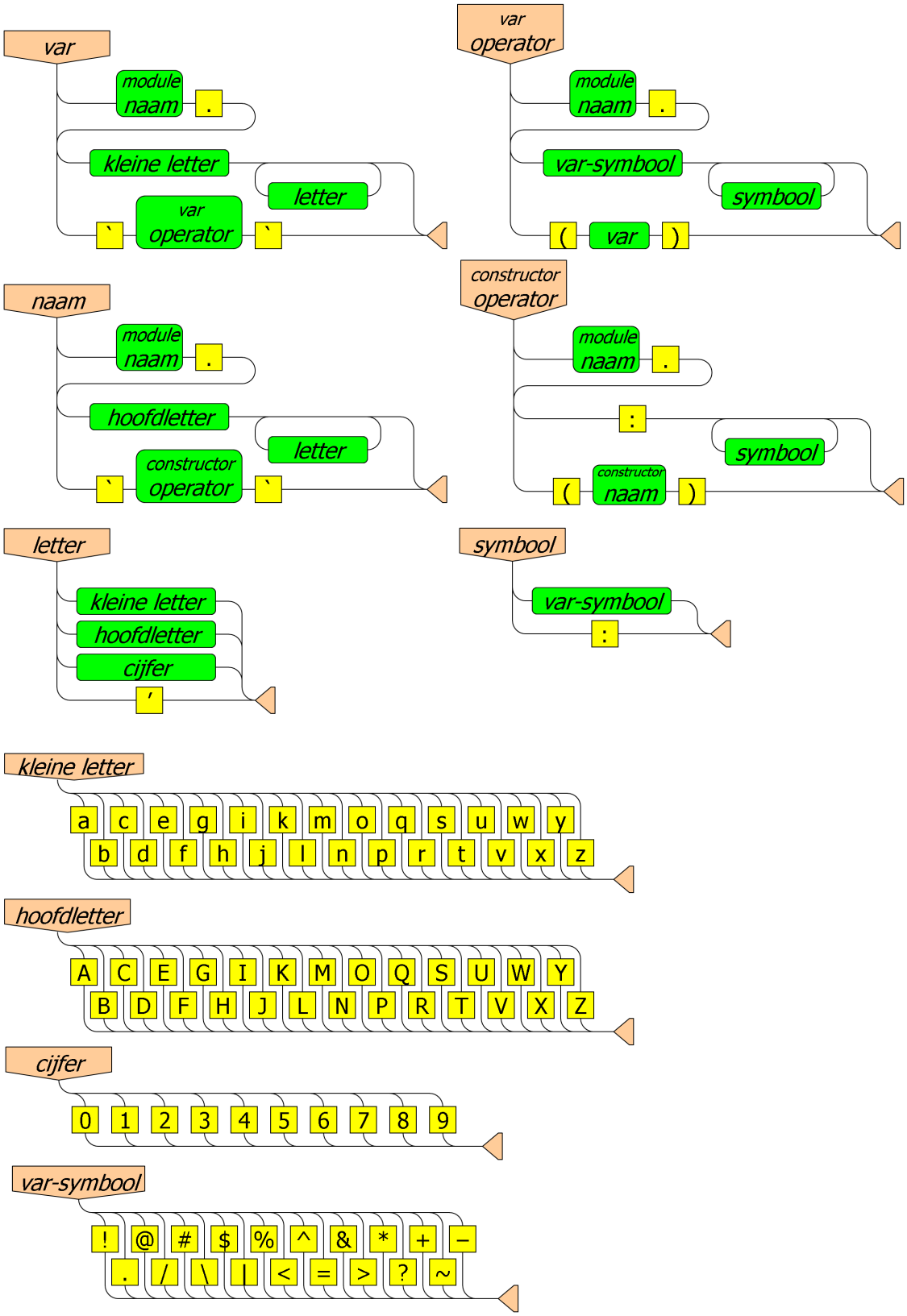
Als een variabele waarde een functie met twee parameters is, kan hij in plaats van met een variabele ook worden aangeduid met een *operator*. Zo'n operator bestaat niet uit letters, maar uit één of meer *var-symbolen*. Uit het diagram voor *expr* en *functie* blijkt dat zo'n operator tussen zijn twee parameters wordt geschreven in plaats van er voor.

Ook een constante functie met twee parameters (een constructorfunctie dus) kan als operator worden geschreven. Zo'n constructor-operator moet met het symbool : beginnen. Het belangrijkste voorbeeld is de ingebouwde lijst-constructorfunctie : zelf.

Een uit letters bestaande variabele of constante-naam kan als operator gebruikt worden door hem tussen back-quotes te zetten. Omgekeerd kan een uit symbolen bestaande operator juist als functie worden gebruikt door hem tussen haakjes te zetten.

In sommige (maar niet alle) situaties mag een variabele, naam of operator *gekwalificeerd* worden met een module-naam.

De precieze opbouw staat beschreven in de diagrammen. Anders dan in de rest van de syntax mag er tussen de onderdelen géén spaties of commentaar staan (ook niet tussen de modulenaam, de punt en de gekwalificeerde naam).



Niet als variabele mogen worden gebruikt de volgende gereserveerde woorden: case, class, data,

default, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type, en where.

De woorden qualified, as, en hiding hebben weliswaar een bijzondere betekenis in een klasse-header, maar mogen elders wel als variabele worden gebruikt.

Niet als *operator* mogen worden gebruikt de volgende gereserveerde symbolencombinaties:

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

Bijlage C

Test yourself

List-based functions

(i) The function *intersperse* :: $a \to [a] \to [a]$ puts its first argument between all the elements of its second argument. Thus *intersperse* `'a' "xyz"` results in `"xayaz"`. Which is the correct definition?

(a) *intersperse* a **as** = *tail* $\circ$ *concat* $\circ$ *map* $(\lambda x \to [a, x])$ \$ **as**

(b) *intersperse* a **as** = *tail* $[(a : e) \mid e \leftarrow \mathbf{as}]$

(c) *intersperse* a **as** = *foldr* $(\lambda e\ r \to (a : e : r))$ $[]$ **as**

(d) *intersperse* a **as** = *foldl* $(\lambda r\ e \to (a : e : r))$ $[]$ **as**

(ii) Which of the following is a correct definition of *inits*?

(a) *foldr* $(\lambda x\ r \to map\ (x:)\ r)$ $[]$

(b) *foldr* $(\lambda x\ r \to map\ (x:)\ r)$ $[[]]$

(c) *foldr* $(\lambda x\ r \to [[]] : map\ (x:)\ r)$ $[[]]$

(d) *foldr* $(\lambda x\ r \to [] : map\ (x:)\ r)$ $[[]]$

(iii) Which of the following is a correct definition of *tails*?

(a) *reverse* $\circ$ *map reverse* $\circ$ *inits* $\circ$ *reverse*

(b) *map reverse* $\circ$ *reverse* $\circ$ *tails* $\circ$ *reverse*

(c) *reverse* $\circ$ *inits* $\circ$ *map reverse* $\circ$ *reverse*

(d) *reverse* $\circ$ *map* (*inits* $\circ$ *reverse*) $\circ$ *reverse*

(iv) Which of the following is a correct definition of *transpose*?

(a) *foldr* (*zipWith* (:)) (*repeat* [])

(b) *foldr* (*zipWith* (++)) (*repeat* [])

(c) *foldr* (*zipWith* (:)) (*repeat* [[]])

(d) *foldl* (*zipWith* (++)) (*repeat* [])

(v) We compare the two expressions *foldr* $(\circ)$ *id* and *foldl* $(\circ)$ *id*. Which of the following holds?

(a) These expressions can replace each other everywhere.

(b) There are situations where the first one terminates, but the second one does not.

(c) There are situations where the second one terminates, but the first one does not.

(d) None of the above.

(vi) In the context of the functions

$$\begin{aligned} &\mathit{until}\ p\ f\ a \mid p\ a &&= a \\ &\qquad\qquad\quad \mid \mathit{True} &&= \mathit{until}\ p\ f\ (f\ a) \\ &\mathit{iterate}\ f\ x &&= x : \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

which of the following holds?

(a) $\mathit{until}\ p\ f \circ \mathit{until}\ p\ f$ can safely be replaced by $\mathit{until}\ p\ f$

(b) $\mathit{iterate}\ g \circ \mathit{iterate}\ f$ can safely be replaced by $\mathit{iterate}\ (g \circ f)$

(c) $\mathit{foldl}\ (-)\ 0$ can safely be replaced by $\mathit{foldr}\ (-)\ 0$

(d) $\mathit{until}\ p\ f \circ \mathit{until}\ q\ f$ can safely be replaced by $\mathit{until}\ q\ f$ if $p \Rightarrow q$ holds

(vii) What is the result of $\mathit{foldr}\ ((+) \circ (2*))\ 3\ [3, 3]$?

(a) 9

(b) 15

(c) 27

(d) 30

(viii) What is the result of $\mathit{foldr}\ ((*) \circ (*2))\ 2\ [2, 2, 2]$?

(a) 16

(b) 32

(c) 64

(d) 128

(ix) What is the result of $\mathit{foldr}\ ((-) \circ (+2))\ 2\ [2, 2, 2]$?

(a) 0

(b) 2

(c) -2

(d) -6

(x) Welke van de volgende uitspraken is waar m.b.t. de expressie $\mathit{takeWhile}\ \mathit{True}\ (\mathit{repeat}\ \mathit{True})$:

(a) resultaat is gelijk aan $\mathit{repeat}\ \mathit{True}$

(b) resultaat is gelijk aan $[\,]$

(c) evaluatie gaat in een loop, en er verschijnt geen resultaat

(d) is niet goed getypeerd

(xi) Which of the following is equivalent to $[x + y \mid x \leftarrow [1\,..\,10], \mathit{even}\ x, y \leftarrow [1\,..\,10]]$?

(a) $\mathit{map}\ (+) \circ \mathit{filter}\ (\mathit{even} \circ \mathit{fst})\ \$\ [(x, y) \mid x \leftarrow [1\,..\,10], y \leftarrow [1\,..\,10]]$

(b) $\mathit{concat} \circ \mathit{map}\ ((\mathit{flip}\ \mathit{map}\ [1\,..\,10]) \circ (+)) \circ \mathit{filter}\ \mathit{even}\ \$\ [1\,..\,10]$

(c) $\mathit{map}\ (\lambda x \rightarrow \mathit{map}\ (x+)\ [1\,..\,10]) \circ \mathit{concat} \circ \mathit{filter}\ \mathit{even}\ \$\ [1\,..\,10]$

(d) $\mathit{concat}\ (\mathit{zipwith}\ (+)\ [2, 4\,..\,10]\ [1\,..\,10])$

Merk op: $\mathit{flip}\ f\ x\ y = f\ y\ x$.

(xii) The function *intString* converts an *Int*-value to its *String* representation. We have the following variants:

$$\begin{array}{lll}
intString1 = reverse & \circ\ map\ (digitChar \circ (`rem`10)) & \circ\ takeWhile\ (\not\equiv 0) \circ iterate\ (/10) \\
intString2 = map\ (digitChar \circ (`rem`10)) & \circ\ reverse & \circ\ takeWhile\ (\not\equiv 0) \circ iterate\ (/10) \\
intString3 = map\ digitChar & \circ\ reverse & \circ\ map\ (`rem`10) \circ takeWhile\ (\not\equiv 0) \circ iterate\ (/10)
\end{array}$$

Which of the following statements is true?

(a) Only *intString1* is a solution.

(b) Only *intString2* is a solution.

(c) Only *intString3* is a solution.

(d) **All definitions are equivalent.**

(xiii) Which of the following definition is equivalent to the function *tails* in the lecture notes?

(a) $tails = reverse \circ inits \circ reverse$

(b) $tails = inits \circ reverse$

(c) $tails\ xs = xs : tails\ (tail\ xs)$
This does not work for [] as argument.

(d) $tails = reverse \circ map\ reverse \circ inits \circ reverse$

(xiv) Which expressions are equivalent, i.e., can replace each other in any context?

(a) $takeWhile\ p \circ dropWhile\ p$ and $dropWhile\ p$

(b) $takeWhile\ p \circ dropWhile\ p$ and id

(c) $takeWhile\ p \circ dropWhile\ q$ and $dropWhile\ q \circ takeWhile\ p$

(d) $takeWhile\ p \circ takeWhile\ q$ and $takeWhile\ (\lambda x \to p\ x \wedge q\ x)$

(e) $takeWhile\ p \circ dropWhile\ q$ and $takeWhile\ (\lambda x \to q\ x \wedge p\ x)$

(f) $takeWhile\ p \circ dropWhile\ p$ and $dropWhile\ p \circ takeWhile\ p$

(xv) Someone tries to write a function $revDigits :: Int \to Int$ that "reverses the digits in an *Int*"; so 123 is mapped onto 321 and 5612 is mapped to 2165, etc. Which is the correct solution?

(a) $revDigits\ i = foldl\ (\lambda ds\ x \to x : ds)$ "" $(show\ i)$

(b) $revDigits\ i = foldr\ (\lambda x\ ds \to ds +\!\!+ [x])$ "" $(show\ i)$

(c)
$$\begin{array}{l}
revDigits\ i = revDigits'\ i\ 0 \\
\quad \textbf{where}\ revDigits'\ 0\ r = r \\
\qquad\quad revDigits'\ i\ r = revDigits'\ (i\ `div`\ 10)\ (r * 10 + i\ `mod`\ 10)
\end{array}$$

(d)
$$\begin{array}{l}
revDigits\ i = revDigits'\ i\ 0 \\
\quad \textbf{where}\ revDigits'\ 0\ r = r \\
\qquad\quad revDigits'\ i\ r = revDigits'\ (i\ `mod`\ 10)\ (r * 10 + i\ `div`\ 10)
\end{array}$$

(xvi) What is the correct definition of the function *segs* that returns all segments from a list? For example $segs\ [2,3,4] = [[\,],[4],[3],[3,4],[2],[2,3],[2,3,4]]$.

(a)

$$\begin{aligned} segs\ [\,] &= [[\,]] \\ segs\ (x : xs) &= segs\ xs +\!\!+ map\ (x:)\ (inits\ xs) \end{aligned}$$

(b)

$$\begin{aligned} segs\ [\,] &= [\,] \\ segs\ (x : xs) &= segs\ xs +\!\!+ map\ (x:)\ (inits\ xs) \end{aligned}$$

(c)

$$\begin{aligned} segs\ [\,] &= [[\,]] \\ segs\ (x : xs) &= map\ (x:)\ segs\ xs +\!\!+ (inits\ xs) \end{aligned}$$

(d)

$$segs\ xs = zipWith\ (+\!\!+)\ (inits\ xs)\ (tails\ xs)$$

(xvii) Which of the following counts the number of subsets of a set of integers that sum up to a specific value (remember that all elements of a set differ from each other)?

(a)
$$\begin{aligned} &count\ [\,]\ 0 = 1 \\ &count\ [\,]\ \_ = 0 \\ &count\ (x : xs)\ v = count\ xs\ (v - x) \end{aligned}$$

(b)
$$\begin{aligned} &count\ [\,]\ 0 = 1 \\ &count\ xs\ v \mid v < 0 \quad = 0 \\ &\qquad\qquad \mid xs \equiv [\,] \quad = 0 \\ &\qquad\qquad \mid otherwise = count\ xs\ (v - head\ xs) + count\ (tail\ xs)\ v \end{aligned}$$

(c)
$$\begin{aligned} &count\ \_\ 0 = 1 \\ &count\ xs\ v = \textbf{if}\ v \leqslant 0\ \textbf{then}\ 0\ \textbf{else}\ sum\ [r \mid x \leftarrow xs, r \leftarrow count\ xs\ (v - x)] \end{aligned}$$

(d)
$$count\ xs\ v = sum \circ map\ (const\ 1) \circ filter\ (v \equiv)\ \$\ segs\ xs$$

(xviii) What is the type of $map\ (map\ map)$?

(a) $[[a \to b]] \to [[[a] \to [b]]]$

(b) $[a \to b] \to [[[a] \to [b]]]$

(c) $[[a \to b]] \to [[[a \to b]]]$

(d) $[[a \to b] \to [[a] \to [b]]]$

(xix) Which of the following following are expressions that can have type $[a \to a] \to [[a] \to [a]]$?

(a) $map\ map$

(b) $map\ (\lambda x \to [x])$

(c) $map\ (\lambda x \to [x] \to [x])$

(d) $(\lambda [x] \to map\ x)$

(e) None of the above

(xx) Which of the two expressions can replace each other in a program?

(a) $map\ f\ (concat\ zss)$ and $concat\ (map\ (map\ f)\ zss)$

(b) $map\ f \circ map\ g$ and $map\ (g \circ f)$

(c) $map\ (map\ f)$ and $(map\ map)\ f$

(d) $foldl\ (\oplus)\ e$ and $foldr\ (flip\ (\oplus))\ e$, where $flip\ f\ x\ y = f\ y\ x$

Types

(i) Wat is het type van $concat \circ concat$

(a) $[[[a]]] \to [a]$
(b) $[a] \to [[a]] \to [a]$
(c) $[[a]] \to [[a]] \to [[a]]$
(d) geen van (a) t/m (c)

(ii) Which is a type of *foldl map*?

(a) $[b] \to [b \to b] \to [b]$
(b) $[a] \to [a \to b] \to [b]$
(c) $[b] \to [b \to a] \to [b]$
(d) None of the above.

(iii) Wat is het type van $map \circ foldr$?

(a) $(a \to a \to a) \to [a] \to [[a] \to a]$
(b) $(a \to a \to a) \to [b] \to [b \to a]$
(c) $(b \to a \to a) \to [a] \to [[b] \to a]$
(d) $(b \to a \to a) \to [b] \to [[a] \to a]$

(iv) Which is the type of *map foldr*?

(a) $(a \to b \to b) \to [a \to b] \to [b]$
(b) $[a \to b \to b] \to [b \to [a] \to b]$
(c) $[a \to a \to b] \to [[a] \to b]$
(d) $[a \to b \to b] \to [[a] \to b \to b]$

(v) Which of the following is the type of $concat \circ concat$

(a) $[[a]] \to [[a]] \to [[a]]$
(b) $[[a]] \to [[a]] \to [a]$
(c) $[[[a]]] \to [a]$
(d) $[a] \to [[a]] \to [a]$

(vi) What is the type of *map map*?

(a) $[[a \to b]] \to [[a] \to [b]]$
(b) $[a \to b] \to [[a] \to [b]]$
(c) $[a \to b] \to [[[a]] \to [b]]$
(d) the expression *map map* is not well-typed

(vii) Compare the two expressions *map* (*map map*) and (*map map*) *map*. Which of the following holds?

(a) These expressions can replace each other everywhere, without changing the meaning of the program.
(b) There are situations where the first one terminates, but the second one does not.
(c) At least one of the expressions is not well-typed.
(d) None of the above.

(viii) Compare the two expressions $id\ (id\ id)$ and $(id\ id)\ id$. Which of the following holds?

(a) These expressions can replace each other everywhere, without changing the meaning of the program.

(b) There are situations where the first one terminates, but the second one does not.

(c) At least one of the expressions is not well-typed.

(d) None of the above.

(ix) Compare the two expressions $foldl\ (flip\ (:))\ [\,] \circ reverse$ and $reverse \circ foldl\ (flip\ (:))\ [\,]$, where:

$$flip\ f\ a\ b = f\ b\ a$$

Which of the following holds?

(a) These expressions can replace each other everywhere, without changing the meaning of the program.

(b) There are situations where the first one terminates, but the second one does not.

(c) At least one of the expressions is not well-typed.

(d) None of the above.

(x) Wat is het type van *map concat*

(a) $[[[a]]] \to [a]$

(b) $[[[a]]] \to [[a]]$

(c) $[[a]] \to [a]$

(d) geen van (a) t/m (c)

Data Types

(i) Which of the following is True?

(a) The data type *Maybe* represents the chance that the evaluation of an expression terminates.

(b) *Maybe* takes two type arguments: one indicating the type of the value to return when the expression terminates and one if it does not.

(c) Can be used to represent failure of some computation.

(d) Cannot occur inside a list.

(ii) A student defines a module with the definitions of the data type *Tree* and the variable v:

$$v = Node\ 3\ [Leaf\ 5, Node\ 4\ [Leaf\ 6, Node\ 5\ [\,]]] :: Tree$$

For which of the following definitions of *Tree* will the compiler not emit complaints when loading this module?

(a)

$$\begin{aligned} \textbf{data}\ Tree = {} & Node\ Int\ [Tree\ Int] \\ \mid {} & Leaf\ Int \end{aligned}$$

This is not a correct type since *Tree* should not take an argument.

(b)

$$\begin{aligned}\textbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ \mathit{Int}\ [\,\mathit{Tree}\ a\,]\\ \mid\ \mathit{Leaf}\ a\end{aligned}$$

Incorrect, since the type in the expression does not get *Int* as an argument.

(c)

$$\begin{aligned}\textbf{data}\ \mathit{Tree} = \mathit{Node}\ \mathit{Int}\ [\,\mathit{Tree}\ a, \mathit{Tree}\ a\,]\\ \mid\ \mathit{Leaf}\ \mathit{Int}\end{aligned}$$

Incorrectly formed type.

(d) **The compiler complains for all of the above.**

(iii) Which of the following holds for the search trees?

(a) Elements can only occur once

(b) You cannot search in an empty tree

(c) You cannot delete an element that is not in the tree.

(d) The time to find an element depends on the way the tree was constructed.

(iv) Which type is inferred for the variable t in the declaration $t = \mathit{map}\ \mathit{foldr}\ [\,]$? Let op!

(a) $[\,a\,]$

(b) $b \to [\,b\,] \to [\,b\,]$

(c) $[\,b \to [\,a\,] \to b\,]$

(d) geen van (a) t/m (c)

(v) Welk van de volgende types is equivalent aan een binaire zoekboom? Voor de volledigheid:

$$\begin{aligned}&\textbf{data}\ \mathit{GTree}\ f\ a = \mathit{GLeaf}\\ &\qquad\qquad\quad \mid \mathit{GNode}\ a\ (f\ (\mathit{GTree}\ f\ a))\\ &\textbf{data}\ \mathit{Paar}\ a\quad = \mathit{Paar}\ a\ a\end{aligned}$$

(a) **type** $\mathit{ZB}\ a = \mathit{GTree}\ \mathit{Paar}\ (a, a)$

(b) **type** $\mathit{ZB}\ a = \mathit{GTree}\ [\,]\ a$

(c) **type** $\mathit{ZB}\ a = \mathit{GTree}\ \mathit{Paar}\ a$

(d) **type** $\mathit{ZB}\ a = \mathit{GTree}\ (a, a)\ a$

Syntactic Sugar

(i) Someone who want to program list comprehension free, translates the right-hand side of the following definition into comprehension-free code. Which of the alternatives is the correct one?

$$\mathit{segs}\ \mathit{xs} = [\,] : [\,t \mid i \leftarrow \mathit{inits}\ \mathit{xs}, t \leftarrow \mathit{tails}\ i, \neg\ (\mathit{null}\ t)\,]$$

(a)
$$\begin{aligned}&[\,] : \mathit{concat}\ (\mathit{map}\ f\ (\mathit{inits}\ \mathit{xs}))\\ &\quad\ \textbf{where}\ f\ i = \mathit{concat}\ (\mathit{map}\ g\ (\mathit{tails}\ i))\\ &\qquad\qquad\ \textbf{where}\ g\ t = \textbf{if}\ \neg\ (\mathit{null}\ t)\ \textbf{then}\ [\,]\ \textbf{else}\ [\,t\,]\end{aligned}$$

$$
\begin{array}{ll}
 & [\,] : \mathit{concat}\ (\mathit{map}\ f\ (\mathit{inits}\ xs)) \\
\text{(b)} & \quad \textbf{where}\ f\ i = \mathit{concat}\ (\mathit{map}\ g\ (\mathit{tails}\ i)) \\
 & \qquad\qquad \textbf{where}\ g\ t = \textbf{if}\ \neg\ (\mathit{null}\ t)\ \textbf{then}\ [t]\ \textbf{else}\ [\,]
\end{array}
$$

$$
\begin{array}{ll}
 & [\,] : \mathit{concat}\ (\mathit{map}\ f \circ \mathit{map}\ g)\ (\mathit{inits}\ xs) \\
\text{(c)} & \quad \textbf{where}\ f\ i = \mathit{tails}\ i \\
 & \qquad\qquad g\ t = \textbf{if}\ \mathit{null}\ t\ \textbf{then}\ [\,]\ \textbf{else}\ [t]
\end{array}
$$

$$
\begin{array}{ll}
 & [\,] : \mathit{filter}\ (\neg \circ \mathit{null}) \circ \mathit{concat}\ (\mathit{map}\ f \circ \mathit{map}\ g)\ (\mathit{inits}\ xs) \\
\text{(d)} & \quad \textbf{where}\ f\ i = \mathit{tails}\ i \\
 & \qquad\qquad g\ t = [t]
\end{array}
$$

(ii) Someone who wants to program without list-comprehensions, translates the right-hand side of the definition

$$\mathit{flups}\ xs = [t \mid i \leftarrow \mathit{tails}\ xs, \neg\ (\mathit{null}\ i), t \leftarrow \mathit{inits}\ i]$$

into comprehension-free code. Which of the alternatives is a correct translation?

$$
\begin{array}{ll}
 & \mathit{filter}\ (\neg \circ \mathit{null}) \circ \mathit{concat}\ (\mathit{map}\ f \circ \mathit{map}\ g)\ (\mathit{tails}\ xs) \\
\text{(a)} & \textbf{where}\ f\ i = \mathit{inits}\ i \\
 & \qquad g\ t = [t]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{concat}\ (\mathit{map}\ f\ (\mathit{tails}\ xs)) \\
\text{(b)} & \textbf{where}\ f\ i = \mathit{concat}\ (\mathit{map}\ g\ (\mathit{inits}\ i)) \\
 & \qquad\qquad \textbf{where}\ g\ t = \textbf{if}\ \neg\ (\mathit{null}\ t)\ \textbf{then}\ [\,]\ \textbf{else}\ [t]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{concat}\ (\mathit{map}\ f\ (\mathit{tails}\ xs)) \\
\text{(c)} & \textbf{where}\ f\ i = \textbf{if}\ \neg\ (\mathit{null}\ i)\ \textbf{then}\ \mathit{concat}\ (\mathit{map}\ g\ (\mathit{inits}\ i))\ \textbf{else}\ [\,] \\
 & \qquad\qquad \textbf{where}\ g\ t = [t]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{concat}\ (\mathit{map}\ f \circ \mathit{map}\ g)\ (\mathit{tails}\ xs) \\
\text{(d)} & \textbf{where}\ f\ i = \mathit{inits}\ i \\
 & \qquad\qquad g\ t = \textbf{if}\ \mathit{null}\ t\ \textbf{then}\ [\,]\ \textbf{else}\ [t]
\end{array}
$$

Classes

(i) Which of the items is true for the following definition:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (\equiv), (\not\equiv) :: a \to a \to \mathit{Bool} \\
\quad x \not\equiv y = \neg\ (x \equiv y) \\
\quad x \equiv y = \neg\ (x \not\equiv y)
\end{array}
$$

(a) In a class definition it is not allowed to define functions in terms of each other.

(b) **This is exactly the definition of the class *Eq* from the Haskell report.**

(c) Because *Eq* is built-in into Haskell it can also be used to compare functions.

(d) The function definitions are not allowed here, since they belong to the **instance** declarations and not to the class declaration.

(ii) Using GHCi the Haskell expression: $2 + True$ results in the error message:

```
No instance for (Num Bool)
  arising from use of ‘+’ at <interactive>:1:1
Probable fix: add an instance declaration for (Num Bool)
```

If we follow the hint of the system we have amongst others to:

(a) Define a function *fromInteger* that maps *True* to some integer value.

(b) Define a function for (+) with type $Integer \rightarrow Bool \rightarrow Int$.

(c) **Define a function for** *fromInteger* **that has the type** $Integer \rightarrow Bool$**.**

(d) Both b and c

(iii) In the Haskell prelude the list constructor [ ] has been made an instance of the class *Monad*:

$$\begin{array}{l}\textbf{instance}\ Monad\ [\,]\ \textbf{where}\\ \quad ma \gg\!= a2mb = concat\ (map\ a2mb\ ma)\\ \quad return\ a \quad\ = [\,a\,]\end{array}$$

Which of the following equals $[f\ x\ y \mid x \leftarrow expr1, y \leftarrow expr2\,]$?

(a)

$$\begin{array}{l}\textbf{do}\ return\ (f\ x\ y)\\ \quad \textbf{where}\ x \leftarrow expr1\\ \qquad\quad\ \ y \leftarrow expr2\end{array}$$

(b)

$$\begin{array}{l}\textbf{do}\ x \leftarrow expr1\\ \quad\ \ y \leftarrow expr2\\ \quad\ \ f\ x\ y\end{array}$$

(c)

$$\begin{array}{l}\textbf{do}\ x \leftarrow expr1\\ \quad\ \ y \leftarrow expr2\\ \quad\ \ return\ (f\ x\ y)\end{array}$$

(d)

$$\begin{array}{l}\textbf{do}\ y \leftarrow expr2\\ \quad\ \ x \leftarrow expr1\\ \quad\ \ return\ (f\ x\ y)\end{array}$$

We ontsuikeren het derde alternatief door de **do** uit te drukken in zijn onderliggende betekenis:

$$\textbf{do } x \leftarrow expr1$$
$$\quad y \leftarrow expr2$$
$$\quad return\ (f\ x\ y)$$

$$expr1 \gg\!= (\lambda x \rightarrow \textbf{do } y \leftarrow expr2$$
$$\qquad\qquad\qquad\qquad return\ (f\ x\ y)$$
$$\qquad\quad)$$

$$expr1 \gg\!= (\lambda x \rightarrow expr2 \gg\!= \lambda y \rightarrow return\ (f\ x\ y))$$

$$expr1 \gg\!= (\lambda x \rightarrow expr2 \gg\!= \lambda y \rightarrow [f\ x\ y])$$

$$expr1 \gg\!= (\lambda x \rightarrow concat\ (map\ (\lambda y \rightarrow [f\ x\ y])\ expr2)$$

$$concat\ (map\ (\lambda x \rightarrow concat\ (map\ (\lambda y \rightarrow [f\ x\ y])\ expr2))\ expr1)$$

en dit is ook wat je krijgt als je de lijstdefinitie ontsuikert.

Alternatief (a) is helemaal geen correct Haskell, in (b) ontbreekt de *return*, en in (d) staan de x en de y verkeerd om.

(iv) Which of the following is true?

(a) **If we want to** *show* **a value of type** $[a]$ **we have always to make sure that** *show* **is also defined for values of type** a**.**

(b) We can call *show* on values of type $[a]$, without having defined *show* for a, as long as a itself is also a list type.

(c) If we define *show* for $[a]$, then *show* for values of type a is automatically constructed.

(d) We cannot define *show* for the polymorphic type $[a]$ since we cannot make this work for all possible types a at the same time.

(v) Someone, who is not importing the *Prelude*, defines the following instance of *Eq*:

$$\textbf{instance } Eq\ a \Rightarrow Eq\ (a, [a]) \textbf{ where}$$
$$(a, b) \equiv (c, d) = (a \equiv c)$$

Which of the following is true:

(a) This is illegal since the function $(\not\equiv)$ is not defined, which is also a member function of the class *Eq*.

(b) This is illegal since it is not defined how to compare values of type $[a]$.

(c) This is illegal since you do not refer to the variables b and d.

(d) None of the above.

(vi) Welke van de volgende uitspraken is waar?

(a) Een klasse in Haskell erft (via de context) van ten hoogste één andere klasse.

(b) Een type mag in een module maar voor een enkele klasse geïnstantieerd worden.

(c) Je mag in Haskell geen overlappende klasse-instanties hebben.

(d) Geen van (a) t/m (c).

IO

(i) Which of the following is true?

(a) The function *return* is idempotent (i.e. *return* (*return* a) can safely be replaced by *return* a).

(b) There exists expressions of type *IO* (*IO Int*).

(c) If you define an **instance** of the class *Eq* you have at least to specify the function (≡).

(d) The class *Enum* has a fixed number of instances.

(ii) Which of the following is true?

(a) Expressions of type *IO a* cannot occur inside other expressions.

(b) Expressions of type *IO a* can only occur as a subexpression of expressions of type *IO b* for a suitable *b*.

(c) The expression *return* (*return a*) can be replaced by *return a*.

(d) The expressions **do** *return a* is the same as *return a*.

(iii) Which of the following is true?

(a) Only a value with the name *main* can have a value of type *IO* ()

(b) Expressions of type *IO a* can only occur in a **do**-construct.

(c) The function *main* in the module *Main* must have type *IO* ().

(d) The function *main* cannot call itself recursively.

(iv) Wat is de correcte uitspraak over het volgende programma?

```
main = do c ← return 'c'
          c ← return (do c ← getChar
                         return (putChar c))
          c
          c
          return ()
```

(a) Leest twee karakters in (met echo) en doet verder niets.

(b) Geeft als uitvoer `'c'cc`

(c) Leest een karakter in en drukt dat eindeloos af

(d) Leest twee karakters in (met echo) en drukt die weer af.

Misc

(i) What is the result of the following parser application: *pMany* (*pSymb* `'a'`) `"aaa"`?

(a) `"aaa"`

(b) (`"aaa"`, `""`)

(c) [(`"aaa"`, `""`), (`"aa"`, `"a"`), (`"a"`, `"aa"`), (`""`, `"aaa"`)]

(d) None of the above

(ii) Which of the following holds:

(a) The function *sieve* in the lecture notes only works because all elements in the list of candidates are different.

(b) The function *sieve* in the lecture notes only works because the list of candidates is a strictly increasing list.

(c) The function *sieve* can easily be written using a *foldr*.

(d) None of the above.

(iii) Which of the following holds?

(a) Because Haskell is lazily evaluated the operator $\wedge$ is not associative.

(b) Because Haskell is lazily evaluated the operator $\wedge$ is associative.

(c) The associativity of $\wedge$ is not related to lazy evaluation.

(d) The operator $\wedge$ is neither associative nor non-associative.

(iv) Which of the following holds?

(a) Tupling increases the efficiency of your program by a constant factor.

(b) Use of *foldl′* is to be preferred over the use of *foldr* since it will consume less space.

(c) Use of strict functions instead of lazy ones makes your program run faster.

(d) None of the above.

(v) Welke van de volgende uitspraken is waar:

(a) De functie *foldl* gebruikt altijd minder ruimte dan *foldr*.

(b) De functie *foldr* kan al een stukje van een resultaat opleveren als het lijst argument nog niet helemaal bekend is.

(c) Zoeken in een zoekboom is altijd sneller dan zoeken in een lijst.

(d) Omdat lijstconcatenatie associatief is moet functiecompositie dat ook zijn.

(vi) Wat is het correcte type van de functie (:=) uit wxHaskell?

(a) $a \to Attr\ w\ a \to Prop\ w$

(b) $Attr\ w\ a \to a \to Prop\ w$

(c) $Attr\ w \to a \to Prop\ w\ a$

(d) $Prop\ w\ a \to a \to Attr\ w\ a$

(vii) Welke van de volgende uitspraken is waar m.b.t. de ontleedmethodes zoals behandeld op college:

(a) Een parser levert *altijd* een lijst op.

(b) De combinator <$> is eenvoudig uit te drukken m.b.v. *pSucceed* en <|>.

(c) De operator <*> is rechts-associatief

(d) De prioriteit van <*> is lager dan die van <|>.

(viii) Welke van de volgende uitspraken is waar m.b.t. de ontleedmethodes zoals behandeld op college:

(a) Een parser levert ook altijd een lege lijst op, zodat andere alternatieven ook geprobeerd worden.

(b) Het type van <*> is $Parser\ a \to Parser\ (a \to b) \to Parser\ b$

(c) Een correcte definitie voor *pChainl* is

$$
\begin{aligned}
&pChainl :: Parser\ (a \to a \to a) \to Parser\ a \to Parser\ a \\
&pChainl\ (\oplus)\ p = applyall\ <\$>\ p\ <*>\ pMany\ (flip\ <\$>\ (\oplus)\ <*>\ p) \\
&applyall\ a\ [\,] = a \\
&applyall\ a\ (f : fs) = applyall\ (f\ \ a)\ fs
\end{aligned}
$$

(d) De parser gebouwd met *pSucceed* slaagt alleen als de rest van de invoer ongelijk aan [] is.

(ix) Welke van de volgende uitspraken is waar:

(a) Elke operator die commutatief is, is ook associatief.

(b) Het definiëren van klasse-instanties voor geneste datatypes levert geen extra problemen op.

(c) Omdat functiecompositie associatief is vergt lijstconcatenatie constante tijd.

(d) Omdat lijstconcatenatie is ingebouwd vergt die altijd constante tijd.

(x) Which of the following holds with respect to the Prolog interpreter?

(a) The function *unify* always succeeds if neither of the arguments contains variables.

(b) The function *unify* always fails if both arguments are variables.

(c) The function *unify* always returns a substituation that is larger than its argument substitution if it succeeds.

(d) If a aunification with the head of a a rule succeeds the terms in the right hand side are added to the current goal set.

Bijlage D

Antwoorden

2.1 Haskell Brooks Curry Born: 12 Sept 1900 , Died: 1 Sept 1982. Curry's main work was in mathematical logic with particular interest in the theory of formal systems and processes. He formulated a logical calculus using inferential rules. His works include Combinatory Logic (1958) (with Robert Feys) and Foundations of Mathematical Logic (1963).

2.2

`=>`	gereserveerd symbool (staat in de lijst)
`3a`	niets (naam moet met letter beginnen)
`a3a`	naam
`::`	gereserveerd symbool (staat in delijst)
`:=`	operator [constructor]
`:e`	niets (opdracht is niet reserved)
`X_1`	naam [constructor]
`<=>`	operator
`a'a`	naam
`_X`	niets (naam moet met letter beginnen)
`***`	operator
`'a'`	niets (naam moet met letter beginnen)
`A`	naam [constructor]
`in`	gereserveerd woord (staat in lijst)
`:-<`	operator [constructor]

2.3 `4000.02`, `80.0`, `200000.0`

2.4 $x = 3$ is een definitie van een constante (mag bijvoorbeeld achter **where** staan). $x \equiv 3$ is een boolse expressie (met de waarde *True* of *False*).

2.5 De twee functies zijn:

$$\begin{aligned} &\mathit{aantalOpl}\ a\ b\ c \mid d > 0 = 2 \\ &\qquad\qquad\qquad\quad \mid d \equiv 0 = 1 \\ &\qquad\qquad\qquad\quad \mid d < 0 = 0 \\ &\qquad\qquad\qquad\quad \textbf{where}\ d = b * b - 4.0 * a * c \\ &\mathit{aantalOpl}'\ a\ b\ c = 1 + \mathit{signum}\ (b * b - 4.0 * a * c) \end{aligned}$$

2.6 Je kunt nu stukken programma inactief maken door er commentaar van te maken, ook als dat stuk programma zelf commentaar bevat.

2.7 Vraag het type aan de interpreter met bijvoorbeeld `:type tail`. Specificeer de types zelf door:

$$
\begin{array}{l}
tail :: [a] \to [a] \\
sqrt :: Float \to Float \\
pi \;\; :: Float \\
exp :: Float \to Float \\
(\uparrow) \;\; :: Num\ a \Rightarrow a \to Int \to a \\
(\not\equiv) :: Eq\ a \Rightarrow a \to a \to Bool \\
aantalOpl :: Float \to Float \to Float \to Int
\end{array}
$$

2.8 Respectievelijk *False* en *True*. In C is de eerste expressie false, wat in C gecodeerd wordt als 0. De tweede expressie betekent in C echter 'x wordt door 3 gedeeld', met de waarde 2 en het neveneffect dat x deze waarde krijgt. (In C wordt de ongelijkheid genoteerd door ! =).

2.9 'Syntax error' betekent 'vormfout'. Bij een syntax error staan de symbolen van een formule niet in de goede volgorde. Bij een type error is er wel sprake van een expressie, maar kloppen de types van de parameters niet met de functie of operator die gebruikt wordt.

2.10 $3 :: Int$ en $even :: Int \to Bool$, dus $even\ 3 :: Bool$. Je moet controleren of de parameter 'past' bij de functie; zo ja, dan is het resultaat van het type zoals gespecificeerd in de functie. Bij polymorfe functies gaat dat ongeveer hetzelfde: $head :: [a] \to a$ en $[1,2,3] :: [Int]$; het past dus, want $[Int]$ is een speciaal geval van $[a]$. Maar dan moet wel a gelijk zijn aan Int. Het type van $head\ [1,2,3]$ is dus niet a, maar Int.

2.11 De expressies hebben de volgende types:

$$
\begin{array}{lll}
until\ even & :: (Int \to Int) & \to Int \to Int \\
until\ or & :: ([Bool] \to [Bool]) & \to [Bool] \to [Bool] \\
foldr\ (\wedge)\ True & :: [Bool] & \to Bool \\
foldr\ (\wedge) & :: Bool & \to [Bool] \to Bool \\
foldr\ until & :: (a \to a) & \to [a \to Bool] \to a \to a \\
map\ sqrt & :: [Float\ ] & \to [Float] \\
map\ filter & :: [a \to Bool] & \to [[a] \to [a]] \\
map\ map & :: [a \to b] & \to [[a] \to [b]]
\end{array}
$$

2.12

a. bij de aanroep $fac\ (-3)$ wordt eerst $fac\ (-4)$ berekend, waarvoor $fac\ (-5)$ nodig is. De 'basiswaarde' 0 wordt op deze manier nooit bereikt, en er ontstaat een oneindig lang durende berekening.

b. De ordening volgens welke de parameter bij de recursieve aanroep 'eenvoudiger' is, moet naar onder begrensd zijn, en het basisgeval moet deze ondergrens behandelen.

2.13 In een lijst is ook de volgorde van de elementen van belang, en het aantal maal dat een element voorkomt.

2.14

a.
$$
\begin{array}{lll}
x \uparrow 0 & & = 1 \\
x \uparrow n & \mid even\ n & = square\ (x \uparrow (n\ `div`\ 2)) \\
 & \mid otherwise & = square\ (x \uparrow (n\ `div`\ 2)) * x \\
 & & \\
square\ x & & = x * x
\end{array}
$$

b. De nieuwe definitie bereikt het eindantwoord veel sneller:

```
2^10                    2^10
2*2^9                   sq  (2^5)
2*2*2^8                 sq  (sq  (2^2)*2)
2*2*2*2^7               sq  (sq  (sq  (2^1))*2)
```

```
2*2*2*2*2^6                 sq  (sq  (sq  (sq (2^0)*2))*2)
2*2*2*2*2*2^5               sq  (sq  (sq  (sq 1 *2))*2)
2*2*2*2*2*2*2^4             sq  (sq  (sq  (1*2))*2)
2*2*2*2*2*2*2*2^3           sq  (sq  (sq 2) *2)
2*2*2*2*2*2*2*2*2^2         sq  (sq  4 *2)
2*2*2*2*2*2*2*2*2*2^1       sq  (16*2)
2*2*2*2*2*2*2*2*2*2*2^0     sq  32
2*2*2*2*2*2*2*2*2*2*1       1024
2*2*2*2*2*2*2*2*2*2
2*2*2*2*2*2*2*2*4
2*2*2*2*2*2*2*8
2*2*2*2*2*2*16
2*2*2*2*2*32
2*2*2*2*64
2*2*2*128
2*2*256
2*512
1024
```

Voor hogere machten dan 10 is de tijdwinst nog groter.

2.15 Een lijst van lijsten, waarbij steeds drie keer hetzelfde element gekopieerd is:

$$[[0,0,0],[1,1,1],[3,3,3],[6,6,6],[10,10,10]]$$

3.1 $x + h$ is in zijn geheel parameter van f. Als de haakjes er niet staan, zou f alleen op x worden toegepast, omdat functie-toepassing de hoogste prioriteit heeft. Bij $f\ x$ hoeven er daarom geen haakjes te staan. Omdat / een hogere prioriteit heeft dan $-$, moeten om de linker parameter van / ook haakjes staan.

3.2

- Het eerste paar haakjes is overbodig, omdat functie-applicatie naar links associeert. Het tweede paar is echter wel nodig, omdat anders de eerste *plus* vier parameters zou krijgen.
- Beide haakjesparen zijn overbodig. Het eerste omdat haakjes om losse parameters niet nodig zijn (wel moet er in dit geval een spatie tussen *sqrt* en 3.0, omdat er anders de naam *sqrt3* zou staan). Het tweede haakjespaar is overbodig omdat functie-applicatie een hogere prioriteit heeft dan optelling (dat kon u niet weten, maar wel uitproberen).
- De haakjes om de getallen zijn overbodig. De haakjes om de operator niet, omdat de operator hier in prefix-notatie (als een soort functie) wordt gebruikt.
- Haakjes zijn overbodig, want vermenigvuldigen heet uit zichzelf al een hogere prioriteit dan optellen.
- Haakjes zijn nodig, want optellen heeft een lagere prioriteit dan vermenigvuldigen.
- Het tweede paar haakjes is overbodig, omdat $\rightarrow$ naar rechts associeert. Het eerste paar is wel nodig, omdat het hier een functie met een functie-parameter betreft, en niet een functie met drie parameters.

3.3 Door rechts-associatie is a^{b^c} hetzelfde als $(a^b)^c$: eerst b^c berekenen, en dan a tot die macht verheffen. Zou machtsverheffen links associëren, dan is a^{b^c} hetzelfde als $(a^b)^c$, maar dat kun je ook al zonder haakjes schrijven als $a^{b \cdot c}$. Associatie naar rechts bespaart dus haakjes.

3.4 De operator $\circ$ is associatief, want

$$\begin{aligned} & ((f \circ g) \circ h)\ x \\ = & (f \circ g)\ (h\ x) \\ = & f\ (g\ (h\ x)) \\ = & f\ ((g \circ h)\ x) \\ = & (f \circ (g \circ h))\ x \end{aligned}$$

3.5 Omdat er dan haakjes nodig zijn in expressies als $0 < x \ \&\& \ x < 10$.

3.6 *waardeIn0*, *plus*, *diff*.

3.7 Omdat *na* naar rechts associeert, mogen er om het rechter pijltje en zijn parameters straffeloos extra haakjes geplaatst worden. Je krijgt dan:

$$na :: (b \to c) \to ((a \to b) \to (a \to c))$$

Hieruit blijkt dat *na* een functie is met één parameter van type $b \to c$ en een resultaat van type $(a \to b) \to (a \to c)$. In de volgende definitie heeft de functie *na* inderdaad maar één parameter:

$$\begin{aligned} &na\ y = h \\ &\qquad \textbf{where}\ h\ f\ x = y\ (f\ x) \end{aligned}$$

3.8

$$g\ `endan`\ f = f \circ g$$

3.9

$$\begin{aligned} &f\ rente\ eind\ start \\ &\quad \mid eind \leqslant start = 0 \\ &\quad \mid otherwise \quad = 1 + f\ rente\ eind\ ((1 + rente) * start) \end{aligned}$$

3.10 Omdat voor de vereenvoudiging van *verbeter b* het functievoorschrift van de te inverteren functie nodig is. Als de te inverteren functie een parameter is, is dat functievoorschrift niet bekend; het enige wat er opzit is de functie in (veel) punten uit te proberen; dit gebeurt in *nulpunt*.

3.11

3.12 Voor lijsten hadden we:

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

wat we even omschrijven naar de $f\ a$ vorm:

$$map :: (a \to b) \to ([\,]\ a) \to ([\,]\ b)$$

Vervangen we nu $[\,]$ door $c \to$ dan krijgen we:

$$map :: (a \to b) \to (c \to a) \to (c \to b)$$

Omdat we verder niets over a, b of c weten waar we gebruik van kunnen maken, rest ons niet veel anders dan *map* voor de functor $c \to$ te definiëren als:

$$map\ a2b\ c2a = a2b \circ c2a$$

of

$$map\ a2b\ c2a = c2a\ `endan`\ a2b$$

blz. 54 **4.1** Voor de functie f met $f\ x = x^3 - a$ geldt $f'\ x = 3x^2$. De formule voor *verbeter b* in paragraaf 4 is in dit geval als volgt te vereenvoudigen:

$$\begin{aligned} & b - \frac{f\ b}{f'\ b} \\ = \quad & b - \frac{b^3 - a}{3b^2} \\ = \quad & b - \quad \frac{b^3}{3b^2} + \frac{a}{3b^2} \\ = \quad & \tfrac{1}{3}(2b + a/b^2) \end{aligned}$$

De functie *derdemachtswortel* luidt dan:

$$\begin{array}{ll} \mathit{derdemachtswortel}\ x = & \mathit{until}\ \mathit{goedGenoeg}\ \mathit{verbeter}\ 1.0 \\ \textbf{where} & \mathit{verbeter}\ y \quad\ = (2.0 * b + a\ /\ (b * b))\ /\ 3.0 \\ & \mathit{goedGenoeg}\ y = y * y * y \mathrel{\tilde{=}} x \end{array}$$

4.2 Door gebruik te maken van de lambda-notatie hoeft de functie f niet apart een naam te krijgen:

$$\mathit{inverse}\ g\ a = \mathit{nulpunt}\ (\lambda x \to g\ x - a)\ 0.0$$

5.3 We maken gebruik van recursie over de lijst van acties:

$$\begin{array}{l} \mathit{sequence}\ (a : \textbf{as}) = \textbf{do}\ a \\ \qquad\qquad\qquad\qquad\quad \mathit{sequence}\ \textbf{as} \\ \mathit{sequence}\ [\,] = \mathit{return}\ () \end{array}$$

5.6

```
getLine :: IO String
getLine = do c ← getChar
             if c ≡ '\n' then return ""
                         else do s ← getLine
                                 return (c : s)
```

6.11 Schrijf $[1, 2]$ als $1 : (2 : [\,])$ en pas driemaal de definitie van $+\!\!+$ toe:

$$\begin{array}{ll} & [1, 2] +\!\!+ [\,] \\ = & (1 : (2 : [\,])) +\!\!+ [\,] \\ = & 1 : ((2 : [\,]) +\!\!+ [\,]) \\ = & 1 : (2 : ([\,] +\!\!+ [\,])) \\ = & 1 : (2 : [\,]) \\ = & [1, 2] \end{array}$$

6.12 $\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$

6.13 *True* zijn de expressies 4, 5 en 6.

6.14 De functie *box* zet zijn parameter als singleton in een lijst als hij aan p voldoet, en levert anders de lege lijst op:

$$\begin{array}{ll} \mathit{box}\ x \mid p\ x & = [x] \\ \quad \mid \mathit{otherwise} & = [\,] \end{array}$$

6.15 $\mathit{repeat} = \mathit{iterate}\ \mathit{id}$ **where** $\mathit{id}\ x = x$

6.16 ...

6.17 ...

6.18 $\mathit{qEq}\ (x, y)\ (p, q) = x * q \equiv p * y$

6.19 Het resultaat van $(a+bi) * (c+di)$ kan berekend worden door de haakjes uit te werken: $ac + adi + bci + bdi^2$. Vanwege $i^2 = -1$ is dit gelijk aan $(ac-bd) + (ad+bc)\,i$. Voor de uitkomst van $1/(a+bi)$ lossen we x en y op uit $(a+bi) * (x+yi) = (1+0i)$. Dit geeft twee vergelijkingen met twee onbekenden: $ax - by = 1$ en $ay + bx = 0$. De tweede vergelijking geeft $y = (-b/a)x$. Invullen in de eerste vergelijking geeft $ax + \frac{b^2}{a}x = 1$ dus $x = \frac{a}{a^2+b^2}$ en $y = \frac{-b}{a^2+b^2}$.

```
type Complex = (Float, Float)

cPlus, cMin, cMaal, cDeel :: Complex → Complex → Complex
cPlus  (a, b) (c, d)      = (a + c, b + d)
cMin   (a, b) (c, d)      = (a − c, b − d)
cMaal  (a, b) (c, d)      = (a ∗ c − b ∗ d, a ∗ d + b ∗ c)
cDeel  (a, b) (c, d)      = cMaal (a, b) (c / k, −d / k)
  where
    k = c ∗ c + d ∗ d
```

6.20 De waarde van *stringInt* `['1','2','3']` is $((0 \star$ `'1'`$) \star$ `'2'`$) \star$ `'3'`, waarbij de operator $\star$ de operatie 'vermenigvuldig de linker waarde met 10 en tel de *digitValue* van de volgende character erbij op' is. Je begint aan de linkerkant, en kunt dus *foldl* gebruiken:

```
stringInt = foldl f 0
  where
    f n c = 10 ∗ n + digitValue c
```

In dit geval kan niet *foldr* gebruikt worden, omdat de operator niet associatief is. Dit is trouwens ook eens een voorbeeld waarbij het type van de twee parameters van de operator niet hetzelfde is. Dat kan, kijk maar naar het type van *foldl*.

6.21 Gebruik inductie, d.w.z. laat zien dat:

```
(reverse ∘ reverse) []       = id []
(reverse ∘ reverse) (x : xs) = id (x : xs)
```

6.22

```
transpose = foldr (zipWith (:)) (repeat [])
```

7.1 Niet alleen worden de eerste zeven en de laatste vier elementen omgewisseld. Door het effect in de recursieve aanroep worden ook die zeven elementen anders gerangschikt:

segs [1,2,3,4] = [[1] , [1,2] , [1,2,3] , [1,2,3,4] , [2] , [2,3] , [2,3,4] , [3] , [3,4] , [4] , []]

7.2 *segs* $= ([\,]:) \circ$ *filter* $(\not\equiv [\,]) \circ$ *concat* $\circ$ *map inits* $\circ$ *tails*

7.3 $(n{+}1)$, $1 + (n^2{+}n)/2$, 2^n, $n!$, $\binom{n}{k}$.

7.5 De functie tails maakt niet alleen gebruik van een recursieve aanroep op de straat van de lijst, maar ook van de straat zelf. Probeer nu eens of het je wel lukt door het product te berekenen van de *tails* en de *id*.

7.6 Hint: tupel de berekening van *segs* met die van *inits*.

7.7 Daar waar de functie *subs* kiest voor 'x in het resultaat opnemen' kiest *bins* voor 'een 1 in het resultaat opnemen'. Waar de functie *subs* kiest voor 'x niet in het resultaat opnemen' kiest *bins* voor 'een 0 in het resultaat opnemen':

```
bins 0       = [[]]
bins (n + 1) = map ('0':) binsn ++ map ('1':) binsn
  where
    binsn = bins n
```

7.8 Het kan zowel recursief (*gaps*) als met gebruikmaking van andere functies (*gaps′*):

$$
\begin{array}{ll}
gaps\ [\,] & = [\,] \\
gaps\ (x : xs) & = xs : map\ (x:)\ (gaps\ xs) \\
gaps'\ xs & = zipWith\ (+\!\!+)\ (init\ (inits\ xs))\ (tail\ (tails\ xs))
\end{array}
$$

7.9 Een matrix die een afbeelding beschrijft van een p-dimensionale ruimte naar een q-dimensionale ruimte heeft p kolommen en q rijen. Schrijven we $M(p,q)$ voor zo'n matrix, dan is 'type' van matrixvermenigvuldiging:

$$(\times) :: M(q,r) \to M(p,q) \to M(p,r)$$

De volgorde van de letters is hetzelfde als in het type van functiesamenstelling:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$

(Matrices kunnen in Haskell niet getypeerd worden met inachtneming van hun dimensie. Dit is omdat lijsten hetzelfde type hebben, ongeacht hun lengte. Onderscheid kan wel gemaakt worden als we tupels gebruiken in plaats van lijsten, maar dan moet er voor elke dimensie een apart stel functies geschreven worden.)

7.10
$$
\begin{array}{ll}
 & \det \binom{a\ b}{c\ d} \\
= & altsum\ [\ a * det(d)\ ,\ b * det(c)\ ] \\
= & altsum\ [\ a * d,\ b * c\ ] \\
= & a * d - b * c
\end{array}
$$

7.11 Het is het handigste om aan de hand van een voorbeeld te kijken welke functies na elkaar uitgevoerd moeten worden. Zie de figuur. De functie luidt dus:

$$
\begin{array}{l}
matInv\ (Mat\ m) = (\ \ Mat \\
\qquad\qquad\qquad\quad \circ\ zippWith\ (*)\ pmmp \\
\qquad\qquad\qquad\quad \circ\ mapp\ ((/.d) \circ det \circ Mat) \\
\qquad\qquad\qquad\quad \circ\ map\ (gaps \circ transpose) \\
\qquad\qquad\qquad\quad \circ\ gaps \\
\qquad\qquad\qquad\quad \circ\ transpose \\
\qquad\qquad\qquad\quad)\ \ m \\
\quad \textbf{where} \\
\quad\ d\ \ \ = det\ (Mat\ m) \\
\quad\ pmmp = iterate\ tail\ plusMinEen
\end{array}
$$

7.12 De functie *cp* combineert de elementen van twee lijsten op alle mogelijke manieren met elkaar. De functie *crossprod* combineert de elementen van een *lijst van* lijsten op alle mogelijke manieren met elkaar. We schrijven de functie met inductie. Als er slechts één lijst is, vormen de elementen daarvan in hun eentje de 'combinaties van één element'. Dus:

$$crossprod\ [xs] = map\ singleton\ xs$$

De functie *singleton* kan geschreven worden als $(:[\,])$. Voor het recursieve geval bekijken we bijvoorbeeld de lijst [[1,2] , [3,4] , [5,6]]. Een recursieve aanroep van *crossprod* op de staart levert de lijst [[3,5], [3,6], [4,5], [4,6]]. Deze elementen moeten op alle manieren gecombineerd worden met [1,2]. Dit geeft de definitie:

$$crossprod\ (xs : xss) = cpWith\ (:)\ xs\ (crossprod\ xss)$$

De definitie van het basisgeval kan ook anders geschreven worden:

$$crossprod\ [xs] = cpWith\ (:)\ xs\ [[\,]]$$

Wat het *crossprod* van een lege lijst is, is niet geheel duidelijk. Maar als we definiëren

$$crossprod\ [\,] = [[\,]]$$

$$\begin{matrix}147\\258\\369\end{matrix}$$

$$\downarrow \quad \textit{transpose}$$

$$\begin{matrix}123\\456\\789\end{matrix}$$

$$\downarrow \quad \textit{gaps}$$

$$\begin{bmatrix}\begin{matrix}???\\456\\789\end{matrix}, \begin{matrix}123\\???\\789\end{matrix}, \begin{matrix}123\\456\\???\end{matrix}\end{bmatrix}$$

$$\downarrow \quad \textit{map transpose}$$

$$\begin{bmatrix}\begin{matrix}?47\\?58\\?69\end{matrix}, \begin{matrix}1?7\\2?8\\3?9\end{matrix}, \begin{matrix}14?\\25?\\36?\end{matrix}\end{bmatrix}$$

$$\downarrow \quad \textit{map gaps}$$

$$\left[\begin{bmatrix}???\\?58\\?69\end{bmatrix}, \begin{matrix}?47\\???\\?69\end{matrix}, \begin{matrix}?47\\?58\\???\end{matrix}\right],$$

$$\begin{bmatrix}???\\2?8\\3?9\end{bmatrix}, \begin{matrix}1?7\\???\\3?9\end{matrix}, \begin{matrix}1?7\\2?8\\???\end{matrix}\Bigg],$$

$$\begin{bmatrix}???\\25?\\36?\end{bmatrix}, \begin{matrix}147\\???\\36?\end{matrix}, \begin{matrix}147\\25?\\???\end{matrix}\Bigg]\Bigg]$$

$$\downarrow \quad \textit{mapp}\ ((/d) \circ \textit{det} \circ \textit{Mat})$$

$$\begin{matrix}[\,[D_1/d,\ D_2/d,\ D_3/d]\,,\\ \ [D_4/d,\ D_5/d,\ D_6/d]\,,\\ \ [D_7/d,\ D_8/d,\ D_9/d]\,]\end{matrix}$$

$$\downarrow \quad \textit{zippWith}\ (*)\ \textit{pmmp}$$

$$\begin{matrix}[\,[+D_1/d,\ -D_2/d,\ +D_3/d]\,,\\ \ [-D_4/d,\ +D_5/d,\ -D_6/d]\,,\\ \ [+D_7/d,\ -D_8/d,\ +D_9/d]\,]\end{matrix}$$

dan is de definitie voor een lijst met één element gelijk aan die van een lijst met meer elementen:

$$crossprod\ [xs] = cpWith\ (:)\ xs\ (crossprod\ [\,])$$

Daarmee heeft de definitie de structuur gekregen van *foldr*, en kan dus ook geschreven worden als:

$$crossprod = foldr\ (cpWith\ (:))\ [[\,]]$$

De lengte van het *crossprod* van een lijst van lijstjes is het product van de lengtes van de lijstjes:

$$lengthCrossprod\ xs = product\ (map\ length\ xs)$$

Vandaar ook de naam *cross product*.

7.13 Als we ervoor zorgen dat er nooit staarten met nullen worden opgeslagen, dan is de graad 1 minder dan de lengte van de lijst (behalve voor het 0-polynoom):

$$\begin{aligned} pGraad\ [\,] &= 0 \\ pGraad\ xs &= length\ xs - 1 \end{aligned}$$

Er is dan wel een vereenvoudig-functie nodig, die staarten met nullen verwijdert:

$$\begin{aligned} &pEenvoud\ [\,] && = [\,] \\ &pEenvoud\ (x:xs) \mid x \equiv 0.0 \wedge staart \equiv [\,] && = [\,] \\ &\qquad\qquad\quad \mid otherwise && = x : staart \\ &\textbf{where} \\ &\quad staart = pEenvoud\ xs \end{aligned}$$

Voor het optellen van twee polynomen kunnen de overeenkomstige termen worden opgeteld, waarna het resultaat wordt vereenvoudigd:

$$pPlus\ xs\ ys = pEenvoud\ (zipWith\ (+)\ xs\ ys)$$

Definitie van *pMaal* geschiedt met inductie naar het eerste polynoom. Als dit het 0-polynoom is, is het resultaat ook het 0-polynoom is. Anders moet elke term van het tweede polynoom vermenigvuldigd worden met het eerste element (de constante term) van het eerste polynoom. De rest van het eerste polynoom wordt met het tweede polynoom vermenigvuldigd, waarna alle exponenten één verhoogd worden. Dat laatste kan door een 0 op kop van het resultaat te zetten. De twee resulterende polynomen worden opgeteld met *pPlus*. Het resultaat hoeft niet meer vereenvoudigd te worden, want dat doet *pPlus* al:

$$\begin{aligned} pMaal\ [\,]\ ys &= [\,] \\ pMaal\ (x:xs)\ ys &= pPlus\ (map\ (x*)\ ys) \\ &\qquad\qquad (0.0 : pMaal\ xs\ ys) \end{aligned}$$

8.1 Net als *zoekOp* gaat *zoekOpBoom* fout als de gezochte waarde niet in de verzameling zit, omdat er geen definitie voor *zoekOpBoom Blad z* is gegeven. Desgewenst kan een regel worden toegevoegd, waarin afhankelijk van de toepassing in dit geval een speciale waarde wordt opgeleverd. De te gebruiken speciale waarde zou eventueel zelfs als extra parameter aan *zoekOpBoom* kunnen worden meegegeven.

$$\begin{aligned} &zoekOpBoom && :: Ord\ a \Rightarrow Boom\ (a,b) \to a \to b \\ &zoekOpBoom\ (Tak\ (x,y)\ li\ re)\ z \mid z \equiv x &&= y \\ &\qquad\qquad\qquad\qquad\quad \mid z < x &&= zoekOpBoom\ li\ z \\ &\qquad\qquad\qquad\qquad\quad \mid z > x &&= zoekOpBoom\ re\ z \end{aligned}$$

8.2 De functie *map* transformeert een functie in een functie tussen *lijsten* van resp. zijn domein en bereik. Aangezien *map* zelf het type $(a \to b) \to ([a] \to [b])$ heeft, staan in het type van *map map* vierkante haken om domein en bereik:

$$map\ map :: [a \to b] \to [[a] \to [b]]$$

8.3 $until\ p\ f\ x = hd\ (dropWhile\ p\ (iterate\ f\ x))$

8.4 De operator $<$ op lijsten kan gedefinieerd worden met de volgende recursieve definitie:

$$\begin{array}{lll} xs & < [\,] & = False \\ [\,] & < ys & = True \\ (x : xs) & < (y : ys) & = x < y \vee (x \equiv y \wedge xs < ys) \end{array}$$

De volgorde waarin de eerste twee regels gegeven staan is van belang. Omdat de regels in volgorde worden geprobeerd, zal bij het vergelijken van twee lege lijsten de eerste regel gebruikt worden, wat inderdaad de bedoeling is.

8.5 Een definitie van *length* met behulp van *foldr* luidt:

$$\begin{array}{l} length = foldr\ op\ 0 \\ \quad \textbf{where} \\ \quad\quad x\ `op`\ n = n + 1 \end{array}$$

We beginnen met het voorlopige resultaat 0, en de operator *op* telt hierbij voor elk element in de lijst één op. De parameter n stelt het aantal elementen in de rest van de lijst voor, dat al geteld is door de recursieve aanroep in de definitie van *foldr*. De parameters van de operator *op* hebben niet hetzelfde type: de linker parameter is van willekeurig type (het type van de lijstelementen doet er niet toe), de rechter parameter en het resultaat echter zijn van type *Int*. Dus het type van de functie die aan *foldr* wordt meegegeven is $a \to Int \to Int$. Inderdaad accepteert de functie *foldr* functies met het type $a \to b \to b$.

8.6 De functie *qsort* kan als volgt gedefinieerd worden:

$$\begin{array}{lll} qsort & :: & Ord\ a \Rightarrow [a] \to [a] \\ qsort\ [\,] & = & [\,] \\ qsort\ (x : xs) & = & qsort\ (filter\ (\leqslant x)\ xs) \\ & & \quad +\!\!+ [x] \\ & & \quad +\!\!+ qsort\ (filter\ (>x)\ xs) \end{array}$$

Het verschil met *msort* is dat die functie twee willekeurige deel-lijsten neemt, die na het recursieve sorteren met enige moeite samengevoegd moeten worden, terwijl *qsort* de moeite investeert voor het selecteren van elementen in de deel-lijsten, zodat ze na het recursieve sorteren eenvoudigweg aan elkaar geplakt kunnen worden.

8.7 De functie *groepeer* kan als volgt gedefinieerd worden:

$$\begin{array}{lll} groepeer & :: Int \to [a] \to [[a]] \\ groepeer\ n & = takeWhile\ (\neg \circ null) \\ & \quad \circ map\ (take\ n) \\ & \quad \circ iterate\ (drop\ n) \end{array}$$

Als parameter van *takeWhile* gebruiken we $\neg \circ null$ in plaats van $\not\equiv [\,]$, omdat door het vermijden van de operator $\not\equiv$ het niet nodig is dat de lijstelementen vergelijkbaar zijn. De functie *null* is in de prelude gedefinieerd door

$$\begin{array}{l} null\ [\,] = True \\ null\ xs = False \end{array}$$

8.8 De functie *mapBoom* past een gegeven functie toe op alle elementen die in de bladeren staan opgeslagen:

```
mapBoom                 :: (a → b) → Boom2 a → Boom2 b
mapBoom f (Blad2 x)     = Blad2 (f x)
mapBoom f (Tak2 p q)    = Tak2 (mapBoom f p) (mapBoom f q)
```

De functie *foldboom* past een gegeven operator toe in elke *Tak*:

```
foldBoom                :: (a → a → a) → Boom2 a → a
foldBoom op (Blad2 x)   = x
foldBoom op (Tak2 p q)  = op (foldBoom op p) (foldBoom op q)
```

8.9 De inductieve versie:

```
diepte (Blad2 x)    = 0
diepte (Tak2 p q)   = max (diepte p) (diepte q) + 1
```

De versie met gebruik van map en fold:

```
diepte = foldBoom max ∘ mapBoom 0
```

8.10 We schrijven een hulpfunctie, met als extra parameter de diepte (en dus het aantal toe te voegen extra spaties) van de boom.

```
toonBoom                = toonBoom' 0
toonBoom' n Blad        = replicate n ' ' ++ "Blad\n"
toonBoom' n (Tak x p q) = toonBoom' (n + 5) p
                            ++ replicate n ' ' ++ show' x ++ "\n"
                            ++ toonBoom' (n + 5) q
```

8.11 Als de boom helemaal scheef is gegroeid, bevat hij het minimale aantal bladeren: $n + 1$. Een helemaal gevulde boom heeft 2^n bladeren.

8.12 We delen de lijst in twee ongeveer gelijke stukken en een los element, en passen de functie recursief toe op de twee helften. Dat levert twee ongeveer even diepe bomen. Die bomen voegen we samen toe één boom, waarbij ook het losse element wordt toegevoegd:

```
maakBoom [ ] = Blad
maakBoom xs = Tak x (maakBoom as) (maakBoom bs)
  where
    as      = take k xs
    (x : bs) = drop k xs
    k       = length xs / 2
```

8.13

```
df (Knoop l a r) = a : (df l ++ df r)
df Blad          = [ ]

bf t = concat levels
  where levels (Knoop l a r) = [a] : plak (levels l) (levels r)
        levels Blad          = [ ]
        plak (l : ls) (r : rs) = (l ++ r) : plak ls rs
        plak [ ]      rs       = rs
        plak ls       [ ]      = ls
```

We merken op dat de wijze van lijstopbouw niet de efficiëntste is, maar hier wel de duidelijkste. Later komen we hier op terug. De oplossing voor de infix opsomming vesrchilt nauwelijks van de depth-first oplossing.

8.14

$$
\begin{aligned}
&\mathit{paden} :: \mathit{Boom}\ a \to [[a]] \\
&\mathit{paden}\ \mathit{Blad} = [[\,]] \\
&\mathit{paden}\ (\mathit{Knoop}\ l\ a\ r) = \mathit{map}\ (a{:})\ (\mathit{paden}\ l +\!\!+ \mathit{paden}\ r)
\end{aligned}
$$

8.15 We berekenen het langste pad samen tegelijk met zijn lengte (deze techniek het het *tupelen* van berekeningen).

$$
\begin{aligned}
&\mathit{longestPath}\ t = \mathit{result} \\
&\quad \textbf{where}\ (\mathit{result}, \_) \qquad\quad = \mathit{lp}\ t \\
&\qquad\quad \mathit{lp}\ \mathit{Blad} \qquad\quad = ([\,], 0) \\
&\qquad\quad \mathit{lp}\ (\mathit{Knoop}\ l\ a\ r) = \textbf{let}\ (\mathit{leftp}, ll) \quad = \mathit{lp}\ l \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{rightp}, rl) = \mathit{lp}\ r \\
&\qquad\qquad\qquad\qquad\quad \textbf{in if}\ ll \geqslant rl\ \textbf{then}\ (a : \mathit{leftp}, ll + 1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (a : \mathit{rightp}, rl + 1)
\end{aligned}
$$

9.1 Om de rekenkundige operatoren op complexe getallen te kunnen gebruiken, moeten die tot instance van *Num* gemaakt worden:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Num}\ \mathit{Complex}\ \textbf{where} \\
&\quad (x, y) + (p, q) = (x + p, y + q) \\
&\quad (x, y) * (p, q)\ \ = (x * p - y * q, x * q + y * p)
\end{aligned}
$$

enzovoort.

9.2 In de instance-declaratie is + gedefinieerd als $(+) = \mathit{primPlusInt}$. Deze constante-definitie wordt de eerste keer dat hij gebruikt wordt 'uitgewerkt'. Een volgende keer dat + gebruikt wordt
blz. 181 wordt daarvoor direct *primPlusInt* aangeroepen. Vergelijk de functie *som* in paragraaf 14.

9.3 Verzamelingen worden als beschermd type gedefinieerd:

$$\textbf{data}\ \mathit{Set}\ a = \mathit{Set}\ [a]$$

(anders krijgen we straks 'overlapping instances' met de gewone gelijkheid op lijsten). De deelverzamelingstest controleert of alle elementen van de eerste verzameling een element van de tweede verzameling zijn:

$$\mathit{subset}\ (\mathit{Set}\ xs)\ (\mathit{Set}\ ys) = \mathit{and}\ (\mathit{map}\ (\in ys)\ xs)$$

Twee verzamelingen zijn gelijk als ze een deelverzameling van elkaar zijn. Dit idee wordt gebruikt in de instance-declaratie:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Set}\ a)\ \textbf{where} \\
&\quad x \equiv y = \mathit{subset}\ x\ y \wedge \mathit{subset}\ y\ x
\end{aligned}
$$

Ongelijkheid hoeft niet gedefinieerd te worden, want daarvoor is er een default-definitie in de klasse.

9.4 De klasse-declaratie is niet moeilijk:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Finite}\ a\ \textbf{where} \\
&\quad \mathit{members} :: [a]
\end{aligned}
$$

De elementen van *Bool* kunnen gewoon worden opgesomd. De elementen van *Char* kunnen makkelijker met *map* bepaald worden:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Finite}\ \mathit{Bool}\ \textbf{where} \\
&\quad \mathit{members} = [\mathit{False}, \mathit{True}]
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Finite}\ \mathit{Char}\ \textbf{where} \\
&\quad \mathit{members} = \mathit{map}\ \mathit{chr}\ [0\,..\,127]
\end{aligned}
$$

blz. 114 De lijst met alle mogelijke tweetupels kan gemaakt worden door de functie *cp* uit opgave 7.12
blz. 111 en paragraaf 7 te gebruiken, of met een lijst-comprehensie (één van de twee definities is inactief
blz. 28 gemaakt door er commentaar van te maken met behulp van -- ; zie paragraaf 2). Voor de lijst met ale mogelijke verzamelingen kan de functie *subs* uit paragraaf 7 gebruikt worden: blz. 95

```
instance (Finite a, Finite b) ⇒ Finite (a, b) where
  members = cp members members
    -- members = [ (x,y) — x¡-members, y¡-members ]

instance Finite a ⇒ Finite (Set a) where
  members = map Set (subs members)
```

De verzameling functies in de functieruimte met domein *xs* en bereik *ys* kan recursief worden gedefinieerd:

```
funspace [x] ys     = map const ys
funspace (x : xs) ys = [ λp → if p ≡ x then y else f p
                       | y ← ys
                       , f ← funspace xs ys
                       ]
```

Hiermee kan een functieruimte tussen eindige verzamelingen als eindige verzameling worden gedefinieerd:

```
instance (Eq a, Finite a, Finite b) ⇒ Finite (a → b) where
  members = funspace members members
```

Testen op gelijkheid van twee functies geschiedt nu door de twee functies op alle mogelijke (eindig vele) parameters toe te passen, en de resultaten te vergelijken. Het resultaat-type moet dus een instance van *Eq* zijn:

```
instance (Finite a, Eq b) ⇒ Eq (a → b) where
  f ≡ g = and [f x ≡ g x | x ← members]
```

10.1 In een ring distribueert vermenigvuldigen over aftrekken. Associativiteit van + wordt in onderstaand bewijs gebruikt zonder het te vermelden.

	$a * (b - c)$	$a * b - a * c$
	$=$ (def. $-$)	$=$ (def. $-$)
	$a * (b + neg\ c)$	$a * b + neg(a * c)$
	$=$ (0 neutraal voor $+$)	$=$ (0 neutraal voor $+$)
	$a * (b + neg\ c) + 0$	$a * (b + 0) + neg\ (a * c)$
	$=$ (*neg* inverse van $+$)	$=$ (*neg* inverse van $+$)
	$a * (b + neg\ c) + a * c + neg\ (a * c)$	$a * (b + neg\ c + c) + neg\ (a * c)$
	$=$ (distributiviteit)	
	$a * (b + neg\ c + c) + neg\ (a * c)$	

Deze stelling wordt gebruikt in het volgende bewijs:

	$a * 0$	0
	$=$ (*neg* inverse van $+$)	$=$ (*neg* inverse van $+$)
	$a * (0 + neg\ 0)$	$a * 0 + neg\ (a * 0)$ A
	$=$ (def. $-$)	$=$ (def. $-$)
	$a * (0 - 0)$	$a * 0 - a * 0$
	$=$ (vorige stelling)	
	$a * 0 - a * 0$	

10.2 De bedoelde foutmelding is '`overlapping instances for class "Monoid"`'.

10.3 Het kan maar op één manier, volgens onderstaande tabel:

$<+>$	E	A	B	C
E	E	A	B	C
A	A	E	C	B
B	B	C	E	A
C	C	B	A	E

10.4 De functie *fromInteger* hoort thuis in *Ring*. Er is immers een notie '**1**' voor nodig; *fromInteger* kan dan geïmplementeerd worden door herhaald **1** bij **0** op te tellen:

$$\begin{aligned} &\mathit{fromInteger} && :: \mathit{Ring}\ a \Rightarrow \mathit{Int} \to a \\ &\mathit{fromInteger}\ 0 && = \mathbf{0} \\ &\mathit{fromInteger}\ (n+1) && = \mathbf{1} <+> \mathit{fromInteger}\ n \end{aligned}$$

10.5 De uitdrukking *c1 + c2* moet vervangen worden door *c1* $<+>$ *c2*, en de uitdrukking $c \not\equiv 0.0$ door $c \not\equiv \mathbf{0}$. Het type wordt:

$$\mathit{pEenvoud} :: \mathit{Groep}\ a \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a$$

10.6 De functie termineert niet. De recursie stopt immers pas als *orde f* $<$ *orde g*. Als g het nulpolynoom is, dan is *orde g* gelijk aan -1. Er is geen enkel polynoom f waarvan de orde kleiner is dan -1.

10.7 Functies zijn een instance van *Monoid* met functie-compositie als operator en de identiteits-
blz. 54
functie als neutraal element. Float-functies vormen een groep, met de functie *inverse* uit paragraaf 4 als *neg*-functie. De functies moeten dan wel inverteerbaar en differentieerbaar zijn.

10.8 De voorwaarde $A@v = k \cdot v$ is equivalent aan $A@v = (k \cdot I)@v$, met I de identiteitsmatrix. Dit is weer gelijkwaardig aan $(A - k \cdot I)@v = O$, met O de nulvector. Als een lineaire afbeelding voor alle vectoren de nulvector oplevert, dan moet de determinant van de matrix 0 zijn (en omgekeerd). De matrix $k \cdot I$ is een matrix waarvan de elementen polynoompjes in k zijn (namelijk 0 of k). We beschouwen nu ook de matrix A als matrix van polynomen. Dan kan det $(A - k \cdot I)$ berekend worden: een polynoom $p[k]$. Een getal k is een eigenwaarde als het een oplossing is van $p[k] = 0$.

12.1 De data-declaratie wordt uitgebreid met vier nieuwe constructoren:

$$\begin{aligned} \mathbf{data}\ \mathit{Expr} = {} & ..... \\ & \mid \mathit{Sin}\ \mathit{Expr} \\ & \mid \mathit{Cos}\ \mathit{Expr} \\ & \mid \mathit{Exp}\ \mathit{Expr} \\ & \mid \mathit{Log}\ \mathit{Expr} \end{aligned}$$

De definitie van *afg* wordt daarom ook uitgebreid met vier nieuwe patronen voor deze vier nieuwe expressies:

$$\begin{aligned} &\mathit{afg}\ (\mathit{Sin}\ f)\ dx = \mathit{Cos}\ f :*: \mathit{afg}\ f\ dx \\ &\mathit{afg}\ (\mathit{Cos}\ f)\ dx = \mathit{Con}\ 0.0 :-: \mathit{Sin}\ f :*: \mathit{afg}\ f\ dx \\ &\mathit{afg}\ (\mathit{Exp}\ f)\ dx = \mathit{Exp}\ f :*: \mathit{afg}\ f\ dx \\ &\mathit{afg}\ (\mathit{Log}\ f)\ dx = \mathit{afg}\ f\ dx :/: f \end{aligned}$$

12.2 De functie *norep* moet recursief worden toegepast op eventuele deelstatements. De *Repeat* kan worden verwijderd door hem te vervangen door *While*. Van het predicaat moet daarbij de *Not* genomen worden, omdat 'while' doorgaat, maar 'repeat' juist stopt als het predicaat 'true' is. Bovendien wordt in de functie rekening gehouden met het feit dat het statement achter 'repeat' minstens eenmaal moet worden uitgevoerd.

$$
\begin{aligned}
&\mathit{norep} :: \mathit{Stat} \to \mathit{Stat}\\
&\mathit{norep}\ (\mathit{Assign}\ v\ e) = \quad \mathit{Assign}\ v\ e\\
&\mathit{norep}\ (\mathit{If}\ p\ s1\ s2) = \quad \mathit{If}\ p\ (\mathit{norep}\ s1)\ (\mathit{norep}\ s2)\\
&\mathit{norep}\ (\mathit{While}\ p\ s) = \quad \mathit{While}\ p\ (\mathit{norep}\ s)\\
&\mathit{norep}\ (\mathit{Repeat}\ s\ p) = \quad \mathit{Compound}\ [\mathit{norep}\ s\\
&\qquad\qquad\qquad\qquad , \mathit{While}\ (\mathit{Not}\ p)\ (\mathit{norep}\ s)]\\
&\mathit{norep}\ (\mathit{Compound}\ ss) = \mathit{Compound}\ (\mathit{map}\ \mathit{norep}\ ss)
\end{aligned}
$$

14.1 De versie uit de opgave is efficiënter, omdat $\mathit{map}\ (x{:})\ (\mathit{inits}\ xs)$ korter is dan $\mathit{segs}\ xs$, en $+\!\!+$ lineair is in zijn linker parameter, maar constant in zijn rechter.

14.2 De optimalisaties zijn als volgt:

	voor optimalisatie		na optimalisatie	
alg.	soort	complexiteit	soort	complexiteit
b.	kleiner/1/lineair	$\mathcal{O}(n^2)$	helft/2/lineair	$\mathcal{O}(n \log n)$
c.	kleiner/1/const	$\mathcal{O}(n)$	helft/1/const	$\mathcal{O}(\log n)$
d.	helft/2/const	$\mathcal{O}(n)$	helft/1/const	$\mathcal{O}(\log n)$
e.	kleiner/2/const	$\mathcal{O}(2^n)$	kleiner/1/const	$\mathcal{O}(n)$
f.	helft/2/lineair	$\mathcal{O}(n \log n)$	helft/2/const	$\mathcal{O}(n)$

14.3 De operator $+\!\!+$ is het meest efficiënt als hij rechts-associërend wordt uitgerekend (zie paragraaf 14.a). Dit gebeurt bij *foldr*. Het gebruik van *foldl* is minder efficiënt, omdat $+\!\!+$ dan links-associërend wordt uitgerekend. Is één van de lijsten in *xss* oneindig, dan is het resultaat van zowel de *foldr*- als de *foldl*-versie een oneindige lijst. De lijsten achter de oneindige lijst krijg je nooit te zien, maar de oneindige lijst zelf in ieder geval wel. Bij gebruik van foldl' komt er echter helemaal geen antwoord. De oneindige lijst moet immers eerst geconcateneerd worden met de lijsten die er achter komen (niet-lazy), voordat je het resultaat te zien krijgt. Daar kun je lang op wachten...

blz. 184

14.4

Wet *som na map-lineaire-functie*
Voor de som van een lineaire functie toegepast op een lijst getallen geldt de volgende wet:

$$\mathit{sum}\ (\mathit{map}\ ((k+) \circ (n*))\ xs) = k * \mathit{len}\ xs + n * \mathit{sum}\ xs$$

Bewijs met inductie naar *xs*:

`xs`	`sum (map ((k+).(n*)) xs)`	`k*len xs + n*sum xs`

`[]`	`sum (map ((k+).(n*)) [])`
	= (def. `map`)
	`sum []`
	= (def. `sum`)
	`0`
		`k*len [] + n*sum []`
		= (def. *len* en *sum*)
		`k*0 + n*0`
		= (def. `*`)
		`0+0`
		= (def. `+`)
		`0`

`x:xs`	`sum (map ((k+).(n*)) (x:xs))`
	= (def. `map`)
	`sum (k+n*x : map ((k+).(n*)) xs)`
	= (def. `sum`)
	`k+n*x + sum (map ((k+).(n*)) xs)`
		`k*len (x:xs) + n*sum (x:xs)`
		= (def. *len* en *sum*)
		`k*(1+len xs) + n*(x+sum xs)`
		= (distributie $*$)
		`k + k*len xs + n*x + n*sum xs`
		= (+ commutatief)
		`k+n*x + k*len xs + n*sum xs`

14.5 Bewijs eerst

$$foldr\ (\oplus)\ e\ (xs +\!\!+ ys) = (foldr\ (\oplus)\ e\ xs) \oplus (foldr\ (\oplus)\ e\ ys)$$

analoog aan 'sum na $+\!\!+$'. Daarna verloopt het bewijs analoog aan 'sum na concatenatie'.

14.6 Neem voor e de lege lijst, en definieer $g\ x\ ys = f\ x : ys$. Het bewijs verloopt met inductie naar xs:

	`map f`	`foldr g []`
`xs`	`map f xs`	`foldr g [] xs`

`[]`	`map f []`
	= (def. `map`)
	`[]`
		`foldr g [] []`
		= (def. `foldr`)
		`[]`

`x:xs`	`map f (x:xs)`
	= (def. `map`)
	`f x : map f xs`
		`foldr g [] (x:xs)`
		= (def. `foldr`)
		`g x (foldr g [] xs)`
		= (def. `g`)
		`f x : (foldr g [] xs)`

14.7 Het bewijs verloopt met inductie naar xs:

	`len . subs`	`(2^) . len`
`xs`	`len (subs xs)`	`2 ^ (len xs)`

`[]`	`len (subs [])`
	= (def. `subs`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
	= (def. `len`)
	`1 + 0`
		`2 ^ (len [])`
		= (def. `len`)
		`2 ^ 0`
		= (def. `(^)`)
		`1`
		= (eigenschap +)
		`1 + 0`

`x:xs`	`len (subs (x:xs))`
	= (def. `subs`)
	`len (map (x:)(subs xs) ++ subs xs)`
	= (lengte na $+\!\!+$)
	`len (map (x:)(subs xs)) + len (subs xs)`
	= (lengte na map)
	`len (subs xs) + len (subs xs)`
	= ($n + n = 2 * n$)
	`2 * len (subs xs)`
		`2 ^ (len (x:xs))`
		= (def. `len`)
		`2 ^ (1+len xs)`
		= (def. `(^)`)
		`2 * 2^(len xs)`

14.8

Wet *deelrijen na map*
Voor alle functies f op lijsten geldt:

$$subs \circ map\ f = map\ (map\ f) \circ subs$$

Het bewijs verloopt met inductie naar xs:

	`subs . map f`	`map (map f) . subs`
`xs`	`subs (map f xs)`	`map (map f) (subs xs)`

`[]`	`subs (map f [])`
	= (def. `map`)
	`subs []`
	= (def. `subs`)
	`[[]]`
		`map (map f) (subs [])`
		= (def. `subs`)
		`map (map f) [[]]`
		= (def. `map`)
		`[ map f [] ]`
		= (def. `map`)
		`[[]]`

`x:xs`	`subs (map f (x:xs))`
	= (def. `map`)
	`subs (f x:map f xs)`
	= (def. `subs`)
	`map (f x:)(subs (map f xs))`
	`++ subs (map f xs)`
		`map (map f)(subs (x:xs))`
		= (def. `subs`)
		`map (map f) (map (x:)(subs xs)`
		`++ subs xs)`
		= (map na $+\!\!+$)
		`map (map f) (map (x:) (subs xs))`
		`++ map (map f) (subs xs)`
		= (map na funtiecompostie)
		`map (map f.(x:)) (subs xs)`
		`++ map (map f) (subs xs)`
		= (map na op-kop)
		`map ((f x:).map f) (subs xs)`
		`++ map (map f) (subs xs)`
		= (map na funtiecompostie)
		`map (f x:) (map (map f)(subs xs))`
		`++ map (map f) (subs xs)`

14.9 Bewijs van wet 12, met inductie naar z:

`z`	`(x*y)^z`	`x^z * y^z`

`Nul`	`(x*y)^Nul`
	= (def. `^`)
	`Volg Nul`
	= (def. `+`)
	`Volg Nul + Nul`
		`x^Nul * y^Nul`
		= (def. `^`)
		`Volg Nul * Volg Nul`
		= (def. `*`)
		`Volg Nul + Nul*Volg Nul`
		= (def. `*`)
		`Volg Nul + Nul`

`Volg z`	`(x*y)^Volg z`
	= (def. `^`)
	`(x*y) * (x*y)^z`
		`x^Volg z * y^Volg z`
		= (def. `^`)
		`(x*x^z) * (y * y^z)`
		= ($*$ associatief)
		`((x*x^z)*y) * y^z`
		= ($*$ associatief)
		`(x*(x^z*y)) * y^z`
		= ($*$ commutatief)
		`(x*(y*x^z)) * y^z`
		= ($*$ associatief)
		`((x*y)*x^z) * y^z`
		= ($*$ associatief)
		`(x*y) * (x^z * y^z)`

Bewijs van wet 13, met inductie naar y:

y	(x^y)^z	x^(y*z)
Nul	(x^Nul)^z = (def. ^) (Volg Nul)^z = (wet 13a, volgt) Volg Nul	x^(Nul*z) = (def. *) x^Nul = (def. ^) Volg Nul
Volg y	(x^Volg y)^z = (def. ^) (x*(x^y))^z = (wet 12) x^z * (x^y)^z	x^(Volg y*z) = (def. *) x^(z+y*z) = (wet 11) x^z * x^(y*z)

Bewijs van wet 13a, gebruikt in het bewijs hierboven, met inductie naar x:

x	(Volg Nul)^x	Volg Nul
Nul	(Volg Nul)^Nul = (def. ^) Volg Nul	Volg Nul
Volg x	(Volg Nul)^(Volg x) = (def. ^) Volg Nul * (Volg Nul)^x = (def. *) (Volg Nul)^x + Nul*(Volg Nul)^x = (def. *) (Volg Nul)^x + Nul = (wet 1) (Volg Nul)^x	Volg Nul

Bibliografie

[1]

[2] 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.

[3] Paul McJones. History of fortran and fortran ii.

[4] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[5] Martin Odersky. The scala experiment: can we provide better language support for component systems? In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–167, New York, NY, USA, 2006. ACM.

[6] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.