Functional Programming

© Copyright 1992–1995 Jeroen Fokker and
Department of Computer Science, Utrecht University

This text may be reproduced for educational purposes under
the following restrictions:

- the text will not be edited or truncated;
- especially this message will be reproduced too;
- the duplicates will not be sold for profits;

You can reach the author at: Jeroen Fokker, Vakgroep Informatica,
Postbus 80089, 3508 TB Utrecht, e-mail jeroen@cs.ruu.nl.

Dutch edition:
1st print September 1992
2nd print February 1993
3rd reviewed print September 1993
4th reviewed print September 1994
5th reviewed print September 1995

Spanish edition, translated by Hielko Ophoff:
based on 4th reviewed print September 1994

English edition, translated by Dennis Gruijs and Arjan van IJzendoorn:
based on 5th reviewed print September 1995

Contents

1 Functional Programming *1*
1.1 Functional Languages *1*
1.1.1 Functions *1*
1.1.2 Languages *1*
1.2 The Gofer-interpreter *2*
1.2.1 Evaluating expressions *2*
1.2.2 Defining functions *4*
1.2.3 Internal commands *5*
1.3 Standard functions *5*
1.3.1 Primitive/predefined *5*
1.3.2 Names of functions and operators *6*
1.3.3 Functions on numbers *7*
1.3.4 Boolean functions *8*
1.3.5 Functions on lists *8*
1.3.6 Functions on functions *9*
1.4 Function definitions *9*
1.4.1 Definition by combination *9*
1.4.2 Definition by case distinction *10*
1.4.3 Definition by pattern matching *11*
1.4.4 Definition by recursion or induction *12*
1.4.5 Layout and comments *13*
1.5 Typing *14*
1.5.1 Kinds of errors *14*
1.5.2 Prototyping of expressions *15*
1.5.3 Polymorphism *16*
1.5.4 Functions with more parameters *17*
1.5.5 Overloading *17*
Exercises *18*

2 Numbers and functions *21*
2.1 Operators *21*
2.1.1 Operators as functions and vice versa *21*
2.1.2 Priorities *21*
2.1.3 Association *22*
2.1.4 Definition of operators *23*
2.2 Currying *23*
2.2.1 Partial parametrization *23*
2.2.2 Parentheses *24*
2.2.3 Operator sections *24*
2.3 Functions as a parameter *25*
2.3.1 Functions on lists *25*
2.3.2 Iteration *26*
2.3.3 Composition *27*
2.3.4 The lambda notation *28*
2.4 Numerical functions *28*
2.4.1 Calculation with whole integers *28*
2.4.2 Numerical differentiating *30*

2.4.3 Home-made square root *31*
2.4.4 Zero of a function *32*
2.4.5 Inverse of a function *33*
Exercises *34*

3 Data structures *37*
3.1 Lists *37*
3.1.1 Structure of a list *37*
3.1.2 Functions on lists *38*
3.1.3 Higher order functions on lists *42*
3.1.4 Sorting lists *43*
3.2 Special lists *45*
3.2.1 Strings *45*
3.2.2 Characters *45*
3.2.3 Functions on characters and strings *46*
3.2.4 Infinite lists *48*
3.2.5 Lazy evaluation *48*
3.2.6 Functions on infinite lists *49*
3.2.7 List comprehensions *51*
3.3 Tuples *52*
3.3.1 Use of tuples *52*
3.3.2 Type definitions *54*
3.3.3 Rational numbers *54*
3.3.4 Tuples and lists *55*
3.3.5 Tuples and Currying *56*
3.4 Trees *57*
3.4.1 Datatype definitions *57*
3.4.2 Search trees *59*
3.4.3 Special uses of data definitions *62*
Exercises *63*

4 List Algorithms *67*
4.1 Combinatorial Functions *67*
4.1.1 Segments and sublists *67*
4.1.2 Permutations and combinations *69*
4.1.3 The @-notation *70*
4.2 Matrix calculus *71*
4.2.1 Vectors and matrices *71*
4.2.2 Elementary operations *73*
4.2.3 Determinant and inverse *77*
4.3 Polynomials *79*
4.3.1 Representation *79*
4.3.2 Simplification *80*
4.3.3 Arithmetic operations *82*
Exercises *83*

5 Code transformation *85*
5.1 Efficiency *85*
5.1.1 Time *85*
5.1.2 Complexity analysis *85*
5.1.3 Improving efficiency *87*
5.1.4 Memory usage *90*
5.2 Laws *92*
5.2.1 Mathematical laws *92*
5.2.2 Gofer laws *94*
5.2.3 Proving laws *94*
5.2.4 Inductive proofs *96*
5.2.5 Improving efficiency *98*
5.2.6 properties of functions *101*

5.2.7 Polymorphism *105*
5.2.8 Proofs of mathematical laws *107*
Exercises *110*

A Exercises *113*
A.1 Complex numbers *113*
A.2 Texts *115*
A.3 Formula manipulation *117*
A.4 Predicate logic *119*
A.5 The class 'Set' *122*

B ISO/ASCII table *124*

C Gofer manual page *125*

D Gofer standard functions *127*

E Literature *132*

F Answers *1*

Chapter 1

Functional Programming

1.1 Functional Languages

1.1.1 Functions

In the 40s the first computers were built. The very first models were 'programmed' by means of huge connection boards. Soon the program was stored in computer memory, introducing the first programming languages.

Because in those days computer use was very expensive, it was obvious to have the programming language resemble the architecture of the computer as close as possible. A computer consists of a central processing unit and a memory. Therefore a program consisted of instructions to modify the memory, executed by the processing unit. With that the *imperative programming style* arose. Imperative programming language, like Pascal and C, are characterized by the existence of assignments, executed sequentially.

Naturally there were methods to solve problems before the invention of computers. With that the need for memory changed by instructions in a program was never really acknowledged. In math, for at least the last four hundred years, *functions* have played a much more central rôle. Functions express the connection between parameters (the 'input') and the result (the 'output') of certain processes.

In each computation the result depends in a certain way on the parameters. Therefore a function is a good way of specifying a computation. This is the basis of the *functional programming style*. A 'program' consists of the definition of one or more functions. With the 'execution' of a program the function is provided with parameters, and the result must be calculated. With this calculation there is still a certain degree of freedom. For instance, why would the programmer need to prescribe in what order independent subcalculations must be executed?

With the ongoing trend of cheaper computer time and more expensive programmers it gets more and more important to describe a calculation in a language closer to the 'human world' than to the world of a computer. Functional programming languages match the mathematical tradition, and are not influenced too strongly by the concrete architecture of the computer.

1.1.2 Languages

The theoretical basis of imperative programming was already founded in the 30s by Alan Turing (in England) and John von Neuman (in the USA). The theory of functions as a model for calculation comes also from the 20s and 30s. Some of the founders are M. Schönfinkel (in Germany and Russia), Haskell Curry (in England) and Alonzo Church (in the USA).

It lasted until the beginning of the 50s until someone got the idea to really use this theory as a basis for a programming language. The language Lisp of John McCarthy was the first functional programming language, and for years it remained the only one. Although Lisp is still being used, this is not the language which satisfies modern demands. Because of the increasing complexity of computer programs the need for better verification of the program by the computer arose.

With that the use of *prototyping* plays an important rôle. Therefore it is no surprise that in the 80s a huge number of prototyped functional programming languages came into existence: ML, Scheme (an adjustment to Lisp), Miranda and Clean are just a few examples.

Eventually every scientist developed its own language. To stop this rampant behavior a large

number of prominent scientists designed a new language, unifying all the advantages of the different languages. The first implementations of this language, called Haskell, were made in the early 90s. Because the languages is very ambitious, it is very hard to implement. The language Gofer is a slightly simplified Haskell-like language, used for theoretical and educative goals. Due to its wide availability this language became popular very quickly. Whether the language will eventually last cannot be answered; nevertheless it could not do any harm to study Gofer, since this language is very close to Haskell, which is broadly accepted.

The languages ML and Scheme are also popular; however these languages have made some concessions in the direction of imperative languages. They are therefore less fit as an example of a purely functional language. Miranda is a real functional language, nevertheless it is very hard to get so it would not be appropriate either.

In this reader the language Gofer is being used. In this language many concepts are implemented in a consequent way, making the language easy to learn. Who eventually knows Gofer will have little trouble to understand other functional languages.

1.2 The Gofer-interpreter

1.2.1 Evaluating expressions

In a functional programming language you can define functions. The functions are meant to be used in an expression, of which the value must be computed. To compute the value of an expression with a computer, a program is needed which understands the function definitions. Such a program is called an *interpreter*.

For the Gofer language used in this reader a Gofer-interpreter is available. This interpreter is initiated by typing the name of the program, `gofer`. If you do this, the following text will appear on screen:

```
%gofer
Gofer Version 2.30  Copyright (c) Mark P Jones 1991
Reading script file "/usr/staff/lib/gofer/prelude":
Parsing.........................................................
Dependency analysis.............................................
Type checking...................................................
Compiling.......................................................
```

In the file `prelude` (which is located in this example in `/usr/staff/lib/gofer`) are the definitions of standard functions. The first the Gofer-interpreter does, is analyze these standard functions. 'Prelude' literally means 'introductory part': this file is interpreted before any other functions are being defined. In this case there are no new functions; therefore the interpreter says by

```
Gofer session for:
/usr/staff/lib/gofer/prelude
```

that only the definitions in the prelude can be used. Finally the interpreter reports you can type `:?` for a brief explanation:

```
Type :? for help
?
```

The question mark means the interpreter is now ready to evaluate the value of an expression. Because mathematical functions are defined in the prelude, the interpreter can now be used as a calculator:

```
? 5+2*3
11
(5 reductions, 9 cells)
?
```

The interpreter calculates the value of the expression entered, where `*` denotes multiplication. After reporting the result (11) the interpreter reports the calculation took '5 reductions' (a measure for the amount of time needed) and '9 cells' (a measure for the amount of memory used). The question mark shows the interpreter is ready for the next expression.

The functions familiar from the calculator can also be used in an expression:

```
? sqrt(2.0)
1.41421
(2 reductions, 13 cells)
?
```

The function `sqrt` calculates the square root of a number. Because functions in a functional language are so frequently used, brackets may be left out in a function call (use in an expression). In complicated expressions this makes an expression much more readable. So the former call to `sqrt` can also be written as:

```
? sqrt 2.0
1.41421
(2 reductions, 13 cells)
```

In mathematical books it is conventional that 'writing next to each other' of expressions means multiplying those expressions. Therefore it is needed to put brackets when calling a function. However, in Gofer expressions a function call is much more common than multiplication. That is why 'writing next to each other' in Gofer is interpreted as a function call, and multiplication must be written down explicitly (with an `*`):

```
? sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1.0
(8 reductions, 27 cells)
```

Large amounts of numbers can be put into a *list* in Gofer. Lists are denoted with square brackets. There is a number of standard functions operating on lists:

```
? sum [1..10]
55
(91 reductions, 130 cells)
```

In this example `[1..10]` is the Gofer notation for the list of numbers from 1 to 10. The standard function `sum` can be applied to such a list to calculate the sum (55) of those numbers. Just as with `sqrt` and `sin` the (round) parentheses are redundant when calling the function `sum`.

A list is one of the ways to compose data, making it possible to apply functions to large amounts of data. Lists can also be the result of a function:

```
? sums [1..10]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
(111 reductions, 253 cells)
```

The standard function `sums` returns next to the sum of the numbers in the list also all the intermediate results. All these values are in a list, which in its whole is the result of the function `sums`. There are more standard functions manipulating lists. What they do can often be guessed from the name: `length` determines the length of a list, `reverse` reverses the order of a list, and `sort` sorts the elements of a list from small to large.

```
? length [1,5,9,3]
4
(18 reductions, 33 cells)
? reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
(99 reductions, 199 cells)
? sort [1,6,2,9,2,7]
[1, 2, 2, 6, 7, 9]
(39 reductions, 110 cells)
```

In an expression more functions can be combined. It is for example possible to first sort a list and then reverse it:

```
? reverse (sort [1,6,2,9,2,7])
[9, 7, 6, 2, 2, 1]
(52 reductions, 135 cells)
```

As conventional in mathematical literature, $g\ (f\ x)$ means that f should be applied to x and g should be applied to the result of that. The parentheses in this example are (even in Gofer!) necessary, to indicate that $(f\ x)$ is a parameter to g as a whole.

1.2.2 Defining functions

In a functional programming language it is possible to define new functions by yourself. The function can be used afterwards, along with the standard functions in the prelude, in expressions. Definitions of a function are always stored in a file. This file can be made with any word processor or editor you like.

It would be clumsy to leave the Gofer interpreter, start the editor to edit the function definition, quit the editor, and restart the Gofer interpreter for each minor change in a function definition. For this purpose it has been made possible to launch the editor *without* leaving the Gofer interpreter; when you leave the editor the interpreter will be immediately ready to process the new definition.

The editor is called by typing '`:edit`', followed by the name of a file, for example:

```
? :edit new
```

From the colon at the beginning of the line the interpreter knows that `edit` is not a function, but an internal command. The interpreter temporarily freezes to make space for the editor. The Gofer interpreter on a Unix computer uses `vi` (but it is possible to choose another editor[1]).

For instance, the definition of the factorial function can be put in the file '`new`'. The factorial of a number n (mostly written as $n!$) is the product of the numbers 1 to n, for example $4! = 1*2*3*4 = 24$. In Gofer the definition of the function `fac` could look like:

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

This definition uses the notation for 'list of numbers between two values' and the standard function `product`.

When the function has been entered the editor can be quit (in `vi` this can be done with `ZZ` or `:wq`). After that the `?` appears again, showing Gofer is activated again. Before the new function can be used, Gofer needs to know that the new file contains function definitions. This can be told by entering the internal command `:load`, so:

```
? :load new
Reading script file "new":
Parsing..........................................................
Dependency analysis..............................................
Type checking....................................................
Compiling........................................................
Gofer session for:
/usr/staff/lib/gofer/prelude
new
?
```

After analyzing the new file Gofer reports that next to the `prelude` the definition in the file `new` can be used:

```
? fac 6
720
(59 reductions, 87 cells)
```

It is possible to add definitions to a file when it is already loaded. Then it is sufficient to just type `:edit`; the name of the file needs not to be specified.

For example a function which can be added to a file is the function 'n choose k': the number of ways in which k objects can be chosen from a collection of n objects. According to statistics literature this number equals

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

This definition can, just as with `fac`, be almost literally been written down in Gofer:

```
choose n k  =  fac n / (fac k * fac (n-k))
```

The function definitions in a file may call the other definitions: for example `choose` uses the function `fac`. Of course you may also use the standard functions.

[1]The editor is set by the value of the *environment variable* `EDITOR`, which can be changed by the Unix command `setenv`.

After quitting the editor the changed file is automatically examined by Gofer; it is not necessary to repeat the `:load` command. Immediately it can be determined in how many ways a committee of three persons can be chosen from a group of ten people:

```
? choose 10 3
120
(189 reductions, 272 cells)
```

1.2.3 Internal commands

Next to `:edit` and `:load` there is a number of other instructions which are meant directly for the interpreter (so they should not be interpreted as expressions). All these instructions start with a colon.

The possible instructions are:

:? This is an instruction to list all the possible instructions. Handy to find out what an instruction was called ('You need not know everything, you only need to know how to find it').

:quit With this instruction you close a Gofer session.

:load *files(s)* After this instruction Gofer knows the functions defined in the specified file(s). With `:load` without filenames Gofer will forget everything except the prelude.

:also *files(s)* With this instruction you can add files at any moment, without forgetting the previously loaded files.

:edit *file* This is an instruction to create or change the specified file. If the filename is left out, the most recently edited file is opened by the editor, and automatically reloaded after quitting the editor.

:reload This is an abbreviation to reload the most recently loaded file (for example if you change it in another window)

:find *function-name* With this instruction the editor is opened, exactly at the location where the mentioned function is defined.

:type *expression* This instruction determines the type (see section 1.5) of the expression. page 14

:set ±*letter* With this instruction a number of options can be turned on or off. For example the number of reductions and cells used are usually reported after each calculation. With the instruction `:set -s` this is turned off. After entering `:set +s` you enabled this feature again.

The most important options are `s`, `t`, `g` and `i`:

- **s**tatistical information (number of reductions and cells) after each calculation;
- **t**ype every result of a calculation (see section 1.5.2); page 15
- **g***arbage collection* will be reported (see section 5.1.4) page 90
- **i**nteger constants are treated specially (see section **??**). page **??**

The other available options are with normal use of Gofer not of importance.

1.3 Standard functions

1.3.1 Primitive/predefined

Except for function definitions Gofer programs can also contain definitions of constants and operators. A *constant* is a function without parameters. This is a constant definition:

```
pi = 3.1415926
```

A *operator* is a function with two parameters which is written *between* the parameters instead of in front of them. In Gofer it is possible to define your own operators. The function `choose` from section 1.2.2 maybe could have been defined as an operator, for example as !^! : page 4

```
n !^! k  =  fac n / (fac k * fac (n-k))
```

In the prelude over two hundred standard functions and operators are defined. The main part of the prelude consists of conventional function definitions, like you can write yourself. The function `sum` for example is only in the prelude because it is so often used; if it would not have been in there you could have written your own definition. Just as with your own definitions, this definition can be inspected by the instruction `:find`. This is a useful way to find out what a standard function does. For example, the definition of `sum` is found to be

```
sum = foldl' (+) 0
```

Of course you will need to know what the standard function `foldl'` does, but you can find that out too…

Other functions, like the function `primPlusInt` which adds two whole numbers, cannot be defined; they are in a 'magical' way always there. These functions are called *primitive functions*; their definition is built-in in the interpreter. You can try to find them, but this will not make it any clearer:

```
primitive primPlusInt "primPlusInt"
```

The number of primitive functions in the prelude is kept as small as possible. Most of the standard functions are defined in plain Gofer. These functions are called *predefined* functions.

1.3.2 Names of functions and operators

In the function definition

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

`fac` is the name of the function defined, and `n` the name of its parameter.

Names of functions and parameters have to start with a lower case letter. After that more letters may follow (both upper as lower case), but also numbers, the prime (`'`) and the underscore (_). Lower case and upper case letters are considered to be different letters. A few examples of possible function names or parameter names are:

```
f    sum    x3    g'    to_the_power_of    longName
```

The underscore sign is mostly used to make long names easier to read. Another way to achieve that is to start each word in the identifier (except for the first) with a capital. This is common in many programming languages.

Numbers and primes in a name can be used to emphasize the dependencies of some functions or parameters. However, this is only meant for the human reader; as far as the interpreter is concerned, the name `x3` is as related to `x2` as to `qX'a_y`.

Names beginning with a capital are used for special functions and constants, the so-called *constructor functions*. The definitions of these functions are described in section 3.4.1. page 57

There are 16 names which cannot be used for functions or variables. These *reserved keywords* have a special meaning to the interpreter. The reserved words in Gofer are:

```
case      class    data     else
if        in       infix    infixl
infixr    instance let      of
primitive then     type     where
```

The meaning of the reserved words will be treated later in this reader.

Operators consist of one or more symbols. An operator can consist of one symbol (for example `+`), but also of two (`&&`) or more (`!^!`) symbols. The symbols you can build an operator with are:

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

Permitted operators include:

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

The operators on the first of these two lines are predefined in the prelude. The operators on the second line can still be defined. Operators starting with a colon (`:`) are meant for constructor functions.

There are eleven symbol combinations which may not be used as operators, because they have a special meaning in Gofer. It concerns the following combinations:

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

But there are still lots of combinations left to express your ascii-art capabilities...

1.3.3 Functions on numbers

There are two kinds of numbers available in Gofer:

- Integer numbers, like `17`, `0` and `-3`;
- Floating-point numbers, like `2.5`, `-7.81`, `0.0`, `1.2e3` and `0.5e-2`.

The character `e` in floating-point numbers means 'times ten to the power of'. For example `1.2e3` denotes the number $1.2 \cdot 10^3 = 1200.0$. The number `0.5e-2` is in fact $0.5 \cdot 10^{-2} = 0.005$.

The four mathematical operators addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) can be used on both whole and floating-point numbers:

```
? 5-12
-7
? 2.5*3.0
7.5
```

With division of whole numbers the fractional part is neglected:

```
? 19/4
4
```

If exact division is desired floating-point numbers must be used:

```
? 19.0/4.0
4.75
```

The two kinds of numbers cannot be combined unconditionally:

```
? 1.5+2
ERROR: Type error in application
*** expression     : 1.5 + 2
*** term           : 1.5
*** type           : Float
*** does not match : Int
```

The four operators are all 'primitives' (for both whole numbers as for floating-point numbers)

Besides that the prelude defines a number of standard functions on whole numbers. These functions are not 'built in', but just 'predefined' and could have been, if they were not in the prelude, defined manually. Some of these predefined functions are:

`abs`	the absolute value of a number
`signum`	-1 for negative numbers, 0 for zero, 1 for positive numbers
`gcd`	the greatest common divisor of two numbers
`^`	the 'power'-operator (involution)

On floating-pint numbers there are some functions defined, however, these *are* 'built in':

`sqrt`	the square root function
`sin`	the sine function
`log`	the natural logarithm
`exp`	the exponential function (e-to-the-power-of)

There are two primitive function converting whole numbers to floating-point numbers and the other way around:

`fromInteger`	converts a whole number into a floating point number
`round`	rounds a floating-point number to a whole number

Real numbers have to remain below a certain maximum. The size of that maximum depends on the system you use. Mostly this maximum is 2^{31} (over 2.1 thousand million). But on small systems this maximum could be 2^{15} (32768). If the result of a calculation would be larger than this maximum, an idiotic value should be expected:

```
? 3 * 100000000
300000000
? 3 * 1000000000
-1294967296
```

Also floating-point values are limited (depending on the system 10^{38} or 10^{308}), and have a smallest positive value (10^{-38} or 10^{-308}). Also the mathematical precision is limited to 8 or 15 significant digits:

```
? 123456789.0 - 123456780.0
8.0
```

It would be wise to always use whole numbers for discrete values. Floating-point values can be used for continuous values like distances and weights.

In this reader it is assumed that all values are smaller than the allowed maxima.

1.3.4 Boolean functions

The operator `<` determines if a number is smaller than another number. The result is the constant `True` (if it is true) or the constant `False` (if it is false):

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

The values `True` and `False` are the only elements of the set of *truth values* or *Boolean values* (named after the English mathematician George Boole). Functions (and operators) resulting such a value are called *Boolean functions.*

Next to `<` there is also an operator `>` (greater than), an operator `<=` (smaller or equal to), and an operator `>=` (greater or equal to). Furthermore, there is the operator `==` (equal to) and an operator `/=` (not equal to). Examples:

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

Results of Boolean functions can be combined with the operators `&&` ('and') and `||` ('or'). The operator `&&` only returns `True` if the results left *and* right are true:

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

The 'or' operator needs only one of the two statements to be true (both may be true also):

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

There is a function `not` swapping `True` and `False`. Furthermore there is a function `even` which checks if a whole number is even:

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

1.3.5 Functions on lists

In the prelude a number of functions and operators is defined on lists. Of these only *one* is a so-called primitive (the operator `:`), the rest is defined in plain Gofer.

Some functions on lists are already discussed: `length` determines the length of a list, `sum` calculates the sum of a list of whole numbers, and `sums` which calculates all partial sums of a list of whole numbers.

The operator `:` adds an extra element in front of a list. The operator `++` concatenates two lists. For instance:

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

The function `null` is a Boolean function on lists. This function checks if a list is empty (contains no elements). The function `and` operates on a list of which the elements are Booleans; `and` checks if all the elements in the list are `True`:

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

Some functions have two parameters. The function `take` receives a number and a list as parameter. If the number is n, the function will return the first n elements of the list:

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

1.3.6 Functions on functions

In the functions discussed so far, the parameters were numbers, Booleans or lists. However, the parameter of a function can be a function itself too! An example of that is the function `map`, which takes two parameters: a function and a list. The function `map` applies the parameter function to all the elements of the list. For example:

```
? 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]
```

Functions with functions as a parameter are frequently used in Gofer (why did you think it was called a functional language, eh?). In chapter 2 more of these functions will be discussed. page 21

1.4 Function definitions

1.4.1 Definition by combination

The easiest way to define functions is to combine a number of other functions, for example standard functions from the prelude:

```
fac n  =  product [1..n]
odd x  =  not (even x)
square x  =  x*x
sum_of_squares list  =  sum (map square list)
```

Functions can also get more than one parameter:

```
choose n k  =  fac n / (fac k * fac (n-k))
abcFormula 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)
                     ]
```

Functions with zero parameters are usually called ‘constants’:

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

So each function definition is of the form:

- the name of the function
- the name(s) of the optional parameter(s)
- a = sign
- an expression which may contain the parameters, standard functions and self defined functions.

A function returning a Boolean has on the right hand side of the = sign an expression with a Boolean value:

```
negative x  =  x < 0
positive x  =  x > 0
iszero   x  =  x == 0
```

Note the difference between the = and the ==. The sole equals sign (=) separates in function definitions the left hand side from the right hand side. A double equals sign (==) is an operator, just as < and >.

In the definition of the function `abcFormula` the expressions `sqrt(b*b-4.0*a*c)` and `(2.0*a)` are used twice. This not only results in a lot of typing, but the computation of such an expression is a waste of time: the identical sub-expressions are computed twice. To prevent this, it is possible in Gofer to name the sub-expressions. The improved definitions will be as follows:

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

With the statistics option of the interpreter enabled (by entering the command `:set +s`) the difference is very clear:

```
? abcFormula' 1.0 1.0 0.0
[0.0, -1.0]
(18 reductions, 66 cells)
? abcFormula 1.0 1.0 0.0
[0.0, -1.0]
(24 reductions, 84 cells)
```

The keyword `where` is not the name of a function: it is one of the 'reserved keywords' summarized in section 1.3.2. After '`where`' there is a number of definitions. In this case definitions of the constants `d` and `n`. These constants may be used in the expression after which `where` is typed. They cannot be used outside this function: they are *local definitions*. It might seem to be odd to call `d` and `n` 'constants', because the value with each function call of `abcFormula'` can be different. But during the calculation of *one* call of `abcFormula'`, with a given `a`, `b` and `c`, they are constant.

page 6

1.4.2 Definition by case distinction

Sometimes it might be needed to distinguish more cases in a function definition. An example is the absolute value function `abs`: a negative parameter results in something else than a positive parameter. In Gofer this is denoted as follows:

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

It is also possible to use more than two cases. This is done in the definition of the function `signum`:

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

The definitions of the different cases are 'guarded' by Boolean expressions, which is why they are called *guards*.

If a function is defined in this way, the guards will be tried one by one. At the first guard which is true, the expression on the right hand side of the = sign will be calculated. So the last guard could be replaced by `True` (or the constant `otherwise`).

So the description of the form of a function definition is more extensive than suggested in the previous section. A complete description of 'function definition' is:

- the name of the function;

- the names of zero or more parameters;
- a = sign and an expression, *or*: one or more 'guarded expressions';
- if desired the word `where` followed by local definitions.

Where each 'guarded expression' consists of a `|` sign, a boolean expression, a `=` sign, and an expression[2]. However, this description is not complete either...

1.4.3 Definition by pattern matching

The parameters of a function in a function definition, like `x` and `y` in

```
f x y  =  x * y
```

are called the *formal parameters* of the function. At calls this function is provided with the *actual parameters*. For example, in the function call

```
f 17 (1+g 6)
```

`17` is the actual parameter matched with `x`, and `(1+g 6)` the actual parameter matched with `y`. When calling a function these actual parameters are substituted in the places of the formal parameters in the definition. So the result of the call is `17*(1+g 6)`.

So actual parameters are *expressions*. Formal parameters have been *names*. In most programming languages formal parameters need always to be a name. However, in Gofer there are a few other possibilities: a formal parameter may also be a *pattern*.

An example of a function definition, where a pattern is used as a formal parameter is:

```
f [1,x,y]  =  x+y
```

This function only operates on lists with exactly three elements, of which the first element must be 1. Of such a list the second and third element will be added. So the function is not defined for shorter or longer lists, or on lists where the first element does not equal 1. (It is quite normal that functions are not defined for all actual parameters. For example, the function `sqrt` is not defined for negative numbers, and the operator `/` not for 0 as a second parameter.)

You can define functions with different patterns as a formal parameter:

```
sum' []      =  0
sum' [x]     =  x
sum' [x,y]   =  x+y
sum' [x,y,z] =  x+y+z
```

This function can be applied to lists with zero, one, two or three elements (in the next section this function will be defined on arbitrary long lists). In all cases the elements are added. When calling the function the interpreter 'matches' the actual to the formal parameter; e.g. the call `sum' [3,4]` matches the third line of the definition. The `3` will be matched to `x` in the definition and the `4` to the `y`.

The next constructions are permitted as a pattern:

- Numbers (e.g. `3`);
- The constants `True` and `False`;
- Names (e.g. `x`);
- Lists, of which the elements are patterns too (e.g. `[1,x,y]`);
- the operator `:` with patterns to the left and right (e.g. `a:b`);
- The operator `+` with on the left a pattern and on the right a natural number (e.g. `n+1`);
- The operator `*` with on the right a pattern and on the left a natural number (e.g. `2*x`).

With the help of patterns a number of important functions can be defined.

The operator `&&` from the prelude can be defined in this fashion:

```
False && False  =  False
False && True   =  False
True  && False  =  False
True  && True   =  True
```

With the operator `:` lists can be manipulated. The expression `x:y` means 'put element `x` in front of the list `y`'. But by putting the operator `:` in a pattern, the first element of a list will be taken of the front. With this two very useful standard functions can be written down:

[2]This description looks like a definition itself, with a local definition for 'guarded expression'!

```
head (x:y)  =  x
tail (x:y)  =  y
```

The function `head` returns the first element of a list (the 'head); the function `tail` returns everything but the first element (the 'tail'). Usage of these functions in an expression could be:

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

The functions `head` and `tail` can be applied to almost any list; they are only not defined on the empty list (a list without elements): an empty list has no first element, and certainly no 'tail'.

In patterns containing a `+` or a `*`, the 'matching' of parameters is executed in such a way the variables always represent a *natural* number. It is for example possible to define a function `even`, which returns `True` if a number is even:

```
even (2*n)   =  True
even (2*n+1) =  False
```

When calling `even 5` only the second pattern matches (where `n` is the natural number 2). The first pattern does not match, since `n` would have to be the non-natural number 2.5.

1.4.4 Definition by recursion or induction

In the definition of a function standard functions and self defined functions may be used. But also the function being defined may be used in its own definition! Such a definition is called a *recursive definition* (recursion means literally 'coming again': the name of the function comes back in its own definition). This function is a recursive function:

```
f x  =  f x
```

The name of the function being defined (`f`) is in the defining expression to the right of the `=`-sign. But this definition is not very useful; to compute the value of `f 3`, the value of `f 3` must first be computed, but in order to compute the value of `f 3`, the value of `f 3` must be computed and...

Recursive functions *are* useful under the following two conditions:

- the parameter of the recursive call is em simpler (for example: numerically smaller, or a shorter list) than the parameter of the function to be defined;
- there is a non-recursive definition for a *base case.*

A recursive definition for the factorial function is:

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

Here the base case is `n==0`; in this case the result can be determined directly (without recursion). The case `n>0` contains a recursive call, namely `fac (n-1)`. The parameter with this call (`n-1`) is, as demanded, smaller than `n`.

Another way to distinguish the two cases (the base case and the recursive case), is by the use of patterns:

```
fac 0      =  1
fac (n+1)  =  (n+1) * fac n
```

In this case the parameter of the recursive call (`n`) is also smaller than the parameter of the function being defined (`n+1`).

The use of patterns matches closely with the mathematical tradition of 'defining by induction'. The mathematical definition of involution can be used almost literally as a Gofer function:

```
x ^ 0      =  1
x ^ (n+1)  =  x * x^n
```

When a recursive definition distinguishes between different cases with patterns (instead of Boolean expressions) it is also called an *inductive definition.*

Functions on lists can also be recursive. Here a list is 'smaller' than another if it contains less elements (is shorter). The promised function `sum'` from the previous section which adds numbers in a list of arbitrary length, can be defined in different ways. A normal recursive function (where the distinction is made with Boolean expressions) is:

```
sum' list  |  list==[]    =  0
           |  otherwise   =  head list + sum' (tail list)
```

But here too, an inductive version is possible (where the distinction is being made by pattern matching):

```
sum' []      =  0
sum' (hd:tl) =  hd + sum' tl
```

In most cases a definition with patterns is clearer, since the different parts in the pattern are named immediately (like `hd` and `tl` in the function `sum'`). In the conventional recursive version of `sum'` the standard functions `head` and `tail` are needed to obtain to the different parts of the `list`. And again, in those functions patterns are used.

The standard function `length`, which counts the number of elements in a list, can also be defined inductively:

```
length []      =  0
length (hd:tl) =  1 + length tl
```

Here the value of the `head` element is not being used (only the fact there *is* a head element).

In patterns it is permitted to use the sign '`_`' in this case instead of a name:

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

1.4.5 Layout and comments

Almost everywhere in a program extra white space may be entered, to make the program easier to read. In previous examples there were added extra spaces, to align the `=` signs. Of course there may not be any spaces in the middle of a name of a function or in a number: `len gth` is something different than `length`, and `1 7` means something else than `17`.

Also newlines may be added to make the program more appealing. In the definition of `abcFormula` this was done, because otherwise the line would have gotten very long.

In contrast to other programming languages however, a newline is not entirely without meaning. For instance, look at the next two `where` constructions:

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

The location of the newline (between `x` and `y`, or between `y` and `b`) makes a huge difference.

In a list of definitions Gofer uses the following method to determine what belongs together:

- a line indented *exactly as far as* the previous line, is considered to be a new definition;
- if the line is indented *further*, then it belongs to the previous line;
- if the line is indented *less far*, then it does not belong to the current list of definitions.

The latter is of interest when a `where` clause within another `where` clause is being used, e.g.:

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

Here `w` is a local declaration of `f`, and not of `g`. This is because the definition of `w` is indented less far than that of `v`; it therefore does not belong to the `where` clause of `g` anymore. It is indented as far as the definition of `g`, and therefore belongs to the `where` clause of `f`. Would it have been indented even less far, then it would not even have belonged to that, and you would get an error.

It might look a bit complicated, but it always turns out right if you keep in mind one thing:

equivalent definitions should be indented equally far.

This implies also that all global function definitions should be indented equally far (for example zero positions).

Comments

Everywhere a space may be typed (so almost everywhere) comments may be added. Comments

are ignored by the interpreter, and are meant for eventual human readers of the program. There are two types of comments in Gofer:

- with the symbols `--` a comment is initiated ending at the end of the line;
- with the symbols `{-` a comment is initiated ending at the symbol `-}`.

Exception to the first rule is the case of `--` being part of an operator, for example `<-->`. A stand-alone `--` cannot be an operator: this combination was reserved in section 1.3.2. page 6

Comments with `{-` and `-}` can be *nested*, meaning they may contain pairs of these symbols. The comment is finished at the matching closing symbol. For example in

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

there is *no* function `f` being defined; everything is a comment.

1.5 Typing

1.5.1 Kinds of errors

To err is human, also when designing or typing a function. Fortunately the interpreter is able to warn for some mistakes. If a function definition is not qualified, you will receive an error as soon as it is being analyzed. The definition below contains an error:

```
isZero x  =  x=0
```

The second `=` should have been a `==` (`=` means 'is defined as', and `==` means 'is equal to'). When analyzing this function the interpreter reports:

```
Reading script file "new":
Parsing........................................................
ERROR "new" (line 18): Syntax error in input (unexpected `=')
```

The non-expressions in a program (*syntax errors*) are discovered by the interpreter during the first phase of the analysis: the parsing. Other syntax errors can be open parenthesis without close parenthesis, or using reserved words (like `where`) in places they are not allowed.

Except for syntax errors there are some other errors the interpreter can warn about. A possible error is the calling of a function which is nowhere defined. Often these errors are also the result of a typing error. When analyzing the definition

```
fac x  =  prodduct [1..x]
```

the interpreter reports:

```
Reading script file "new":
Parsing.......................................................
Dependency analysis...................................
ERROR "new" (line 19): Undefined variable "prodduct"
```

There errors are traced during the second phase: the dependency analysis.

The next obstacle for a program is the type checking. Functions which are meant to operate on numbers are not allowed to operate on Boolean values, and not on lists. Functions on lists cannot be applied to numbers, and so forth.

For instance, if a function definition contains the expression `1+True` the interpreter reports:

```
Reading script file "new":
Parsing........................................................
Dependency analysis............................................
Type checking..............................................
ERROR "new" (line 22): Type error in application
*** expression     : 1 + True
*** term           : 1
*** type           : Int
*** does not match : Bool
```

The term `1` has the *type* `Int` (An abbreviation of *integer*, or whole number). Such an integer value cannot be added to `True`, which is of the type `Bool`, (an abbreviation of 'Boolean value').

Other type errors occur when applying the function `length` to something which is not a list, like in `length 3`:

```
ERROR: Type error in application
*** expression     : length 3
*** term           : 3
*** type           : Int
*** does not match : [a]
```

Only if there are no type errors in the program, the fourth phase of analysis (compilation) will be executed. Only then the function can be used.

All errors are reported at the moment the function is being analyzed. The purpose of this is to not encounter unpleasant surprises afterwards. A function which survives the analysis, is guaranteed to be free of type errors.

Some other languages check the typing at the moment the function is being called. In these languages you never know if somewhere in a dark place of the program a type error resides.

The fact that a function survives the analysis does not imply the function is correct. If the function `sum'` would contain a minus sign instead of a plus sign, the interpreter will not complain because it is unable to understand you meant `sum'` to add. These kind of errors, 'logical mistakes', are the hardest to find, because the interpreter does not warn you.

1.5.2 Prototyping of expressions

The type of an expression can be determined by the interpreter command `:type`. Behind `:type` the expression is specified which should be typed. For example:

```
? :type 3+4
3 + 4 :: Int
```

The symbol `::` can be read as 'has type'. The expression will not be executed by `:type` commands; only the type is determined.

There are four basic types:

- `Int`: the type of the whole numbers (*integer numbers* or *integers*);
- `Float`: the type of the floating-point numbers;
- `Bool`: the type of Booleans `True` and `False`;
- `Char`: the type of letters, digits and symbols on the keyboard (*characters*), which will be discussed in section 3.2.2. page 45

Keep in mind that these types are written with a Capital.

Lists can have different types. There are lists of integers, lists of bools, and even lists of lists of integers. All these lists have different types:

```
? :type [1,2,3]
[1,2,3] :: [Int]
? :type [True,False]
[True,False] :: [Bool]
? :type [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

The type of a list is shown by putting the type of the elements of the list between brackets: `[Int]` is the type of a list of whole numbers. All elements of a list have to be of the same type. If not, an error will occur:

```
? [1,True]
ERROR: Type error in list
*** expression     : [1,True]
*** term           : True
*** type           : Bool
*** does not match : Int
```

Functions have types too. The type of a function is determined by the type of the parameter and the type of the result. The type of the function `sum` is:

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

The function `sum` operates on lists of integers and returns a single integer. The symbol `->` in the type of the function should be considered an arrow (→). In handwriting this can be written as such.

Other examples of types of functions are:

```
sqrt :: Float -> Float
even :: Int   -> Bool
sums :: [Int] -> [Int]
```

You can pronounce such a string as '`even` has the type int to bool' or '`even` is a function from int to bool'.

Because functions (just as numbers, Booleans and lists) are typed, it is possible to incorporate functions in a list. All functions in a list must have exactly the same type, because elements of a list must be of the same type. An example of a list of functions is:

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

All three functions `sin`, `cos` and `tan` are functions 'from float to float'; they can be put in a list, which will therefore have the type 'list of functions from float to float'.

The interpreter can determine the type of an expression itself. This also happens when type checking a program. However, it is permitted to write down the type of a function in a program. This is called 'prototyping'. A function definition will then look like:

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

Although such a *type-declaration* is redundant, it has two advantages:
- it will be checked if the function really has the type you declared;
- the declaration makes it much easier for a human reader to understand.

The type declaration need not be specified directly before the definition. For instance, you could start a program with the type declarations of all functions defined within. The declarations will then serve as some kind of index.

1.5.3 Polymorphism

To some functions on lists it does not matter what the type of the elements of the list is. The standard function `length` for example, can determine the length of a list of integers, but also of a list Booleans, and –why not– a list of functions. The type of the function `length` is declared as follows:

```
length :: [a] -> Int
```

This type shows the function has a list as a parameter, but the type of the elements of the list does not matter. The type of these elements is denoted by a *type variable*, in this example `a`. Type variables are, in contrast to plain types like `Int` and `Bool`, written with a lower case letter.

The function `head`, returning the first element of a list, has the following type:

```
head :: [a] -> a
```

This function operates also on lists where the type of the elements is not important. However, the result of the function `head` has the same type as the elements of the list (because it is the first element). That is why the same type variable is used for the type of the elements of the list.

A type containing type variables is called a *polymorphic type* (literally: 'many shaped type'). Functions with a polymorphic type are called polymorphic functions. The phenomenon itself is called *polymorphism*.

Polymorphic functions, like `length` and `head`, have in common that they only use the *structure* of the list. A non-polymorphic function, like `sum`, also uses the characteristics of the *elements* of the list, like 'summability'.

Polymorphic functions are usually generally usable; it is for example very common to calculate the length of a list. That is why a lot of those standard functions defined in the prelude are polymorphic functions.

Not only functions on lists can be polymorphic. The simplest polymorphic function is the identity function (the function which returns its parameter unchanged):

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

The function `id` can operate on elements of arbitrary type (and the result is of the same type). So it can be applied to an integer, for instance `id 3`, but also to a Boolean, for instance `id True`. It can also operate on lists of Booleans, for instance `id [True,False]` or on lists of lists of integers: `id [[1,2,3],[4,5]]`. The function can even be applied to functions of float to float, for instance `id sqrt`, or to functions of lists of integers to integers: `id sum`. As can be seen from the type, the function can be applied to parameters of arbitrary type. So the parameter may also be of the type `a->a`, which makes it possible to apply the function `id` to itself: `id id`.

1.5.4 Functions with more parameters

Even functions with more than one parameter have a type. In the type an arrow is writen between the parameters, and between the last parameter and the result. The function `choose` from section 1.2.2 has two integer parameters and an integer result. Therefore the type is: page 4

```
choose  ::  Int -> Int -> Int
```

The function `abcFormula` from section 1.4.1 has three floating-point numbers as parameters and a list of floating-point numbers as a result. Thus the type is: page 9

```
abcFormula  ::  Float -> Float -> Float -> [Float]
```

In section 1.3.6 the function `map` was already discussed. The function is applied to all elements of the list, so that the result is again a list. The type of `map` is as follows: page 9

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

The first parameter of `map` is a function between two arbitrary types (`a` en `b`), which are not even needed to be the same. The second parameter of `map` is a list, of which the elements have to be of the same type (`a`) as the parameter of the function parameter (because the function parameter has to be applied to the individual elements of the list). The result of `map` is a list, of which the elements have to be of the same type (`b`) as the result of the function parameter.

In the type declaration of `map` an extra pair of parentheses is needed around the type of the first parameter (`a->b`). Otherwise it would express that `map` has *three* parameters: an `a`, a `b`, a `[a]` and a `[b]` as a result. This is of course not what is meant by the author: `map` has two parameters: a `(a->b)` and a `[a]`.

Operators are typed too. This is because operators are ordinary functions with two parameters, they are just written in a different way (between their parameters instead of in front). This does not matter for the type: For instance,

```
(&&)  ::  Bool -> Bool -> Bool
```

1.5.5 Overloading

The operator `+` can be used on two whole numbers (`Int`) or on two floating-point numbers (`Float`). The result is again of the same type. The type of `+` can be both `Int->Int->Int` and `Float->Float->Float`. Still `+` is not really a polymorphic operator. If the type would be `a->a->a`, the operator would have to work on for instance `Bool` parameters too, which is impossible. A function which is 'restricted polymorphically', is called an *overloaded* function.

To still be able to type an overloaded function or operator, the types are divided into *classes*. A class is a set of types with some characteristics in common. In the prelude a few classes are already defined:

- `Num` is the class of which the elements can be added, subtracted, multiplied and divided (numerical types);
- `Ord` is the class of which the elements can be ordered (orderable types)
- `Eq` is the class of which the elements can be compared to each other (equality types).

The operator `+` now has the following type:

```
(+)  ::  Num a => a->a->a
```

This should be read like: '`+` has the type `a->a->a` provided that `a` is a type of class `Num`'.

Note the use of the arrow with the double stick (`=>` or if desired $\Rightarrow$). This has a very distinct meaning than an arrow with a single stick. Such a double arrow can occur only once in a type.

Other examples of overloaded operators are:

```
(<)  ::  Ord a  =>  a -> a -> Bool
(==) ::  Eq  a  =>  a -> a -> Bool
```

Self defined functions can also be overloaded. For example the function

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

has the type

```
square  ::  Num a  =>  a -> a
```

due to the operator `*` used in the definition, which is overloaded.

The usage of classes and the definitions of them is extensively discussed in chapter ??. Classes were only mentioned here to type overloaded operators. page ??

Exercises

1.1 Look up the words 'gofer' and 'gopher' in a dictionary.

1.2 If the following things are in a program, are they:

- something with some fixed meaning (reserved word or symbol);
- name of a function or parameter;
- an operator;
- none of the above

If it is a function or operator, is it a constructor function or constructor operator?

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

1.3 Compute:

```
4e3 + 2e-2
4e3 * 2e-2
4e3 / 2e-2
```

1.4 What is the difference in meaning between `x=3` and `x==3` ?

1.5 Write a function `numberSol` which, given a, b and c, calculates the number of solutions of the equation $ax^2 + bx + c$, in two versions:

- with case distinction
- by combining standard functions

1.6 What is the advantage of 'nested' comments (see section 1.4.5)? page 13

1.7 What is the type of the following functions: `tail`, `sqrt`, `pi`, `exp`, `(^)`, `(/=)` and `numberSol`? How can you ask the interpreter to determine that type, and how can you specify those types yourself in a program?

1.8 Let `x` have the value 5. What is the value of the expressions `x==3` and `x/=3` ? (For those familiar with the programming C language: What are the values of these expressions in the C language)

1.9 What does '*syntax error*' mean? What is the difference between a *syntax error* and a *type error*?

1.10 Determine the types of `3`, `even` and `even 3`. How did you determine the last one? Now determine the types of `head`, `[1,2,3]` and `head [1,2,3]`. What happens when applying a polymorphic function to an actual parameter?

1.11 Using patterns you could try to define the following function to test if a number is a prime:

```
is_prime ((x+2)*(y+2)) = False
is_prime x             = True
```

a. Why did the author write `x+2` and `y+2`, and not just `x` and `y`?

b. Unfortunately this definition is not allowed. Against which rule is sinned?

c. Why would the designer of the language have incorporated this rule?

d. If this rule were not there, how could you define the function `sqrt`?

1.12 As a condition of the being-useful of a recursive function the condition is mentioned in section 1.4.4 that the parameter of the recursive call should be simpler than the parameter of the function being defined, and that there should be a non-recursive base case. Now study the next definition of the factorial function:

page 12

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

a. What happens when evaluating `fac (-3)` ?

b. How can the condition of being useful be formulated more exact?

1.13 What is the difference between a *list* and the mathematical *set*?

1.14 In section 1.4.3 the function `even` is defined by giving separate definitions for even and odd. In section 1.4.4 the recursive definitions for involution is given.

page 11
page 12

a. Now supply an alternative definition for involution, where you handle the cases of n being even and odd separately. You can use the fact that $x^n = (x^{n/2})^2$.

b. Which intermediate results are calculated when calculating 2^{10} when using the old and new definition?

1.15 Given the function:

```
threecopy x   =   [x,x,x]
```

What would be the value of the expression

```
map threecopy (sums [1..4])
```

Chapter 2

Numbers and functions

2.1 Operators

2.1.1 Operators as functions and vice versa

An operator is a function of two parameters that is written between its parameters instead of in front of them. Names of functions consist of letters and digits; 'names' of operators consist of symbols (see section 1.3.2 for the precise rules for naming). page 6

Sometimes it would be more clear to write an operator before its parameters, of a function between them. In Gofer, two special notations are available for this situation:

- An operator in parentheses behaves as the corresponding function;
- A function between *back quotes* behaves as the corresponding operator.

(A 'back quote' is the symbol `, not to be confused with the apostrophe or single quote '. On most keyboards the back quote key is located to the left of the 1-key).

It is thus allowed to write `(+) 1 2` instead of `1+2`. This notation was already used in section 1.5.5 to be able to declare the type of `+`: page 17

```
(+)  ::  Num a => a->a->a
```

The parentheses are necessary because at the lefthand side of `::` an expression is required. A sole operator is not an expression, but a function is.

Conversely, it is possible to write ``1 `f` 2`` instead of `f 1 2`. This notation is used mainly to write an expression more clearly; the expression ``3 `choose` 5`` is easier to read than `choose 3 5`. Of course, this is only possible if the function has two parameters.

2.1.2 Priorities

In primary school we learn that 'multiplication precedes addition'. Said in different words: the priority of multiplication is higher than that of addition. Gofer also knows about these priorities: the value of the expression `2*3+4*5` is 26, not 50, 46, or 70.

There are more levels of priority in Gofer. The comparison operators, like `<` and `==`, have lower priority than the arithmetical operators. Thus the expression `3+4<8` has the meaning that you would expect: 3+4 is compared with 8 (with returns `False`), and not: 3 is added to the result of `4<8` (which would be a type error).

Altogether there ar nine levels of priority. The priorities of the operators in the prelude are as below:

level 9	`.` and `!!`
level 8	`^`
level 7	`*`, `/`, `` `div` ``, `` `rem` `` and `` `mod` ``
level 6	`+` and `-`
level 5	`:`, `++` and `\\`
level 4	`==`, `/=`,`<`, `<=`, `>`, `>=`, `` `elem` `` and `` `notElem` ``
level 3	`&&`
level 2	`||`
level 1	(not used in the prelude)

(As of yet, not al these operators have been discussed; some of them will be discussed in this chapter or in later ones). To override these priorities you can place parentheses in an expression around subexpressions that must be calculated first: in `2*(3+4)*5` the subexpression `3+4` is calculated first, despite the fact that `*` has higher priority than `+`.

Calling of functions (the 'invisible' operator between `f` and `x` in `f x`) has topmost priority. The expression `square 3 + 4` therefore calculates 3 squared, and then adds 4. Even if you write `square 3+4` first the function is called, and only then the addition is performed. To calculate the square of 7 parentheses are required to override the high priority of function calls: `square (3+4)`.

Also while defining functions using patterns (see section 1.4.3) it is important to remember that calling functions has highest priority. In the definition page 11

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

the parentheses around `x:xs` are essential; without parentheses this would be interpreted as `(sum x):xs`, which is not a valid pattern.

2.1.3 Association

The priority rule still leaves undecided what happens when operators with equal priority occur in an expression. For addition, this is not a problem, but for e.g. subtraction this is an issue: is the result of `8-5-1` the value 2 (first calculate 8 minus 5, and subtract 1 from the result), or 4 (first calculate 5 minus 1, and subtract that from 8)?

For each operator in Gofer it is defined in which order an expression containing multiple occurrences of it should be evaluated. In principle, there are four possibilities for an operator, say $\oplus$:

- the operator $\oplus$ *associates to the left*, i.e. $a \oplus b \oplus c$ is interpreted as $(a \oplus b) \oplus c$;
- the operator $\oplus$ *associates to the right*, i.e. $a \oplus b \oplus c$ is evaluated as $a \oplus (b \oplus c)$;
- the operator $\oplus$ is *associative*, i.e. it doesn't matter in which order $a \oplus b \oplus c$ is evaluated;
- the operator $\oplus$ is *non-associative*, i.e. it is not allowed to write $a \oplus b \oplus c$; you always need parentheses to indicated the intended interpretation.

For the operators in the prelude the choice is made according to mathematical tradition. When in doubt, the operators are made non-associative. For associative operators a more or less arbitrary choice is made for left- or right-associativity (selecting the more efficient of the two).

The operators that associate to the **left** are:

- the 'invisible' operator function application, so `f x y` means `(f x) y` (the reson for this is discussed in section 2.2); page 23
- the operator `!!` (see section 3.1.2); page 40
- the operator `-`, so the value of `8-5-1` is 2 (as usual in mathematics), not 4.

The operators that associate to the **right** are:

- the operator `^` (involution), so the value of `2^2^3` is $2^8 = 256$ (as usual in mathematics), not $4^3 = 64$;
- the operator `:` ('put in front of'), making the value of `1:2:3:x` a list that starts with the values 1, 2, and 3.

The **non-**associative operators are:

- the operator `/` and the related operators `div`, `rem` en `mod`. The result of `64/8/2` is therefore neither 4 nor 16, but the error message

  ```
  ERROR: Ambiguous use of operator "/" with "/"
  ```

- the operator `\\` (see exercise 3.6); page 64
- the comparison operators `==`, `<` etcetera: most of the time it is meaningless anyway to write `a==b==c`. To test if `x` is between 2 and 8, don't write `2<x<8` but `2<x && x<8`.

The associative operators are:

- the operators `*` and `+` (these operatoren are evaluated left-associative according to mathematical tradition);
- the operators `++`, `&&` and `||` (these operators are evaluated right-associative for reasons of efficiency);
- the operator `.` (see section 2.3.3). page 27

2.1.4 Definition of operators

If you define an operator yourself, you have to indicate its priority and order of association. As an example, we look at the way in which in the prelude is defined that ^ has priority 8 and associates to the right:

```
infixr  8  ^
```

For operators that should associate to the left you use the reserved word `infixl`, for non-associative operators `infix`:

```
infixl  6  +, -
infix   4  ==, /=, `elem`
```

S By defining the priorities cleverly, is is possible to avoid parentheses as much as possible. Consider for instance the operator 'n choose k' from section 1.2.2: page 4

```
n `choose` k  =  fac n / (fac k * fac (n-k))
```

or using a new symbol combination:

```
n !^! k  =  fac n / (fac k * fac (n-k))
```

Because at some time it might be useful to calculate $\binom{a+b}{c}$, it would be handy if `` `choose` `` had lower priority than `+`; than you could leave out parentheses in ``a+b`choose`c``. On the other hand, expressions like $\binom{a}{b} < \binom{c}{d}$ may be useful. By giving `'choose'` higher priority than `<`, again no parentheses are necessary.

For the priority of `'choose'`, it is therefore best to choose 5 (lower than `+` (6), but higher than `<` (4)). About the associativity: as it is not very customary to calculate ``a`choose`b`choose`c``, it is best to make the operator non-associative. The priority/associativity definition for our new operator will therefore be:

```
infix  5  !^!, `choose`
```

2.2 Currying

2.2.1 Partial parametrization

Suppose `plus` is a function adding two whole numbers. In an expression this function can get two parameters, for instance `plus 3 5`.

In Gofer it is also allowed to give *less* parameters to a function. If `plus` is provided with only *one* parameter, for example `plus 1`, a function remains which still expects a parameter. This function can be used to define other functions:

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

Calling a function with less parameters than it expects is named *partial parametrization*.

A second use of a partially parametrized function is that it can serve as a parameter for another function. The function parameter of the function `map` (applying a function to all elements of a list) is for instance often a partially parametrized function:

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

The expression `plus 5` can be regarded as 'the function adding 5 to something'. In the example, this function is applied by `map` to all elements of the list `[1,2,3]`.

The possibility of partial parametrization gives a brand new look on the type of `plus`. If `plus 1`, just like `successor`, has the type `Int->Int`, then `plus` has to be a function of `Int` (the type of `1`) to that type:

```
plus  ::  Int -> (Int->Int)
```

By assuming `->` associates to the right, the parentheses can be left out:

```
plus :: Int -> Int -> Int
```

This is exactly the notation for the type of a function with two parameters, discussed in section 1.5.4. page 17

In fact, there are no 'functions with two parameters'. There are only functions with *one* parameter, which can return a function if desired. This function has a parameter in its turn, creating the illusion of the original function having two parameters.

This trick, the simulation of functions with more parameters by a function with *one* parameter returning a function, is called *Currying*, after the English mathematician Haskell Curry. The function itself is called a *curried* function. (This tribute is not exactly right, because this method was used earlier by M. Schönfinkel).

2.2.2 Parentheses

The 'invisible operator' function application associates to the left. That means that the expression `plus 1 2` is regarded as `(plus 1) 2`. This corresponds exactly with the type of `plus`: this is a function expecting an integer (1 in the example) and returning a function, which in his turn accepts an integer (2 in the example).

Association of function application to the right would be meaningless scribbling: in `plus (1 2)` first 1 would be applied to 2 (??) and after that `plus` to the result.

If there is a whole sequence of letters in an expression, the first has to accept the others sequentially:

```
f a b c d
```

is interpreted as

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

If `a` has type `A`, `b` type `B` etc, then the type of `f` will be:

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

or, when writing all the parentheses:

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

Without parentheses this is of course much clearer The association of `->` and a function application is chosen in such a way, Currying happens 'without noise': function application associates to the left, and `->` associates to the right. In an easy to remember credo:

if there are no parentheses,
they are placed in such a way, Currying works.

Parentheses are only needed if you want do divert from this. This happens for example in the next cases:

- In the type if a function has a function as a *parameter* (when Currying, a function has a function as a *result*). The type of `map` is for example

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

  The parentheses in `(a->b)` are essential, otherwise `map` would be a function with three parameters.
- In an expression if the result of a function is passed to another function, and not the function itself. For example, if you calculate the square of the sine of a number:

  ```
  square (sin pi)
  ```

  If the parentheses were left out, it would seem like `square` would have to be applied to `sin` (??) and the result of that to `pi`.

2.2.3 Operator sections

To partially parametrize operators there are two special notations available:

- with $(\oplus x)$ the operator $\oplus$ will be partially parametrized with x as the *right* parameter;
- with $(x\oplus)$ the operator $\oplus$ will be partially parametrized with x as the *left* parameter;

These notations are called *operator sections*.

With operator sections a number of functions can be defined:

```
successor    =  (+1)
double       =  (2*)
half         =  (/2.0)
reverse      =  (1.0/)
square       =  (^2)
twoPower     =  (2^)
oneDigit     =  (<=9)
isZero       =  (==0)
```

However, the main application of operator sections is passing the partially parametrized operator to a function:

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

2.3 Functions as a parameter

2.3.1 Functions on lists

In a functional programming language, functions behave in many aspects just like other values, like numbers and lists. For example:

- functions have a *type*;
- functions can be the *result* of other functions (which is used a lot with Currying);
- functions can be used as a *parameter* of other functions.

With this last possibility it is possible to write general functions, of which the specific behavior is determined by a function given as a parameter.

Functions with functions as a parameter are sometimes called *higher-order functions*, to distinguish them from 'homely' numerical functions.

The function `map` is an example of an higher-order function. This function takes care of the principle of 'handling all elements in a list'. What has to be done to the elements of the list, is specified by the function, which, next to the list, is passed to `map`.

The function `map` can be defined as follows:

```
map           ::   (a->b) -> [a] -> [b]
map f []       =   []
map f (x:xs)   =   f x : map f xs
```

The definition uses patterns: the function is defined separately for the case the second parameter is a list without elements, and for the case the list consists of a first element `x` and a remainder `xs`. The function is recursive: in the case of a non-empty list the function `map` is called again. Thereby the parameter is shorter (`xs` is shorter than `x:xs`) so that finally the non-recursive part of the function will be used.

Another often used higher order function on lists is `filter`. This function returns those elements of a list, which satisfy a certain condition. Which condition that is, will be determined by a function which is passed as a parameter to `filter`. Examples of the use of `filter` are:

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

If the list elements are of type `a`, then the function parameter of `filter` has to be of type `a->Bool`. The definition of `filter` is recursive too:

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

In the case the list is not empty (so it is of the form `x:xs`), there are two cases: either the first element `x` satisfies `p`, or it does not. If so, it will be put in the result; the other elements are (by a

recursive call) 'filtered'.

You can design higher order functions by tracking down the similarities in function definitions. For instance, take a look at the definitions of the functions `sum` (calculating the sum of a list of numbers), `product` (calculating the product of a list of numbers), and `and` (checking a list of Boolean values if they are all `True`):

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

The structure of these three definitions is the same. The only difference is the value which is returned for an empty list (`0`, `1` or `True`), and the operator being used to attach the first element to the result of the recursive call (`+`, `*` or `&&`).

By passing these two variables as parameters, a generally useful higher order function is born:

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

Given this function, the other three functions can be defined by partially parametrizing the general function:

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

The function `foldr` can be used in many other cases; that is why it is defined as a standard function in the prelude.

The name of `foldr` can be explained as follows. The value of

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

equals the value of the expression

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

The function `foldr` 'folds' the list into one value, by inserting between all elements the given operator, starting on the right side with the given initial value. (There is also a function `foldl` which starts on the left side).

Higher order functions, like `map` and `foldr`, play in functional languages the role which control structures (like `for` and `while`) play in imperative languages. However, these control structures are built in, while the functions can be defined by yourself. This makes functional languages flexible: there is little built in, but you can make everything.

2.3.2 Iteration

In mathematics *iteration* is often used. This means: take an initial value, apply some function to that, until the result satisfies some condition.

Iteration can be described very well by a higher order function. In the prelude this function is called `until`. The type is:

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

The function has three parameters: the property the final result should satisfy (a function `a->Bool`), the function which is to be applied repeatedly (a function `a->a`), and an initial value (of the type `a`). The final result is also of type `a`. The call `until p f x` can be read as: 'until `p` is satisfied apply `f` to `x`'.

The definition of `until` is recursive. The recursive and non-recursive cases are this time not distinguished by patterns, but by conditions with 'vertical bar/Boolean expression':

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

If the initial value `x` satisfies the property `p` immediately, then the initial value is also the final value. If not the function `f` is applied *once* to `x`. The result, `(f x)`, will be used as a new initial value in the recursive call of `until`.

Like all higher order functions `until` can be called with partially parametrized functions. For instance, the expression below calculates the first power of two which is greater than 1000 (start with 1 and keep on doubling until the result is greater than 1000):

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

In contrast to previously discussed recursive functions, the parameter of the recursive call of `until` is not 'smaller' than the formal parameter. That is why `until` does not always terminate with a result. When calling `until (<0) (+1) 1` the condition is never satisfied; the function `until` will keep on counting indefinitely, and it will never come with a result.

If the computer does not answer because it is in such an infinite recursion, the calculation can be interrupted by pressing the 'ctrl'-key and the C-key at the same time:

```
? until (<0) (+1) 1
ctrl-C
{Interrupted!}
?
```

2.3.3 Composition

If f and g are functions, than $g \circ f$ is the mathematical notation for 'g after f': the function which applies f first, and then g to the result. In Gofer the operator which composes two functions is also very useful, for instance it makes it possible to define:

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

The operator `` `after` `` can be defined as a higher order operator:

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

Not all functions can be composed. The range of f has to be equal to the domain of g. So if f is a function $a \to b$, g has to be a function $b \to c$. The composition of two functions is a function which goes directly from a to c. This is also shown by the type of `after`:

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

Because `->` associates to the right, the third pair of parentheses is redundant. Thus the type of `after` can also be written as

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

The function `after` can thus be regarded as function with three parameters; by the Currying mechanism this is the same as a function with two parameters returning a function (and the same as a function with one parameter returning a function with one parameter returning a function). It is indeed possible to define `after` as a function with three parameters:

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

So it is not necessary to name the function `h` separately in a `where` clause (although it is allowed, of course). In the definition of `odd`, `after` is in fact partially parametrized with `not` and `even`. The third parameter is not provided yet: it will be given if `odd` is called.

The use of the operator `after` may perhaps seem limited, because functions like `odd` can be defined also by

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

However, a composition of two functions can serve as a parameter for another higher order function, and then it is easy it does not need to be named. The next expression evaluates to a list with all odd numbers between 1 and 100:

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

In the prelude the function composition operator is defined. It is typed like a dot (because the character $\circ$ is not on most keyboards). So you can write:

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

The operator is especially useful when many functions have to be composed. The programming can be done at a function level as a whole (see also the title of this reader). low level things like numbers and list have disappeared from sight. Isn't it much nicer to write `f=g.h.i.j.k` instead of `f x=g(h(i(j(k x))))`?

2.3.4 The lambda notation

In section 2.2.1 it was noted that a function which is passed as a parameter to another function often springs from partial parametrization, with or without the operator section notation: page 23

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

In other cases the function passed as a parameter can be constructed by composing other functions:

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

But sometimes it is too complicated to make a function that way, for example if we would like to calculate $x^2 + 3x + 1$ for all x in a list. Then it is always possible to define the function separately in a `where` clause:

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

However, if this happens too much it gets a little annoying to keep on thinking of names for the functions, and then defining them afterwards.

For these situations there is a special notation available, with which functions can be created without giving them a name. So this is mainly important if the function is only needed to be passed as a parameter to another function. The notation is as follows:

`\` *pattern* `->` *expression*

This notation is known as the *lambda notation* (after the greek letter λ; the symbol `\` is the closest approximation for that letter available on most keyboards...)

An example of the lambda notation is the function `\x -> x*x+3*x+1`. This can be read as: 'the function which calculates when given the parameter x the value of $x^2 + 3x + 1$'. The lambda notation is often used when passing functions as a parameter to other functions, for instance:

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

2.4 Numerical functions

2.4.1 Calculation with whole integers

When dividing whole integers (`Int`) the part behind the decimal point is lost: `10/3` equals 3. Still it is not necessary to use `Float` numbers if you do not want to loose that part. On the contrary: often the *remainder* of a division is more interesting than the decimal fraction. The rest of a division is the number which is on the last line of a long division. For instance in the division `345/12`

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

is the quotient 28 and the remainder 9.

The remainder of a division can be determined with the standard function `rem`:

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

The remainder of a division is for example useful in the next cases:

- Calculating with times. For example, if it is now 9 o'clock, then 33 hours later the time will be `(9+33)`rem`24` = 20 o'clock.

- Calculating with weekdays. Encode the days as 0=Sunday, 1=Monday, ..., 6=Saturday. If it is day 3 (Wednesday), then in 40 days it will be `(3+40)‘rem‘7` = 1 (Monday).
- Determination of divisibility. A number is divisible by n if the remainder of the division by n equals zero.
- Determination of digits. The last digit of a number `x` equals `x‘rem‘10`. The next digit equals `(x/10)‘rem‘10`. The second next equals `(x/100)‘rem‘10`, etcetera.

As a more extensive example of calculating with whole numbers two applications are discussed: the calculation of a list of prime numbers and the calculation of the day of the week on a given date.

Calculating a list of prime numbers

A number can be divided by another number if the remainder of the division by that number equals zero. The function `divisible` tests two numbers on divisibility:

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

The denominators of a number are those numbers it can be divided by. The function `denominators` computes the list of denominators of a number:

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

Note that the function `divisible` is partially parametrized with `x`; by calling `filter` *those* elements are filtered out `[1..x]` by which `x` can be divided.

A number is a prime number iff it has exactly two divisors: 1 and itself. The function `prime` checks if the list of denominators indeed consists of those two elements:

```
prime   ::   Int -> Bool
prime x  =   denominators x == [1,x]
```

The function `primes` finally determines all prime numbers up to a given upper bound:

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

Although this may not be the most efficient way to calculate primes, it is the easiest way: the functions are a direct translation of the mathematical definitions.

Compute the day of the week

On what day will be the last New Year's Eve this century?

```
? day 31 12 1999
Friday
```

If the number of the day is known (according to the mentioned coding 0=Sunday etc.) the function `day` is very easy to write:

```
day d m y  =  weekday (daynumber d m y)
weekday 0  =  "Sunday"
weekday 1  =  "Monday"
weekday 2  =  "Tuesday"
weekday 3  =  "Wednesday"
weekday 4  =  "Thursday"
weekday 5  =  "Friday"
weekday 6  =  "Saturday"
```

The function `weekday` uses seven patterns to select the right text (a quoted word is a text; for details see section 3.2.1).

page 45

The function `daynumber` chooses a Sunday in a distant past and adds:

- the number of years passed since then times 365;
- a correction for the elapsed leap years;
- the lengths of this years already elapsed months;
- the number of passed days in the current month.

Of the resulting (huge) number the remainder of a division by 7 is determined: this will be the required day number.

Since the calendar adjustment of pope Gregorius in 1752 the following rule holds for leap years (years with 366 days):

- a year divisible by 4 is a leap year (e.g. 1972);
- but: if it is divisible by 100 it is no leap year (e.g. 1900);
- but: if it is divisible by 400 it *is* a leap year (e.g. 2000).

As origin of the day numbers we could choose the day of the calendar adjustment, but it will be easier to extrapolate back to the fictional year 0. The function `daynumber` will be easier by this extrapolation: the 1st of January of the year 0 would be on a Sunday.

```
daynumber d m y  =  ( (y-1)*365
                    + (y-1)/4
                    - (y-1)/100
                    + (y-1)/400
                    + sum (take (m-1) (months j))
                    + d
                    ) `rem` 7
```

The call `take n xs` returns the first `n` elements of the list `xs`. The function `take` can be defined by:

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

The function `months` should return the lengths of the months in a given year:

```
months y  =  [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
       where  feb  |  leap y       =  29
                   |  otherwise    =  28
```

The function `leap` used will be defined by the previously mentioned rules:

```
leap y  = divisible y 4 && (not(divisible y 100) || divisible y 400)
```

Another way to define this is:

```
leap y  |  divisible y 100 =  divisible y 400
        |  otherwise       =  divisible y 4
```

With this the function `day` and all needed auxiliary functions are finished. It might be sensible to add to the function `day` it can only be used for years after the calendar adjustment:

```
day d m y  |  y>1752  =  weekday (daynumber d m y)
```

calling `day` with a smaller year yields automatically an error.

(End of examples).

When designing the two programs in the examples two different strategies were used. In the second example the required function `day` was immediately defined. For this the auxiliary function `weekday` and `daynumber` were needed. To implement `daynumber` a function `months` was required, and this `months` needed a function `leap`. This approach is called *top-down*: start with the most important, and gradually fill out all the details.

The first example used the *bottom-up* approach: first the function `divisible` was written, and with the help of that one the function `denominators`, with that a function `prime` and concluding with the required `prime`.

It does not matter for the final result (the interpreter does not care in which order the functions are defined). However, when designing a program it can be useful to determine which approach you use, (bottom-up or top-down), or that you even use a concurrent approach (until the 'top' hits the 'bottom').

2.4.2 Numerical differentiating

When computing with `Float` numbers a precise answer is mostly not possible. The result of a division for instance is rounded to a certain number of decimals (depending on the calculational preciseness of the computer):

```
? 10.0/6.0
1.6666667
```

For the computation of a number of mathematical operations, like `sqrt`, also an approximated

value is used. Therefore, when designing your own functions which operate on `Float` numbers it is acceptable the result is also an approximation of the 'real' value.

An example of this is the calculation of the derivative function. The mathematical definition of the derivative f' of the function f is:

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

The precise value of the limit cannot be calculated by the computer. However, an approximated value can be obtained by using a very small value for h (not *too* small, because that would result in unacceptable round-off errors).

The operation 'derivative' is a higher-order function: a function goes in and a function comes out. The definition in Gofer could be:

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

Due to the Currying mechanism the second pair of parentheses in the type can be left out, since `->` associates to the right.

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

So the function `diff` can also be regarded as a function of two parameters: the function of which the derivative should be computed, and the location where its value should be calculated. From this point of view the definition could have been:

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

The two definitions are perfectly equivalent. For clarity of the program the second version might be preferred, since it is simpler (it is not needed to name the function `f'` and then define it). On the other hand the first definition emphasizes that `diff` can be regarded as a function transformation.

The function `diff` is very amenable to partial parametrization, like in the definition:

```
derivative_of_sine_squared  =  diff (squared.sin)
```

The value of `h` is in both definition of `diff` put in a `where` clause. Therefore it is easily adjusted, if the program would have to be changed in the future (naturally, this can also be done in the expression itself, but then it has to be done twice, with the danger of forgetting one).

Even more flexible it would be, to define the value of `h` as a parameter of `diff`:

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

By defining `h` as the first parameter of `flexDiff`, this function can be partially parametrized too, to make different versions of `diff`:

```
roughDiff  =  flexDiff 0.01
fineDiff   =  flexDiff 0.0001
superDiff  =  flexDiff 0.000001
```

2.4.3 Home-made square root

In Gofer the function `sqrt` is built in to calculate the square root of a number. In this section a method will be discussed how you can make your own root function, if it would not have been built in. It demonstrates a technique often used when calculating with `Float` numbers. The function is generalized in section 2.4.5 to inverses of other functions than the square function. At that point will also be explained why the described method works properly. page 33

For the square root of a number x the next property holds:

if y is a good approximation of $\sqrt{x}$
then $\frac{1}{2}(y + \frac{x}{y})$ is a better approximation.

This property can be used to calculate the root of a number x: take 1 as a first approximation, and keep on improving the approximation until the result is satisfactory. The value y is good enough for $\sqrt{x}$ if y^2 is not too different from x.

For the value of $\sqrt{3}$ the approximations y_0, y_1 etc. are as follows:

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

The square of the last approximation is only 10^{-18} wrong from 3.

For the process 'improving an initial value until good enoughthe function `until` from section 2.3.2 can be used: page 26

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

The operator `~=` is the 'about equal to' operator, which can be defined as follows:

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

The higher-order function `until` operates on the *functions* `improve` and `goodEnough` and on the initial value `1.0`.

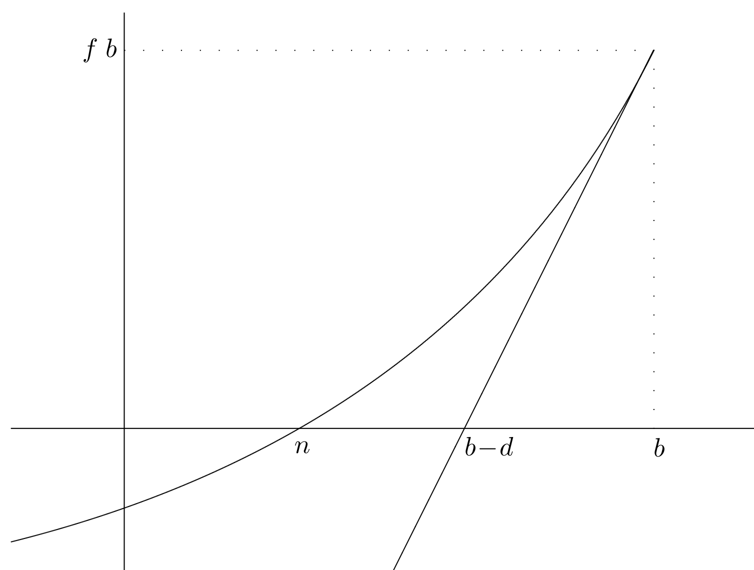
Although `improve` is next to `1.0`, the function `improve` is not applied immediately to `1.0`; instead of that both will be passed to `until`. This is caused by the Currying mechanism: it is like if there were parentheses as in `(((until goodEnough) improve) 1.0)`. Only when looking closely at the definition of `until` it shows that `improve` is still applied to `1.0`.

There are some other quite interesting observations possible with `improve` and `goodEnough`. These functions can, except for their parameter `y`, make use of `x`. So to this functions it looks like `x` is a constant. (Compare the definitions of the 'constants' `d` and `n` in the definition of `abcFormula'` in section 1.4.1). page 9

2.4.4 Zero of a function

Another numerical problem which can be solved by iteration with `until` is the determination of the zero of a function.

Consider a function f of which the zero n is to be found. Make b an approximation for the zero. Then the intersection point of the tangent of f in b with the x axis is a better approximation for the zero (see picture).



The intersection point to be found is on distance d from the first approximation b. The value of d can be calculated as follows. The slope of the tangent of f in b equals $f'(b)$. On the other hand this slope equals $f(b)/d$. So $d = f(b)/f'(b)$.

With this an improvement function is found: if b is an approximation for the zero of f, then $b - f(b)/f'(b)$ is a better approximation. This is known as the 'method of Newton'. (The method does not always work for functions with local extremes: you can keep on 'dangling around'. We do not go into that.)

Just as with `square` the Newton's improve function can be used as a parameter of `until`. As a 'good enough' function we can check this time if the f value in the approximated zero has become small enough.

```
zero f  =  until goodEnough improve 1.0
       where  improve b    =  b - f b / diff f b
              goodEnough b =  f b ~= 0.0
```

As the first approximation 1.0 is chosen, but this could have been 17.93 as well. This point has of course to be in the domain of `f`. The derivative function from section 2.4.2 comes in handy too. page 30

2.4.5 Inverse of a function

The zero of the function f with $f\ x = x^2 - a$ equals $\sqrt{a}$. So the root of a can be determined by searching for the zero of f. Since the function `zero` can be used, `root` can also be written as:

```
root a  =  zero f
       where  f x  =  x*x-a
```

The cubic root can be calculated in the same fashion:

```
cubic a  =  zero f
        where  f x  =  x*x*x-a
```

In fact the inverse of every desired function can be calculated by using this function in the definition of f, for example:

```
arcsin a  =  zero f
      where  f x  =  sin x - a
arccos a  =  zero f
      where  f x  =  cos x - a
```

Slowly a pattern is emerging at the horizon in all these definitions. This is always a signal to define a higher order function, which generalizes this (see also section 2.3.1, where `foldr` was defined as a generalization of `sum`, `product` and `and`). The higher order function is in this case `inverse`, which has as an extra parameter a function g of which the inverse has to be calculated: page 26

```
inverse g a  =  zero f
         where  f x  =  g x - a
```

If you eventual see the pattern, such a higher order function is not any harder to define than the other definitions. The other definitions are special cases of the higher order function, and can now also be written as partial parametrization:

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

The function `inverse` can be used as desired as a function with two parameters (a function and a `Float`) and `Float` as a result, or as a function with *one* parameter (a function) and a function as a result. This is because the type of `inverse` equals

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

Which can also be written as

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

due to the right[1] association of `->`.

The root function from section 2.4.3 uses the Newton method in a way too. This can be seen by taking the definition of `root` above: page 31

[1] In both meanings of the word 'right'

```
root a  =  zero f
      where  f x = x*x-a
```

Then exchange the call of `zero f` by the definition of that:

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

In this specific case it is not needed to find `diff` numerically: the derivative of the used `f` is the function `(2*)`. So the formula of `improve b` can be simplified:

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

This is exactly the improvement formula used in section 2.4.3. page 31

Exercises

2.1 Explain the placement of the parentheses in the expression `(f (x+h) - f x) / h`.

2.2 Which parentheses are redundant in the following expressions?
- `(plus 3) (plus 4 5)`
- `sqrt(3.0) + (sqrt 4.0)`
- `(+) (3) (4)`
- `(2*3)+(4*5)`
- `(2+3)*(4+5)`
- `(a->b)->(c->d)`

2.3 Why is it in mathematics customary to have involution associate to the right?

2.4 Is the operator `after` (`.`) associative?

2.5 In the language Pascal the operator `&&` has the same priority as `*` and `||` has the same priority as `+`. Why is this not convenient?

2.6 For each type, give a possible function
- `(Float -> Float) -> Float`
- `Float -> (Float -> Float)`
- `(Float -> Float) -> (Float -> Float)`

2.7 In section 2.3.3 it is stated that `after` can also be regarded as a function with *one* parameter. page 27 How can this be seen from the type? Give a definition of `after` in the form of `after y =` ...

2.8 Design a function which determines how many years an amount should be in the bank to, with a given interest rate, collect a given desired amount.

```
f interest end start
      | end <= start   =  0
      | otherwise      =  1 + f interest end ((1+interest)*start)
```

2.9 Give a definition of `cubicRoot` which does not use numerical differentiation (indirectly by `zero` is not allowed either).

2.10 Define the function `inverse` from section 2.4.5 by utilizing the lambda notation. page 33

2.11 Why is it possible to write the functions `root` and `cubicRoot` without using the function `diff`, but why is it not possible to write a general function `inverse` in this fashion ?

2.12 Write a function `integrate`, which calculates the integral of a function between two given boundaries. The function does this by dividing the integration area into a (to be specified) number of areas, and then approximating every area by using a linear function. In which order is it best to supply the parameters, to make the function useful with partial parametrization?

Chapter 3

Data structures

3.1 Lists

3.1.1 Structure of a list

Lists are used to group a number of elements. Those elements should be of the *same type*. For every type there exists a type 'list of that type'. Therefore there are lists of integers, lists of floats and lists of functions from int to int. But also a number of lists of the same type can be stored in a list; in this way you get lists of lists of integers, lists of lists of lists of booleans and so forth.

The type of a list is denoted by the type of the elements between square brackets. The types listed above can thus be expressed shorter by `[Int]`, `[Float]`, `[Int->Float]`, `[[Int]]` en `[[[Bool]]]`.

There are several ways to construct a list: enumeration, construction using `:`, and numeric intervals.

Enumeration

Enumeration of the elements often is the easiest method to build a list. The elements must be of the same type. Some examples of list enumerations with their types are:

```
[1, 2, 3]             :: [Int]
[1, 3, 7, 2, 8]       :: [Int]
[True, False, True]   :: [Bool]
[sin, cos, tan]       :: [Float->Float]
[ [1,2,3], [1,2] ]    :: [[Int]]
```

For the type of the list it doesn't matter how many elements there are. A list with three integer elements and a list with two integer elements both have the type `[Int]`. That is why in the fifth example the lists `[1,2,3]` and `[1,2]` can in their turn be elements of one list of lists.

The elements of the elements needn't be constants; they may be determined by a computation:

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

The used function must then deliver the desired type as a result.

There are no restrictions on the number of elements of a list. A list therefore can contain just one element:

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

A list with one element is also called a *singleton list*. The list `[[1,2,3]]` is a singleton list as well, for it is a list of lists containing one element (the list `[1,2,3]`).

Note the difference between an *expression* and a *type*. If there is a type between the square brackets, the whole is a type (for example `[Bool]` or `[[Int]]`). If there is an expression between the square brackets, the whole is also an expression (a singleton list, for example `[True]` or `[3]`).

Furthermore the number of elements of a list can be zero. A list with zero elements is called the *empty list*. The empty list has a polymorphic type: it is a 'list of whatever'. At positions in a polymorphic type where an arbitrary type can be substituted type variables are used (see section 1.5.3); so the type of the empty list is `[a]`:

page 16

```
[]  ::  [a]
```

The empty list can be used in an expression wherever a list is needed. The type is then determined by the context:

`sum []`	`[]` is an empty list of numbers
`and []`	`[]` is an empty list of Booleans
`[ [], [1,2], [3] ]`	`[]` is an empty list of numbers
`[ [1<2,True], [] ]`	`[]` is an empty list of Booleans
`[ [[1]], [] ]`	`[]` is an empty list of lists of numbers
`length []`	`[]` is an empty list (doesn't matter of what type)

Construction using :

Another way to build a list is by using the operator `:`. This operator puts an element in front of a list and as a result makes a longer list.

```
(:)  ::  a -> [a] -> [a]
```

If, for example, `xs` is the list `[3,4,5]` , then `1:xs` is the list `[1,3,4,5]`. Using the empty list and the operator `:` you can construct every list. For example, `1:(2:(3:[]))` is the list `[1,2,3]`. The operator `:` associates to the right, so you can just write `1:2:3:[]`.

In fact this way of constructing lists is the only 'real' way. An enumeration of a list often looks more neatly in programs, but it has the same meaning as the corresponding expression with uses of the operator `:`. That is why an enumeration costs time:

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

Every call to `:` (which you don't see, but which is there all right) costs two reductions.

Numeric intervals

The third way to construct a list is the interval notation: two numeric expression with two dots between and square brackets surrounding them:

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

(Although the dot can be used as a symbol in operators, `..` is not an operator. It is one of the symbol combinations that were reserved for special use in section 1.3.2.) page 6

The value of the expression `[x..y]` is calculated by evaluating `enumFromTo x y`. The function `enumFromTo` is defined as follows:

```
enumFromTo x y  |  y<x       =  []
                |  otherwise =  x : enumFromTo (x+1) y
```

So if `y` is smaller than `x` the list is empty; otherwise `x` is the first element and the next element is one greater (unless it is greater than `y`), etc.

The numeric interval notation is only there for convenience and makes use of language somewhat easier; it would not be a great loss if this construction was not available, because the function `enumFromTo` can be used instead.

3.1.2 Functions on lists

Functions on lists are often defined using *patterns*: the function is defined for the empty list and the list of the form `x:xs` separately. For a list is either empty or has a first element `x` in front of a (possibly empty) list `xs`.

A number of definitions of functions on lists has already been discussed: `head` and `tail` in section 1.4.3, `sum` and `length` in section 1.4.4, en `map`, `filter` and `foldr` in section 2.3.1. Even though these are all standard functions defined in the prelude and you don't have to define them yourself, it is important to look at their definitions. Firstly because they are good examples of functions on lists, secondly because the definition often is the best description of what a standard function does.

page 11
page 12
page 26

In this paragraph more definition of functions on lists follow. A lot of these functions are recursively defined, which means that in the case of the pattern `x:xs` they call themselves with the (smaller) parameter `xs`.

Comparing and ordering lists

Two lists are equal if they contain exactly the same elements in the same order. This is a definition of the function `eq` with which the equality of lists can be tested:

```
[]     `eq` []      =  True
[]     `eq` (y:ys)  =  False
(x:xs) `eq` []      =  False
(x:xs) `eq` (y:ys)  =  x==y && xs `eq` ys
```

In this definition both the first and the second parameter can be empty or non-empty; there is a definition for all four combinations. In the fourth case the corresponding elements are compared (`x==y`) and the operator is called recursively on the tails of the lists (`xs`eq`ys`).

As the operator `==` is used on the list elements this is an overloaded function. Its type is:

```
eq  ::  Eq a => [a] -> [a] -> Bool
```

The function `eq` is not defined in the prelude. Instead lists are made member of the class `Eq`, so that the operator `==` can also be used on lists. The type of the list elements should also be a member of the `Eq` class. Therefore lists of functions are not comparable, because functions themselves are not. However, lists of lists of integers are comparable, because lists of integers are comparable (because integers are). In section 3.1.2 the prelude definition of the operator `==` on lists is discussed. That definition is completely analogous to that of the function `eq` above.

page 39

If the elements of a list can be ordered using $<$, $\leq$ enz., then lists can also be ordered. This is done using the *lexicographical ordering* ('dictionary ordering'): the first elements of the lists determine the order, unless they are same; in that case the second element is decisive unless they are equal as well, etcetera. For example, `[2,3]<[3,1]` and `[2,1]<[2,2]` hold. If one of the two lists is equal to the beginning of the other then the shortest one is the 'smallest', for example `[2,3]<[2,3,4]`. The fact that the word 'etcetera' is used in this description, is a clue that recursion is needed in the definition of the function `se` (smaller than or equal to):

```
se                  ::  Ord a => [a] -> [a] -> Bool
[]     `se` ys      =   True
(x:xs) `se` []      =   False
(x:xs) `se` (y:ys)  =   x<y || (x==y && xs`se`ys)
```

Now that the functions `eq` and `se` are defined, others comparison functions can easily be defined: `ne` (not equal), `ge` (greater than or equal to), `st` (smaller than) and `gt` (greater than):

```
xs `ne` ys  =  not (xs `eq` ys)
xs `ge` ys  =  ys `se` xs
xs `st` ys  =  xs `se` ys && xs `ne` ys
xs `gt` ys  =  ys `st` xs
```

These functions could of course be defined directly using recursion. In the prelude these functions are not defined; instead the usual comparison operators (`<=` etc.) are made to work on lists as well.

Joining lists

Two lists with the same type can be joined to form one list using the operator `++`. This process is also called *concatenation* ('chaining together'). E.g.: `[1,2,3]++[4,5]` results in the list `[1,2,3,4,5]`. Concatenating with the empty list (at the front or at the back) leaves a list unaltered: `[1,2]++[]` gives `[1,2]`.

The operator `++` is a standard function, but can be easily defined in Gofer (which happens in the prelude). So it is not a built-in operator like `:`. The definition reads:

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

In the definition the left parameter is subject to pattern matching. In the non-empty case the operator is called recursively with the shorter list `xs` as an parameter.

There is another function that joins lists. This function, `concat`, acts on a *list* of lists. All lists in the list of lists are joined to form one long list. For example

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

The definition of `concat` is as follows:

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

The first pattern, `[]`, is the empty list; an empty list of lists in this case. The result is an empty list: a list without any elements. In the second case of the definition the list of lists is not empty, so there is a list, `xs`, in front of a rest list of lists, `xss`. First all the rest lists are joined by the recursive call of `concat`; finally the first list `xs` is put in front of that as well.

Beware of the difference between `++` and `concat`: the operator `++` acts on *two* lists, the function `concat` on *a list* of lists. Both are popularly called 'concatenation'. (Compare with the situation of the operator `&&`, that checks whether two Booleans are `True` and the function `and` that checks whether a whole list of Booleans only contains `True` elements).

Selecting parts of lists

In the prelude a number of functions are defined that select parts of a list. With some functions the result is a (shorter) list, with others only one elemen.

As a list is built from a head and a tail, it is easy to retrieve the head and the tail again:

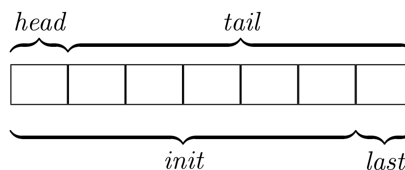
```
head         ::   [a] -> a
head (x:xs)   =   x
tail         ::   [a] -> [a]
tail (x:xs)   =   xs
```

These functions perform pattern matching on their parameters, but there is no separate definition for the pattern `[]`. If these functions are used on an empty list, an error message is shown.

It is not so easy to write a function that selects the *last* element from a list. For that you need recursion:

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

Again this function is undefined for the empty list, because that case is not covered by the two patterns. Just as `head` goes with `tail`, `last` goes with `init`. Here is an overview of these four functions:



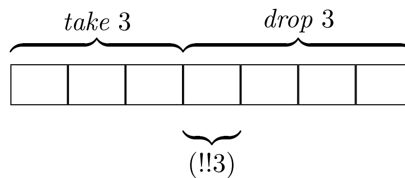
The function `init` selects everything *but* the last element. Therefore you need recursion again:

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

The pattern `x:[]` can (and is) usually written as `[x]`.

In section 2.4.1 a function `take` was presented. Apart from a list `take` has an integer as an parameter, which denotes how many elements of the list must be part of the result. The counterpart of `take` is `drop` that deletes a number of elements from the beginning of the list. Finally there is an operator `!!` that select one specific element from the list. Schematic:

page 30



These functions are defined as follows:

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

Whenever a list is too short as much elements as possible are taken or left out respectively. This follows from the second line in the definitions: if you give the function an empty list, the result is always an empty list, whatever the number is. If these lines were left out of the definitions, then `take` and `drop` would be undefined for lists that are too short.

The operator `!!` select one element from a list. The head of the list is numbered 'zero' and so `xs!!3` delivers the *fourth* element of the list `xs`. This operator can not be applied to too short a list; there is no reasonable value in that case. The definition reads:

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

For high numbers this function costs some time: the list has to be traversed from the beginning. So it should be used economically. The operator is suited to fetch one element from a list. The function `dayofweek` from section 2.4.1 could have been defined this way:

page 29

```
dayofweek d  =  [ "Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday" ] !! d
```

However, if all elements of the lists are used successively, it's better to use `map` or `foldr`.

Reversing lists

The function `reverse` from the prelude reverses the elements of a list. The function can easily be defined recursively. A reversed empty list is still an empty list. In case of a non-empty list the tail should be reversed and the head should be appended to the end of that. The definition could be like this:

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

Properties of lists

An important property of a list is its length. The length can be computed using the function `length`. In the prelude this function is defined as follows:

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

Furthermore, the prelude provides function `elem` that tests whether a certain element is contained in a list. That function `elem` can be defined as follows:

```
elem      ::  a -> [a] -> Bool
elem e xs  =  or (map (==e) xs)
```

The function compares all elements of `xs` with `e` (partial parametrization of the operator `==`). That results in a list of Booleans of which `or` checks whether there is at least one equal to `True`. By the utilization of the function composition operator the function can also be written like this:

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

The function `notElem` on the other hand checks whether an element is not contained in a list:

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

It can also be defined by

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

3.1.3 Higher order functions on lists

Functions can be made more flexible by giving them a function as a parameter. A lot of functions on lists have a function as an parameter. Therefore they are higher order functions.

map, filter en foldr

Previously `map`, `filter` and `foldr` were discussed. These function do something, depending on their function parameter, with every element of a list. The function `map` applies its function parameter to each element of the list:

$$\begin{array}{rcccccccccccc} \texttt{xs} & = & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\ & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ \texttt{map square xs} & = & [& 1 & , & 4 & , & 9 & , & 16 & , & 25 &] \end{array}$$

The `filter` function eliminates elements from a list that do not satisfy a certain Boolean predicate:

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

The `foldr` function inserts an operator between all elements of a list starting at the right hand with a given value:

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

These three standard functions are defined recursively in the prelude. They were discussed earlier in section 2.3.1. page 26

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

By using these standard functions extensively the recursion in other functions can be hidden. The 'dirty work' is then dealt with by the standard functions and the other functions look neater. The function `or`, which checks whether a list of Booleans contains at least one value `True`, is, for example, defined in this way:

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

But is also possible to write this function directly with recursion without the use of `foldr`:

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

A lot of functions can be written as a combination of a call to `foldr` and to `map`. A good example is the function `elem` from the last paragraph:

```
elem e   =  foldr (||) False  .  map (==e)
```

Of course, this function can be defined directly as well, without use of standard functions. Then recursion is necessary again:

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

takewhile and dropwhile

A variant of the `filter` function is the function `takeWhile`. This function has, just like `filter`, a predicate (function with a Boolean result) and a list as parameters. The difference is that `filter` always looks at all elements of the list. The function `takeWhile` starts at the beginning of the list and stops searching as soon as an element is found that doesn't satisfy the given predicate. For example: `takeWhile even [2,4,6,7,8,9]` gives `[2,4,6]`. Different from `filter` the 8 doesn't appear in the result, because the 7 makes `takeWhile` stop searching. The prelude definition reads:

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

Compare this definition to that of `filter`.

Like `take` goes with a function `drop`, `takeWhile` goes with a function `dropWhile`. This leaves out the beginning of a list that satisfies a certain property. For example: `dropWhile even [2,4,6,7,8,9]` equals `[7,8,9]`. Its definition reads:

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

foldl

The function `foldr` puts an operator between all elements of a list and starts with this at the end of the list. The function `foldr` does the same thing, but starts at the beginning of the list . Just as `foldr` `foldl` has an extra parameter that represents the result for the empty list.

Here is an example of `foldl` on a list with five elements:

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

In order to write down a definition for this function, it's convenient to give two examples first:

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

From these examples it becomes evident that the call to `foldl` with the list `x:xs` (with `xs`$=[y,z]$ in the example) is the same as `foldl xs` provided that in the recursive call the starting value is $a \oplus$ `x` instead of a. Having observed this, the definition can be written like this:

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

In the case of associative operators like `+` it doesn't matter that much whether you use `foldr` or `foldl`. Of course, for non-associative operators like `-` the result depends on which function you use.

3.1.4 Sorting lists

All functions on lists discussed up to now are fairly simple: in order to determine the result the lists is traversed once using recursion.

A function that can not be written in this manner is the sorting (putting the elements in ascending order) of a list. For the elements should be completely shuffled in order to accomplish this.

However, using the standard functions it is not that difficult to write a sorting function. There are different approaches to solve the sorting problem. In other words, there are different *algorithms*. Two algorithms will be discussed here. In both algorithms it is required that the elements can be ordered. So it possible to sort a list of integers or a list of lists of integers, but not a list of functions. This fact is expressed by the type of the sorting function:

```
sort  ::  Ord a  =>  [a] -> [a]
```

This means: `sort` acts on lists of arbitrary type `a` provided that `a` is contained in the class `Ord` of types that can be ordered.

Insertion sort

Suppose a sorted list given. Then a new element can be inserted in the right place using the following function:

```
insert             ::  Ord a  =>  a -> [a] -> [a]
insert e []         =  [e]
insert e (x:xs)
      |  e<=x       =  e : x : xs
      |  otherwise  =  x : insert e xs
```

If the list is empty, the new element `e` becomes the only element. If the list is not empty and has `x` as its first element, then it depends on whether `e` is smaller than `x`. If this is the case, `e` is put in front of the list; otherwise, `x` is put in front and `e` must be inserted in the rest of the list. An example of the use of `insert`:

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

For the operation of `insert` the parameter list has to be sorted; only then the result is sorted, too.

The function `insert` can be used to sort a list that is not already sorted. Suppose `[a,b,c,d]` has to be sorted. You can take an empty list (which is sorted) and insert the last element `d`. The result is a sorted list in which `c` can be inserted. The result stays sorted after you insert `b`. Finally `a` can be inserted in the right place and the final result is a sorted version of `[a,b,c,d]`. The expression being computed is:

```
a `insert` (b `insert` (c `insert` (d `insert` [])))
```

The structure of this computation is exactly that of `foldr` with `insert` as the operator and `[]` as starting value. Therefore one possible sorting algorithms is:

```
isort  =  foldr insert []
```

with the function `insert` as defined above. This algorithm is called *insertion sort*.

Merge sort

Another sorting algorithm makes use of the possibility to merge two sorted lists into one. This is what the function `merge` does:

```
merge                ::  Ord a => [a] -> [a] -> [a]
merge []     ys       =  ys
merge xs     []       =  xs
merge (x:xs) (y:ys)
       |  x <= y      =  x : merge xs (y:ys)
       |  otherwise   =  y : merge (x:xs) ys
```

If either one of the lists is empty, the other list is the result. If both lists are non-empty, then the smallest of the two head elements is the head of the result and the remaining elements are merged by a recursive call to `merge`.

Just like `insert`, `merge` supposes that the parameters are sorted. In that case it makes sure that also the result is a sorted list.

On top of the `merge` function you can build a sorting algorithm, too. This algorithm takes advantage of the fact that the empty list and singleton lists (lists with one element) are always sorted. Longer lists can (approximately) be split in two pieces. The two halves can be sorted by recursive calls to the sorting algorithm. Finally the two sorted results can be merged by `merge`.

```
msort           ::  Ord a => [a] -> [a]
msort xs
   |  len<=1     =   xs
   |  otherwise  =  merge (msort ys) (msort zs)
   where ys      =  take half xs
         zs      =  drop half xs
         half    =  len / 2
         len     =  len xs
```

This algorithm is called *merge sort*. In the prelude `insert` and `merge` are defined and a function sort that works like `isort`.

3.2 Special lists

3.2.1 Strings

In an example in section 2.4.1 texts were used as values, e.g. `"Monday"`. A text that is used as a value in a program is called a *string*. A string is a list of which the elements are characters. page 29

Thus all functions that operate on lists can also be used on strings. For example the expression A`"sun"++"day"` results in the string `"Sunday"`, and the result of the expression `tail (take 3 "gofer")` is the string `"of"`.

Strings are denoted between double quotes. The quotes indicate that the text should be taken literally as the value of a string and not as the name of a function. So `"until"` is a string containing five characters, but `until` is the name of a function. Therefore a string must always be surrounded by double quotes. In the result of an expression they are left out by the interpreter:

```
? "Sun"++"day"
Sunday
```

The elements of a string are of type `Char`. That is an abbreviation of the word *character*. Possible characters are not only characters, but also digits and punctuation marks. The type `Char` is one of the four built-in types of Gofer (the other three are `Int`, `Float` and `Bool`).

Values of the type `Char` can be denoted by a character between *single quotes*, e.g. `'B'` or `'*'`. Note the difference with back quotes, which are used to make an operator out of a function.

The three expressions below have very different meanings:

`"f"`	a list of characters (string) that contains one element;
`'f'`	a character;
`` `f` ``	a function `f` viewed as an operator.

The notation with double quotes for strings is merely an abbreviation of an enumeration of a list of characters. The string `"hello"` means the same as the list `['h','a','l','l','o']` or `'h':'e':'l':'l':'o':[]`.

Examples from which it becomes evident that a string is in fact a list of characters, are the expression `hd "bike"` which delivers the character `'b'` and the expression `takeWhile (=='e') "aardvark"` which delivers the string `"aa"`.

3.2.2 Characters

The value of a `Char` can be characters, digits and punctuation marks. It is important to put quotes around a character, because otherwise these symbols mean something else in Gofer:

expression	type	meaning
`'x'`	`Char`	the character `'x'`
`x`	...	the name of e.g. a parameter
`'3'`	`Char`	the digit character `'3'`
`3`	`Int`	the number 3
`'.'`	`Char`	the punctuation mark period
`.`	`(b->c)->(a->b)->a->c`	the function composition operator

There are 128 possible values for the type `Char`:
- 52 characters
- 10 digits
- 32 punctuation marks and the space
- 33 special symbols
- (128 extra symbols: characters with accents, more punctuation marks etc.)

There is a symbol that causes trouble in a string: the double quote. Usually a double quote would mean the end of the string. If you really need a double quote in a string, you have to put the symbol \ (*backslash*) in front. For example:

```
"He said \"hello\" and walked on"
```

This solution brings about a new problem, because now the symbol \ can not be used in a string. So if this symbol should appear in a string, it has to be doubled:

```
"the symbol \\ is called backslash"
```

Such a double symbol counts as one character. And so the length of the string `"\"\"\\\""` is equal to 4. Also, these symbols may appear between single quotes, like in the definitions below:

```
colon        = ':'
doublequote  = '\"'
backslash    = '\\'
apostrophe   = '\''
```

The 33 special characters are used to influence the layout of a text. The most important special characters are the 'newline' and the 'tab'. These characters can also be denoted using the backslash: `'\n'` is the newline character and `'\t'` is the tab character. The newline character can be used to produce a result consisting of more than one line:

```
? "ONE\nTWO\nTHREE"
ONE
TWO
THREE
```

All characters are numbered according to a coding devised by the International Standards Organization (ISO) [1]. There are two (built-in) standard functions that determine the code of a character and deliver the character with a given code, respectively:

```
ord  ::  Char -> Int
chr  ::  Int  -> Char
```

For example:

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

An overview of all characters with their ISO/ASCII numbers is in appendix B. The characters are ordered conforming this coding. By that the type `Char` is part of the class `Ord`. With respect to the characters, the ordering respects the alphabetical ordering on the understanding that the capitals precede the lower case characters. This ordering also carries over into strings; strings are namely lists and these are ordered lexicographically based on the ordering of their elements:

```
? sort ["cow", "bike", "Kim", "Peter"]
["Kim", "Peter", "bike", "cow"]
```

3.2.3 Functions on characters and strings

In the prelude some functions on characters are defined with which you can determine what kind of character a given symbol is:

```
isSpace, isUpper, isLower, isAlpha, isDigit, isAlnum :: Char->Bool
isSpace c     =  c == ' '    ||  c == '\t'  || c == '\n'
isUpper c     =  c >= 'A'    &&  c <= 'Z'
isLower c     =  c >= 'a'    &&  c <= 'z'
isAlpha c     =  isUpper c   ||  isLower c
isDigit c     =  c >= '0'    &&  c <= '9'
isAlphanum c  =  isAlpha c   ||  isDigit c
```

During the definition of a function these functions come in quite useful to separate different cases.

In the ISO coding the code of the digit `'3'` is not 3, but 51. Luckily the digits *are* consecutive in the coding. To retrieve the numeric value of a digit symbol not only the function `ord` has to be

[1] This coding is often referred to as the ASCII coding (American Standard Code for Information Interchange). Nowadays the coding is internationally approved and should be called the ISO coding.

applied, but also 48 has to be subtracted from the result. This is what the function `digitValue` does:

```
digitValue   ::  Char -> Int
digitValue c  =  ord c - ord '0'
```

This function can eventually be secured against 'unauthorized' use by demanding that the parameter is in fact a digit:

```
digitValue c  |  isDigit c  =  ord c - ord '0'
```

The reverse operation is provided by the function `digitChar`: this functions turns an integer (in the range from 0 to 9) into the corresponding digit character:

```
digitChar   ::  Int -> Char
digitChar n  =  chr ( n + ord '0')
```

These two functions are not included in the prelude (but if they are needed, you can of course define them yourself).

The prelude does contain two functions to convert lower case characters to upper case and the other way around:

```
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
```

With the help of `map` these functions can be applied to every element of a string:

```
? map toUpper "Hello!"
HELLO!
? map toLower "Hello!"
hello!
```

Every polymorphic function on lists can also be used on strings. Beside that there are a few functions in the prelude that act on strings specifically:

```
words,   lines   ::   [Char]  -> [[Char]]
unwords, unlines ::  [[Char]] ->  [Char]
```

The function `words` splits a string into a number of smaller strings of which each contains one word of the input string. Words are separated by spaces (like in `isSpace`). The function `lines` does the same thing, but then for the separate lines in the input string separated by newline characters (`'\n'`). Examples:

```
? words "this is a string"
["this", "is", "a", "string"]
? lines "first line\second line"
["first line", "second line"]
```

The functions `unwords` and `unlines` behave in the opposite manner: they join a list of words (lines respectively) into one long string:

```
? unwords ["these", "are", "the", "words"]
these are the words
? unlines ["first line", "second line"]
first line
second line
```

Note that there are no quotes in the result: these are always left out if the result of an expression is a string.

A variant of the function `unlines` is the function `layn`. This function numbers the lines in the result:

```
? layn ["first line", "second line"]
   1) first line
   2) second line
```

You can look up the definitions of these functions in the prelude; at this moment it is only important that you can use them in expressions to get a readable outcome.

3.2.4 Infinite lists

The number of elements in a list can be infinite. The following function `from` delivers an infinitely long list:

```
from n  =  n : from (n+1)
```

Of course, the computer can't hold an infinite number of elements. Fortunately you can already look at the beginning of the list, while the rest of the list is still to be built:

```
? from 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, control-C
{Interrupted!}
?
```

Once you have seen enough elements you can interrupt the computation by pressing control-C.

An infinite list can also be used as an intermediate result, while at the same time the final result is finite. For example this is the case in the following problem: 'determine all powers of three smaller than 1000'. The first ten powers can be calculated using the following call:

```
? map (3^) [1..10]
[3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049]
```

The elements smaller than 1000 can be extracted by `takeWhile`:

```
? takeWhile (<1000) (map (3^) [1..10])
[3, 9, 27, 81, 243, 729]
```

But how do you know beforehand that 10 elements suffice ? The solution is to use the infinite list `from 1` instead of `[1..10]` and so compute *all* powers of three. That will certainly be enough...

```
? takeWhile (<1000) (map (3^) (from 1))
[3, 9, 27, 81, 243, 729]
```

This method can be applied thanks to the fact that the interpreter is rather lazy: work is always postponed as long as possible. That is why the outcome of `map (3^) (from 1)` is not computed fully (that would take an infinite amount of time). Instead only the first element is computed. This is passed on to the outer world, in this case the function `takeWhile`. Only when this element is processed and `takewhile` asks for another element, the second element is calculated. Sooner or later `takeWhile` will not ask for new elements to be computed (after the first number ≥ 1000 is reached). No further elements will be computed by `map`.

3.2.5 Lazy evaluation

The evaluation order (the way expressions are calculated) of Gofer is called *lazy evaluation*. With lazy evaluation an expression (or part of it) is only then computed when it is certain that its value is *really* needed for the result. The opposite of lazy evaluation is *eager evaluation*. With eager evaluation the function result is computed as soon as the actual parameter is known.

The use infinite lists is possible thanks to lazy evaluation. In languages that use eager evaluation (like all imperative languages and some older functional languages) infinite lists are not possible.

Lazy evaluation has a number of other advantages. For example, take a look at the function `prime` from section 2.4.1 that tests whether a number is prime: page 29

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

Would this function determine *all* divisors of `x` and then compare that list to `[1,x]`? No, that would be too much work! At the call `prime 30` the following happens. Firstly, the first divisor of 30 is determined: 1. This value is compared with the first element of the list `[1,30]`. Regarding the first element the lists are equal. Then the second divisor of 30 is determined: 2. This number is compared with the second value of `[1,30]`: the second elements of the lists are not equal. The operator `==` 'knows' that two lists can never be equal again as soon as two different elements are encountered. Therefore `False` can be returned immediately. The other divisors of 30 are never computed!

The lazy behavior of the operator == is caused by its definition. The recursive line from the definition in section 3.1.2 reads: page 39

```
(x:xs) == (y:ys)  =  x==y && xs==ys
```

If x==y delivers the value False, there is no need to compute xs==ys: the final result will always be False. This lazy behavior of the operator && depends in its turn on his definition:

```
False && x  =  False
True  && x  =  x
```

If the left parameter is False, the value of the right parameter is not needed in the computation of the result. (This is the real definition of &&. The definition in section 1.4.3 is also correct, but doesn't exhibit the desired lazy behavior). page 11

Functions that need all elements of a list, can not be used on infinite lists. Examples of such functions are sum and length.

At the call sum (from 1) or length (from 1) even lazy evaluation doesn't help to compute the answer in finite time. In that case the computer will go into trance and will never deliver a final answer (unless the result of the computation isn't used anywhere, for then the computation is of course never performed...).

3.2.6 Functions on infinite lists

In the prelude some functions are defined that result in infinite lists.

The real name of the function from from section 3.2.4 is enumFrom. This function is normally not used in this form, because instead of writing enumFrom n it is allowed to write [n..]. (Compare this notation to [n..m] for enumFromTo n m, discussed in section 3.1.1). page 48 page 38

An infinite list which only contains repetitions of one element can be generated using the function repeat:

```
repeat   ::  a -> [a]
repeat x  =  x : repeat x
```

The call repeat 't' gives the infinite list "ttttttt....

An infinite list generated by repeat can be used as an intermediate result by a function that does have a finite result. For example, the function copy makes a finite number of copies of an element:

```
copy     ::  Int -> a -> [a]
copy n x  =  take n (repeat x)
```

Thanks to lazy evaluation copy can use the infinite result of repeat. The functions repeat and copy are defined in the prelude.

The most flexible function is again a higher order function, which means a function with a function as an parameter. The function iterate has a function and a starting element as parameters. The result is an infinite list in which every element is obtained by applying the function to the previous element. For example:

```
iterate (+1) 3       is  [3, 4, 5, 6, 7, 8, ...
iterate (*2) 1       is  [1, 2, 4, 8, 16, 32, ...
iterate (/10) 5678   is  [5678, 567, 56, 5, 0, 0, ...
```

The definition of iterate, which is in the prelude, reads as follows:

```
iterate     ::  (a->a) -> a -> [a]
iterate f x  =  x : iterate f (f x)
```

This function resembles the function until defined in section 2.3.2. until also has a function and a starting element as parameters. The difference is that until stops as soon as the value satisfies a certain condition (which is also an parameter). Furthermore, until only delivers the last value (which satisfies the given condition), while iterate stores all intermediate results in a list. It has to, because there is no last element of an infinite list... page 26

Next two examples are discussed in which iterate is used to solve a practical problem: displaying a number as a string and generating the list of all prime numbers.

Displaying a number as a string

The function `intString` converts a number into a string that contains the digits of that number. For example: `intString 5678` gives the string `"5678"`. Thanks to this function you can combine the result of a computation with a string, for example as in `intString (3*17)++" lines"`.

The function `intString` can be constructed by executing a number of functions after each other. First the number should be repeatedly divided by 10 using `iterate` (just like in the third example of `iterate` above). The infinite tail of zeroes is not interesting and can be chopped off by `takeWhile`. Now the desired digits can be found as the last digits of the numbers in the list.

The function `intString` can be constructed by the execution of a number of functions after each other. Firstly, the number should be repeatedly divided by 10 using `iterate` (like in the third example of `iterate` above). The infinite tail of zeroes is not interesting and can be chopped off by `takeWhile`. Now the desired digits can be found as the last digits of the numbers in the list; the last digit of a number is equal to the remainder after division by 10. The digits are still in the wrong order, but that can be resolved by `reverse`. Finally the digits (of type `Int`) must be converted to the corresponding digit characters (of type `Char`).

An example clarifies this all:

```
           5678
             ↓ iterate (/10)
[5678, 567, 56, 5, 0, 0, ...
             ↓ takeWhile (/=0)
    [5678, 567, 56, 5]
             ↓ map (‘rem‘10)
       [8, 7, 6, 5]
             ↓ reverse
       [5, 6, 7, 8]
             ↓ map digitChar
   [’5’, ’6’, ’7’, ’8’]
```

The function `intString` can now be simply written as the composition of these five steps. Note that the functions are written down in reversed order, because the function composition operator (`.`) means ‘after’:

```
intString :: Int -> [Char]
intString  =  map digitChar
            . reverse
            . map (‘rem‘10)
            . takeWhile (/=0)
            . iterate (/10)
```

Functional programming is programming with functions!

The list of all prime numbers

In section 2.4.1 `prime` was defined that determines whether a number is prime. With that the (infinite) list of all prime numbers can be generated by page 29

```
filter prime [2..]
```

The `prime` function searches for the divisors of a number. If such a divisor is large, it takes long before the function decides a number is not a prime.

By making clever use of `iterate` a much faster algorithm is possible. This method also starts off with the infinite list `[2..]`:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
```

The first number, 2, can be stored in the list of primes. Then 2 and all its multiples are crossed out. What remains is:

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...
```

The first number, 3, is a prime number. This number and its multiples are deleted from the list:

```
[5, 7, 11, 13, 17, 19, 23, 25, 29, 31, ...
```

The same process is repeated, but now with 5:

```
[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...
```

And you go on and on. The function 'cross out multiples of first element' is always applied to the previous result. This is an application of `iterate` using `[2..]` as the starting value:

```
iterate crossout [2..]
where crossout (x:xs) = filter (not.multiple x) xs
      multiple x y  =  divisible y x
```

(The number `y` is a multiple of `x` if `y` is divisible by `x`). As the starting value is a infinite list. the result of this is an *infinite list of infinite lists*. That super list looks like this:

```
[ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
, [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...
, [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, ...
, [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, ...
, [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 51, 53, ...
, ...
```

You can never see this thing as a whole; if you try to evaluate it, you will only see the beginning of the first list. But you need the complete list to be visible: the desired prime numbers are the first elements of the lists. So the prime numbers can be determined by taking the `head` of each list:

```
primenums ::  [Int]
primenums  =  map head (iterate crossout [2..])
              where  crossout (x:xs) = filter (not.multiple x) xs
```

Thanks to lazy evaluation only that part of each list is calculated that is needed for the desired part of the answer. If you want to know the next prime, more elements of every list are calculated as far as necessary.

Often it is hard (as in this example) to imagine what is computed at what moment. But there is no need: while programming you can just pretend infinite lists really exist; the evaluation order is automatically optimized by lazy evaluation.

3.2.7 List comprehensions

In set theory a handy notation to define sets is in use:

$$V = \{\ x^2\ \mid\ x \in N, x \text{ even}\ \}$$

Analogously to this notation, the so-called set comprehension, in Gofer a comparable notation to construct lists is available. Consequently this notation is called a *list comprehension*. A simple example of this notation is the following expression:

```
[ x*x | x <- [1..10] ]
```

This expression can be read aloud as '`x` square for `x` from 1 to 10'. In a list comprehension there is an expression in front of a vertical bar; this expression can contain a variable. This variable (`x` in the example) is bound in the part after the vertical bar. The notation 'expx ¡- xs' means that `x` ranges over all values in the list `xs`'. For each of these values the value of the expression in front of the vertical bar is computed.

So the example above has the same value as

```
map square [1..10]
```

with the function `square` defined as

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

The advantage of the comprehension notation is the fact that the function that is to be computed every time (here `square`) doesn't need a name.

The list comprehension notation has more possibilities. After the vertical bar more than one variable may appear. Then the expression in front of the vertical bar is computed for every possible combination. For example:

```
? [ (x,y) | x<-[1..2], y<-[4..6] ]
[ (1,4), (1,5), (1,6), (2,4), (2,5), (2,6) ]
```

The last variable changes fastest: for every value of `x`, `y` traverses the list `[4..6]`.

Apart from definitions of ranging variables expressions with Boolean results are allowed after the vertical bar. The meaning of this is demonstrated by the next example:

```
? [ (x,y) | x<-[1..5], even x, y<-[1..x] ]
[ (2,1), (2,2), (4,1), (4,2), (4,3), (4,4) ]
```

In the resulting list only those values of `x` are stored for which `even x` is `True`. By every arrow (`<-`) a variable is defined that can be used in later expressions and in the expression to the left of the vertical bar. Therefore `x` can not only be used in `(x,y)` but also in `even x` and in `[1..x]`. However, `y` can only be used in `(x,y)`. The arrow is especially reserved for this purpose and thus is not an operator!

Strictly speaking the list comprehension notation is superfluous. You can reach the same effect by combinations of `map`, `filter` and `concat`. However, especially in difficult cases the comprehension notation is much easier to understand. Without it the example above should be written like

```
concat (map f (filter even [1..5]))
where  f x  =  map g [1..x]
               where g y  =  (x,y)
```

which is less intuitive.

The interpreter directly translates the list comprehension notation into the corresponding expression with `map`, `filter` and `concat`. Just like the interval notation the comprehension notation is purely for the programmer's convenience.

3.3 Tuples

3.3.1 Use of tuples

All elements in a list should be of the same type. It is not possible to store an integer as well as a string in a list. Still sometimes is necessary to group information of different types together. Information in a register consists for example of a string (name), a boolean (gender) and three integers (date of birth). These pieces of information belong together, but can not be stored in a list.

For such cases, there is another way besides lists to make composite types: *tuples*. A *tuple* consists of a static number of values that are grouped together. The values may be of different types (although that is not obligatory).

Tuples are denoted by round parentheses around the elements (where lists use square brackets). Examples of tuples are:

`(1, 'a')`	a tuple with as elements the integer 1 and the character `'a'`;
`("monkey", True, 2)`	a tuple with three elements: the string `"monkey"`, the boolean `True` and the number 2;
`( [1,2], sqrt)`	a tuple with two elements: the list of integers `[1,2]` and the function from float to float `sqrt`;
`(1, (2,3))`	a tuple with two elements: the number 1 and a tuple containing the numbers 2 and 3.

The tuple of each combination types is a distinct type. The order is important, too. The type of tuples is written by enumerating the types of the elements between parentheses. The four expressions above can be types as follows:

```
(1, 'a')             ::  (Int, Char)
("monkey", True, 2)  ::  ([Char], Bool, Int)
```

```
([1,2], sqrt)        ::  ([Int], Float->Float)
(1, (2,3))           ::  (Int, (Int,Int))
```

A tuple with two elements is called a 2-tuple or a *pair*. Tuples with three elements are called 3-tuples etc. There are no 1-tuples: the expression (7) is just an integer; for it is allowed to put parentheses around every expression. There is a 0-tuple: the value () that has type ().

The prelude provides a number of functions that operate on 2-tuples or 3-tuples. These are good examples of how you define functions on tuples: by pattern matching.

```
fst            ::  (a,b) -> a
fst (x,y)       =  x
snd            ::  (a,b) -> b
snd (x,y)       =  y
fst3           ::  (a,b,c) -> a
fst3 (x,y,z)   =  x
snd3           ::  (a,b,c) -> b
snd3 (x,y,z)   =  y
thd3           ::  (a,b,c) -> c
thd3 (x,y,z)   =  z
```

These functions are all polymorphic, but of course it is possible to write functions that only work on a specific type of tuple:

```
f         ::  (Int,Char) -> [Char]
f (n,c)   =  intString n ++ [c]
```

If two values of the same type are to be grouped, you can use a list. Sometimes a tuple is more appropriate. A point in the plane, for example, is described by two Float numbers. Such a point can be represented by a list or a 2-tuple. In both cases it possible to define functions working on points, e.g. 'distance to the origin'. The function distanceL is the list version, distanceT the tuple version:

```
distanceL      ::  [Float] -> Float
distanceL [x,y]  =  sqrt (x*x+y*y)
distanceT      ::  (Float,Float) -> Float
distanceT (x,y)  =  sqrt (x*x+y*y)
```

As long as the function is called correctly, there is no difference. But it could happen that due to a typing error or a logical error the function is called with three coordinates. In the case of distanceT this mistake is detected during the analysis of the program: a tuple with three numbers is of another type than a tuple with two numbers. However, using distanceL the program is well-typed. Only when the function is really used, it becomes evident that distanceL is undefined for a list of three elements. Here the use of tuples instead of lists helps to detect errors as soon as possible.

Another situation in which tuples come in handy are functions with multiple results. Functions with several parameters are possible thanks to the currying mechanism; functions with more than one result are only possible by 'wrapping' these results up in a tuple. Then the tuple as a whole is the only result.

An example of a function with two results is splitAt which is defined in the prelude. This function delivers the results of take and drop at the same time. Therefore the function could be defined as follows:

```
splitAt       ::  Int -> [a] -> ([a],[a])
splitAt n xs  =  (take n xs, drop n xs)
```

However, the work of both functions can be done simultaneously. That is why in view of efficiency considerations splitAt is really defined as:

```
splitAt              ::  Int -> [a] -> ([a], [a])
splitAt 0     xs      =  ([]   , xs)
splitAt n     []      =  ([]   , [])
splitAt (n+1) (x:xs) =  (x:ys, zs)
                     where  (ys,zs) = splitAt n xs
```

The call splitAt 2 "gofer " gives the 2-tuple ("go","fer"). In the definition (at the recursive call) you can see how you can use such a result tuple: by exposing it to a pattern match (here (ys,zs)).

3.3.2 Type definitions

When you make a lot of use of lists and tuples, type declarations tend to become rather complicated. For example in writing functions on points, like the function `distance` above. The simplest functions are still surveyable:

```
distance   :: (Float,Float) -> Float
difference :: (Float,Float) -> (Float,Float) -> Float
```

But it becomes harder when lists of points or higher order functions are involved:

```
area_polygon   :: [(Float,Float)] -> Float
transf_polygon :: ((Float,Float)->(Float,Float))
                   -> [(Float,Float)] -> [(Float,Float)]
```

In such a case *type definitions* are convenient. With a type definition you can give a (clearer) name to a type, e.g.:

```
type Point = (Float,Float)
```

After this type definition the type declarations become simpler:

```
distance        :: Point -> Float
difference      :: Point -> Point -> Float
area_polygon    :: [Point] -> Float
transf_polygon  :: (Point->Point) -> [Point] -> [Point]
```

It is even better to define 'polygon' using a type definition as well:

```
type Polygon    = [Point]
area_polygon    :: Polygon -> Float
transf_polygon  :: (Point->Point) -> Polygon -> Polygon
```

There are a few things to keep in mind when using type definitions:

- the word `type` is especially reserved for this purpose;
- the name of the newly defined type must start with an upper case character (it is a constant, not a variable);
- a type *declaration* specifies the type of a function; a type *definition* definition defines a new name for a type.

The interpreter regards the newly defined name purely as an abbreviation. When you ask the type of an expression `(Float,Float)` shows up again instead of `Point`. If there are two different names for one type, e.g.:

```
type Point    =  (Float,Float)
type Complex  =  (Float,Float)
```

then these are interchangeable. A `Point` is exactly the same as a `Complex` and `(Float, Float)`. In section 3.4.3 you can find how you define `Point` as a really *new* type. page 63

3.3.3 Rational numbers

An application in which tuples can be used is an implementation of the *Rational numbers*. The rational numbers form the mathematical set **Q**, numbers that can be written as a *ratio*. It is not possible to use `Float` numbers for calculations with ratios: the calculations must be *exact*, such that the outcome of $\frac{1}{2} + \frac{1}{3}$ is the fraction $\frac{5}{6}$ and not the `Float` 0.833333.

Fractions can be represented by a numerator and a denominator, both integer numbers. So the following type definition is obvious:

```
type Fraction =  (Int,Int)
```

Next a number of frequently used fractions get a special name:

```
qZero    =  (0, 1)
qOne     =  (1, 1)
qTwo     =  (2, 1)
qHalve   =  (1, 2)
qThird   =  (1, 3)
qQuarter =  (1, 4)
```

We want to write some functions that perform the most important arithmetical operations on fractions:

```
qMul ::  Fraction -> Fraction -> Fraction
qDiv ::  Fraction -> Fraction -> Fraction
qAdd ::  Fraction -> Fraction -> Fraction
qSub ::  Fraction -> Fraction -> Fraction
```

The problem is that one value can be represented by different fractions. For example, a 'half' can be represented by `(1,2)`, but also by `(2,4)` and `(17,34)`. Therefore the outcome of two times a quarter might 'differ' from 'half'. To solve this problem a function `simplify` is needed that can simplify a fraction. By applying this function after every operation on fractions, fractions will always be represented in the same way. The result of two times a quarter can then be safely compared to a half: the result is `True`.

The function `simplify` divides the numerator and the denominator by their *greatest common divisor*. The greatest common divisor (`gcd`) of two numbers is the greatest number by which both numbers are divisible. Beside it `simplify` makes sure that an eventual minus sign is always in the numerator of the fraction. Its definition reads as follows:

```
simplify (n,d)  =  ( (signum d * n)/g, abs d/g )
                     where  g  = gcd n d
```

A simple definition of `gcd x y` determines the greatest divisor of `x` that also divides `y` using `divisors` and `divisible` from section 2.4.1:

page 29

```
gcd x y  =  last (filter (divisible y') (divisors x'))
     where  x' = abs x
            y' = abs y
```

(The prelude contains a more efficient version of `gcd`:

```
gcd x y  =  gcd' (abs x) (abs y)
     where  gcd' x 0 = x
            gcd' x y = gcd' y (x `rem` y)
```

This algorithm is based on the fact that if `x` and `y` are divisible by `d` then so is ``x`rem`y`` (=`x-(x/y)*y`)).

Using `simplify` we are now in the position to define the mathematical operations. To multiply two fractions, the numerators and denominators must be multiplied ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Then the result can be simplified (to $\frac{5}{6}$):

```
qMul (x,y) (p,q)  =  simplify (x*p, y*q)
```

Dividing by a number is the same as multiplying by the inverse:

```
qDiv (x,y) (p,q)  =  simplify (x*q, y*p)
```

Before you can add two fractions, their denominators must be made the same first. ($\frac{1}{4} + \frac{3}{10} = \frac{10}{40} + \frac{12}{40} = \frac{22}{40}$). The product of the denominator can serve as the common denominator. Then the numerators must be multiplied by the denominator of the other fraction, after which they can be added. Finally the result must be simplified (to $\frac{11}{20}$).

```
qAdd (x,y) (p,q)  =  simplify (x*q+y*p, y*q)
qSub (x,y) (p,q)  =  simplify (x*q-y*p, y*q)
```

The result of computations with fractions is displayed as a tuple. If this is not nice enough, you can define a function `fractionString`:

```
fractionString :: Fraction -> String
fractionString (x,y)
          |  y'==1      =  intString x'
          |  otherwise =  intString x' ++ "/" ++ intString y'
                  where  (x',y') = simplify (x,y)
```

3.3.4 Tuples and lists

Tuples often appear as elements of a list. A list of two-tuples is used frequently for searching (dictionaries, telephone directories etc.). The search function can be easily written using patterns; for the list a 'non-empty list with as a first element a 2-tuple' is used.

```
search  ::  Eq a  =>  [(a,b)] -> a -> b
search ((x,y):ts) s
           | x == s     =  y
```

```
                  | otherwise  =  search ts s
```

The function is polymorphic, so that it works on lists of 2-tuples of arbitrary type. However, the elements should be comparable, which is why the type `a` should be in the `Eq` class. The element to be searched is intentionally defined as the second parameter, so that the function `search` can easily be partially parametrized with a specific search list, for example:

```
telephoneNr =  search telephoneDirectory
translation =  search dictionary
```

where `telephoneDirectory` and `dictionary` can be separately defined as constants.

Another function in which 2-tuples play a role is the `zip` function. This function is defined in the prelude. It has two lists as parameters that are chained together element-wise in the result. For example: `zip [1,2,3] "abc"` results in the list `[(1,'a'),(2,'b'),(3,'c')]`. If the two lists are not of equal length, the shortest determines the size of the result. The definition is rather straightforward:

```
zip               ::  [a] -> [b] -> [(a,b)]
zip []     ys      =  []
zip xs     []      =  []
zip (x:xs) (y:ys)  =  (x,y) : zip xs ys
```

The function is polymorphic and can thus be used on lists with elements of arbitrary type. The name *zip* reflects the fact that the lists are so to speak 'zipped'.

A higher order variant of `zip` is `zipWith`. Apart from the two lists this function also receives a function as a parameter that specifies how the corresponding elements should be combined:

```
zipWith                 ::  (a->b->c) -> [a] -> [b] -> [c]
zipWith f []     ys      =  []
zipWith f xs     []      =  []
zipWith f (x:xs) (y:ys)  =  f x y : zipWith f xs ys
```

This function applies a function (with two parameters) to all elements of the two lists. Besides `zip`, `zipWith` also resembles `map` which applies a function (with one parameter) to all elements of one list.

Given the function `zipWith`, `zip` can be defined as a partial parametrization of that function:

```
zip  =  zipWith makePair
 where  makePair x y = (x,y)
```

3.3.5 Tuples and Currying

With the aid of tuples you can write functions with more than one parameter without the use of the currying mechanism. For a function can have a tuple as its (only) parameter with which two values are smuggled in:

```
add (x,y)  =  x+y
```

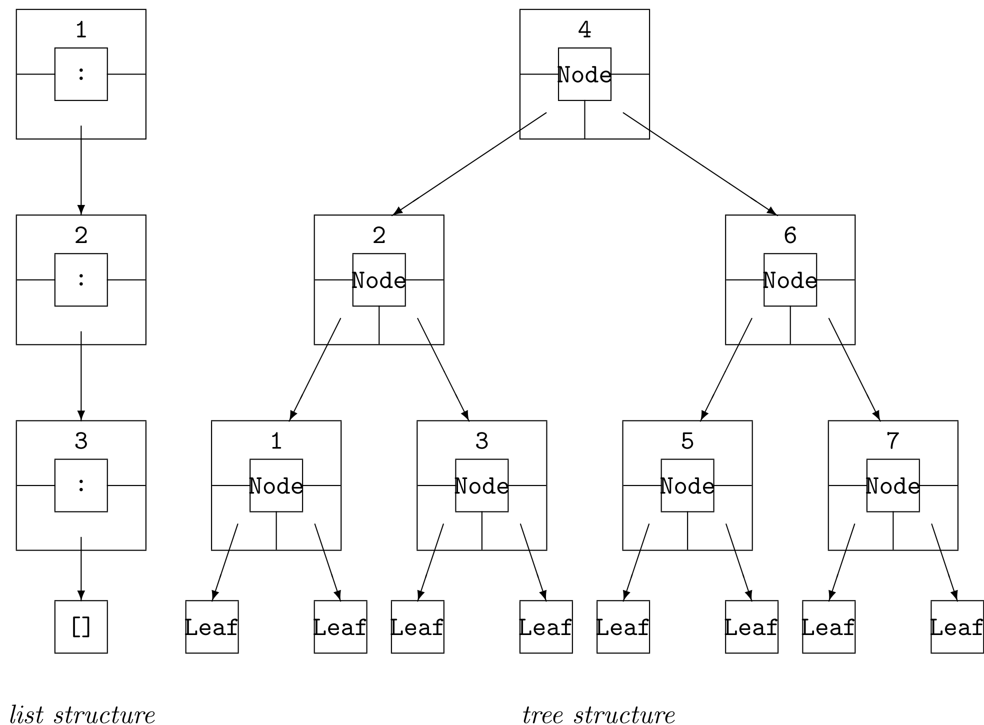
This function definition looks very classical. Most people would say that `add` is a function of two parameters and that the parameters 'naturally' are between parentheses. But in the mean time we know better: this function has one parameter, a tuple; the definition uses a pattern for tuples.

The currying mechanism, however, is preferable to the tupling method. Curried functions can be partially parametrized and functions with a tuple parameter not. Therefore all standard function with more than one parameter use the currying method.

In the prelude a function transformation (a function with a function as parameter and another function as result) is defined that converts a curried function into a function with a tuple parameter. This function is called `uncurry`:

```
uncurry  ::  (a->b->c) -> ((a,b)->c)
uncurry  f (a,b)  =  f a b
```

The other way round `curry` turns a function with a tuple parameter into a curried version. Thus `curry add` with `add` as defined above, *can* be partially parametrized.



list structure *tree structure*

3.4 Trees

3.4.1 Datatype definitions

Lists and tuples are two 'built-in' ways to structure information. When these two ways are not appropriate to represent some information, it is possible to define a new *datatype* yourself.

A datatype is a type that is determined by the way elements can be constructed. A 'list' is a datatype. Elements of a list can be built in two ways:

- as the empty list;
- by applying the operator `:` to an element and a (shorter) list.

In the patterns of functions on lists these two ways return, for example:

```
length []      =  0
length (x:xs)  =  1 + length xs
```

By defining the function for the patterns 'empty list' and 'operator `:` applied to an element and a list' the function is totally defined.

A list is a linear structure; as more elements are appended, the chain becomes longer. Sometimes such a linear structure is not appropriate and a *tree structure* would be better. There are different kinds of tree structures. In the figure a list is compared to a tree that has two branches at every internal node. In little squares it is showed how the structure is constructed. In the case of the list this is done with the use of the operator `:` with two parameters (drawn round it) or with `[]` without parameters. The tree is not built using operators but with functions: `Node` (three parameters) or `Leaf` (no parameters).

Functions you can use to construct a data structure are called *constructor functions*. The tree constructor functions are `Node` and `Leaf`. Names of constructors functions start with a capital, in order to distinguish them from 'normal' functions. Constructors functions written as operators must start with a colon. The constructor function `(:)` of lists is an example and the constructor `[]` is the only exception.

What constructor functions you can use for a new type is specified by a *data definition*. This definition also states the types of the parameters of the constructor functions and whether the new type is polymorphic. E.g. the data definition for trees as described above reads;

```
data Tree a  =  Node a (Tree a) (Tree a)
```

```
              |  Leaf
```

You can pronounce this definition as follows. 'A tree with elements of type a (shortly, a tree of a) can be built in two ways: (1) by applying the function Node to three parameters (one of type a and two of type tree of a), or (2) by using the constant Leaf.'

You can construct trees by using the constructor functions in an expression. The tree drawn in the figure is represented by the following expression:

```
Node 4 (Node 2 (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf)
       )
       (Node 6 (Node 5 Leaf Leaf)
               (Node 7 Leaf Leaf)
       )
```

You don't have to distribute it nicely over the lines; the following is also allowed:

```
Node 4(Node 2(Node 1 Leaf Leaf)(Node 3 Leaf Leaf))
                    (Node 6(Node 5 Leaf Leaf)(Node 7 Leaf Leaf))
```

However, the first definition is clearer. Also keep the layout rule from section 1.4.5 in mind. page 13

Functions on a tree can be defined by making a pattern for every constructor function. The next function, for example, computes the number of Node constructions in a tree:

```
size             ::  Tree a -> Int
size Leaf         =  0
size (Node x p q) =  1 + size p + size q
```

Compare this function to the function length on lists.

There are many more types of trees possible. A few examples:

- Trees in which the information is stored in the leaves (instead of in the nodes as in Tree):

```
data Tree2 a  =  Node2 (Tree2 a) (Tree2 a)
              |  Leaf2 a
```

- Trees in which information of type a is stored in the nodes and information of type b in the leaves:

```
data Tree3 a b  =  Node3 a (Tree3 a b) (Tree3 a b)
                |  Leaf3 b
```

- Trees that split in three branches at every node instead of two:

```
data Tree4 a  =  Node4 a (Tree4 a) (Tree4 a) (Tree4 a)
              |  Leaf4
```

- Trees in which the number of branches in a node is variable:

```
data Tree5 a  =  Node5 a [Tree5 a]
```

  In this tree you don't need a separate constructor for a 'leaf', because you can use a node with no outward branches.

- Trees in which every node only has one outward branch:

```
data Tree6 a  =  Node6 a (Tree6 a)
              |  Leaf6
```

  A 'tree' of this type is essentially a list: it has a linear structure.

- Trees with different kinds of nodes:

```
data Tree7 a b  =  Node7a Int a (Tree7 a b) (Tree7 a b)
                |  Node7b Char  (Tree7 a b)
                |  Leaf7a b
                |  Leaf7b Int
```

3.4.2 Search trees

A good example of a situation in which trees perform better than lists is searching (the presence of) an element in a large collection. You can use a *search tree* for this purpose.

In section 3.1.2 a function `elem` was defined that delivered `True` if an element was present in a list. page 41 Whether this function is defined using the standard functions `map` and `or`

```
elem      ::  Eq a  =>  a -> [a] -> Bool
elem e xs  =  or (map (==e) xs)
```

or directly with recursion

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

doesn't affect the efficiency that much. In both cases all elements of the list are inspected one by one. As soon as the element is found, the function immediately results in `True` (thanks to lazy evaluation), but if the element is not there the function has to examine all elements to reach that conclusion.

It is somewhat more convenient if the function can assume the list is sorted, i.e. the elements are in increasing order. The search can then be stopped when it has 'passed' the wanted element. A drawback is that the elements must not only be comparable (class `Eq`), but also orderable (class `Ord`):

```
elem'         ::  Ord a  =>  a -> [a] -> Bool
elem' e []               =  False
elem' e (x:xs)  |  e<x   =  False
                |  e==x  =  True
                |  e>x   =  elem' e xs
```

A much larger improvement can be achieved if the elements are not stored in a list, but in *search tree*. A search tree is a kind of 'sorted tree'. It is a tree built following the definition of `Tree` from the previous paragraph:

```
data Tree a  =  Node a (Tree a) (Tree a)
             |  Leaf
```

At every node an element is stored and two (smaller) trees: a 'left' subtree and a 'right' subtree (see the figure on page 57). Furthermore, in a search tree it is required that all values in the left subtree are *smaller* than the value in the node and all values in the right subtree *greater*. The values in the example tree in the figure are chosen so that it is in fact a search tree.

In a search tree the search for an element is very simple. If the value you are looking for is equal to the stored value in an node, you are done. If it is smaller you have to continue searching in the left subtree (the right subtree contains larger values). The other way around, if the value is larger you should look in the right subtree. Thus the function *elemTree* reads as follows:

```
elemTree   :: Ord a  =>  a -> Tree a -> Bool
elemTree e Leaf                    =  False
elemTree e (Node x li re) |  e==x  =  True
                          |  e<x   =  elemTree e li
                          |  e>x   =  elemTree e re
```

If the tree is well-balanced, i.e. it doesn't show big holes, the number of elements that has to be searched roughly halves at each step. And the demanded element is found quickly: a collection of thousand elements only has to be halved ten times and a collection of a million elements twenty times. Compare that to the half million steps `elem` costs on average on a collection of a million elements.

In general you can say the complete search of a collection of n elements costs n steps with `elem`, but only ${}^2\log n$ steps with `elemTree`.

Search trees are handy when a large collection has to be searched many times. Also e.g. `search` from section 3.3.4 can achieve enormous speed gains by using search trees. page 55

Structure of a search tree

The form of a search tree for a certain collection can be determined 'by hand'. Then the search tree can be typed in as one big expression with a lot of constructor functions. However, that is an annoying task that can easily be automated.

Like the function `insert` adds an element to a sorted list (see section 3.1.4), the function `insertTree` adds an element to a search tree. The result will again be a search tree, i.e. the element will be inserted in the right place:

page 43

```
insertTree  ::  Ord a  =>  a -> Tree a -> Tree a
insertTree e Leaf                       =  Node e Leaf Leaf
insertTree e (Node x le ri)  |  e<=x  =  Node x (insertTree e le) ri
                             |  e>x   =  Node x le (insertTree e ri)
```

If the element is added to a `Leaf` (an 'empty' tree), a small tree is built from `e` and two empty trees. Otherwise, the tree is not empty and contains a stored value `x`. This value is used to decide whether `e` should be inserted in the left or the right subtree.

By using the function `insertTree` repeatedly, all elements of a list can be put in a search tree:

```
listToTree ::  Ord a  =>  [a] -> Tree a
listToTree  =  foldr insertTree Leaf
```

Compare this function to `isort` in section 3.1.4.

page 43

A disadvantage of using `listToTree` is that the resulting search tree is not always well-balanced. This problem is not so obvious when information is added in random order. If, however, the list which is made into a tree is already sorted, the search tree 'grows cooked'.

```
? listToTree [1..7]
Node 7 (Node 6 (Node 5 (Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf)
Leaf) Leaf) Leaf) Leaf) Leaf) Leaf
```

Although this is a search tree (every value is between values in the left and right subtree) the structure is almost linear. Therefore logarithmic search times are not possible in this tree. A better (not 'linear') tree with the same values would be:

```
Node 4 (Node 2 (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf))
       (Node 6 (Node 5 Leaf Leaf)
               (Node 7 Leaf Leaf))
```

In chapter 8 an adjustment to `insertTree` is discussed that avoids crooked trees and so maintains logarithmic search times.

Sorting using search trees

The functions that are developed above can be used in a new sorting algorithm. For that one extra function is necessary: a function that puts the elements of a search tree in a list preserving the ordering. This function is defined as follows:

```
labels               ::  Tree a -> [a]
labels Leaf           =  []
labels (Node x li re) =  labels li ++ [x] ++ labels re
```

In contrast with `insertTree` this function performs a recursive call to the left *and* the right subtree. In this manner every element of the tree is inspected. As the value `x` is inserted in the right place, the result is a sorted list (provided that the parameter is a search tree).

An arbitrary list can now be sorted by making it into a search tree with `listToTree` and after that summing up the elements in the right order with `labels`:

```
sort  ::  Ord a  =>  [a] -> [a]
sort   =  labels . listToTree
```

Deleting from search trees

A search tree can be used as a database. Apart from the operations enumerate, insert and build, which are already written, a function for deleting elements comes in handy. This function somewhat resembles the function `insertTree`; depending on the stored value the function is called recursively on its left or right subtree.

```
deleteTree  ::  Ord a   =>  a -> Tree a -> Tree a
deleteTree e Leaf    =  Leaf
deleteTree e (Node x li re)
            | e<x   =  Node x (deleteTree e li) re
            | e==x  =  join li re
            | e>x   =  Node x li (deleteTree e re)
```

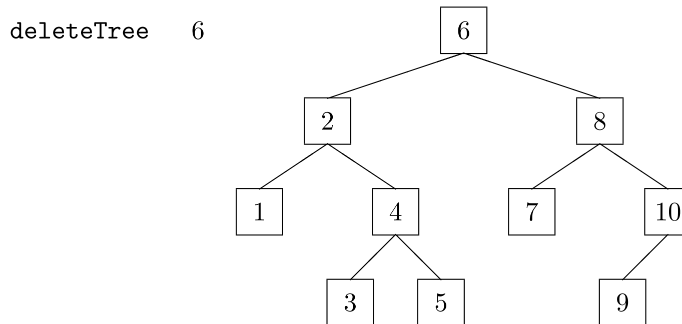
If, however, the value is found in the tree (the case `e==x`) it can't be left out just like that without leaving a 'hole'. That is why a function `join` that joins two search trees is necessary . This function takes the largest element from the left subtree as a new node. If the left subtree is empty, joining is of course no problem:

```
join  ::  Tree a -> Tree a -> Tree a
join Leaf b2  =  b2
join b1   b2  =  Node x b1' b2
                 where  (x,b1') = largest b1
```

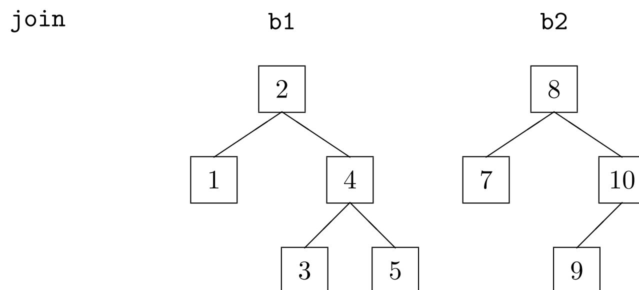
The function `largest`, apart from giving the largest element of a tree, also gives the tree that results after deleting that largest element. These two results are combined in a tuple. The largest element can be found by choosing the right subtree over and over again:

```
largest  ::  Tree a -> (a, Tree a)
largest (Node x b1 Leaf) =  (x, b1)
largest (Node x b1 b2)   =  (y, Node x b1 b2')
                     where  (y,b2') = largest b2
```

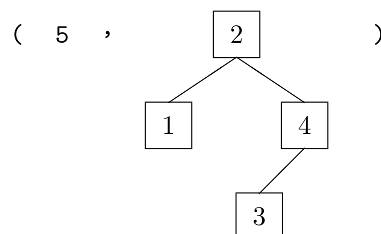
To demonstrate the actions of `deleteTree` we look at an example in which we represent the trees graphically for clearness' sake. At the call



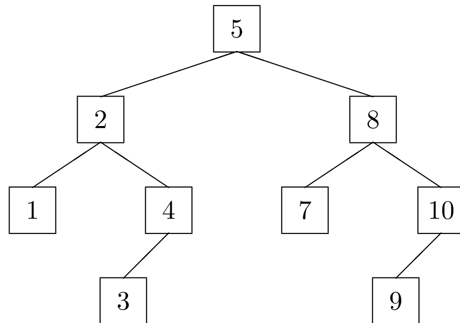
the function `join` is called with the left and right subtree as parameters:



The function `largest` is called by `join` with `b1` as the parameter. That results in a 2-tuple `(x, b1')`:



The trees `b1'` and `b2` are used as left and right subtree in the new search tree:



As the function `largest` is only called from `join` it doesn't have to be defined for a `Leaf`-tree. It is only applied on non-empty trees, because the empty tree is already treated separately in `join`.

3.4.3 Special uses of data definitions

Except for constructing trees of all kinds, data definitions can be used in some other remarkable ways. The datatype mechanism is so universally applicable that things can made for which you need separate constructs in other languages. Three examples of this are: finite types, the union of types and protected types.

Finite types

The constructor functions in a datatype definition are allowed to have no parameters. You could already see that: the constructor function `Leaf` of the type `Tree` had no parameters.

It is allowed that *no* constructor function has parameters. The result is a type that contains exactly as many elements as there are constructor functions: a finite type. The constructor functions serve as constants that point out the elements. An example:

```
data Direction  =  North | East | South | West
```

Functions operating on this type can be written using patterns, e.g.:

```
move             ::  Direction -> (Int,Int) -> (Int,Int)
move North (x,y)  =  (x,y+1)
move East  (x,y)  =  (x+1,y)
move South (x,y)  =  (x,y-1)
move West  (x,y)  =  (x-1,y)
```

The advantages of such a finite type over a coding with integers of characters are:

- function definitions are clear, because the names of the elements can be used, instead of obscure codings;
- the type system would complain if you accidentally add two direction (if the directions were coded as integers, no error message would be given and this could cause a lot of harm to be done)

Finite types are no new feature: in fact, the type `Bool` can be defined in this manner:

```
data Bool  =  False | True
```

This is why `False` and `True` must be written with a capital: they are the constructor functions of `Bool`. (By the way, this definition is not in the prelude. Booleans are not 'pre-defined' but 'built-in'. The reason is other built-in language constructs must already 'know' the Booleans, like case distinction with `|` in function definitions.)

Union of types

All elements of a list must have the same type. In a tuple values of different types can be stored, but the number of elements is constant. However, sometimes you want to make a list in which some elements are integer, others characters.

With a data definition you can make a type `IntOrChar` that has as its elements both integers and characters:

```
data IntOrChar  =  AnInt Int
                |  AChar Char
```

With this you can construct a 'mixed' list:

```
xs :: [IntOrChar]
xs  = [ AnInt 1, AChar 'a', AnInt 2, AnInt 3 ]
```

The only price you have to pay is that every element has to be tagged with the constructor function `AnInt` or `AChar`. This functions can be seen as conversion functions:

```
AnInt :: Int  -> IntOrChar
AChar :: Char -> IntOrChar
```

The behavior is comparable to that of built-in conversion functions like

```
round  :: Float -> Int
chr    :: Int   -> Char
```

Protected types

In section 3.3.2 a drawback of type definitions came forth: if two types are defined in the way, e.g. page 54

```
type Date     = (Int,Int)
type Fraction = (Int,Int)
```

they are interchangeable. Therefore 'Dates' can be used as if they were 'rational numbers' without the type checker complaining.

Using data definitions you can really make new types, so that for example a `Fraction` is not interchangable with any other `(Int,Int)` anymore. Instead of a type definition the following data definition is used:

```
data Fraction = Fract (Int,Int)
```

So there is only one constructor functions. To make a fraction with numerator 3 and denominator 5 it is not sufficient to write `(3,5)`, but you have to write `Rat (3,4)`. Just like with union types `Rat` can be seen as a conversion function from `(Int,Int)` to `Fraction`. You can just as well curry the constructor functions. In that case they don't have a tuple as parameter but two separate values. The corresponding datatype definition is:

```
data Fraction = Fract Int Int
```

This method is often used to make *protected types*. A protected type consists of a data definition and a number of functions that act on the defined type (in the case of `Fraction`, for example `qAdd`, `qSub`, `qMul` en `qDiv`).

The rest of the program (possibly written by another programmer) can use the type via the offered functions. It is, however, not allowed to make use of the way the type is constructed. You can achieve this by keeping the name of the constructor function 'a secret'. If later on the representation has to be changed for some reason, only the four basic functions have to written again; the rest of the program is guaranteed to work without modification.

The name of the constructor function is often chosen to be the same as the name of the type, e.g.

```
data Fraction = Fraction Int Int
```

There is no problem in this; for the interpreter there is no confusion possible (the word `Fraction` in a type, e.g. after `::`, represents a type; in an expression it is the constructor function).

Exercises

3.1 Verify that `[1,2]++[]` delivers `[1,2]` according to the definition of `++`. Hint: write `[1,2]` as `1:(2:[])`.

3.2 Write `concat` as a call to `foldr`.

3.3 Which of the following expressions gives `True` and which `False`:

```
[[]] ++ xs    ==  xs
[[]] ++ xs    ==  [xs]
[[]] ++ xs    ==  [ [], xs ]
[[]] ++ [xs]  ==  [ [], xs ]
[xs] ++ []    ==  [xs]
[xs] ++ [xs]  ==  [xs,xs]
```

3.4 The function `filter` can be defined in terms of `concat` and `map`:

```
filter p  =  concat . map box
      where  box x =
```

Complete the definition of the function `box` used here.

3.5 Write a non-recursive version of `repeat` using `iterate`.

3.6 Write a function with two lists as parameters that deletes the first occurrence of every element of the second list from the first list. (This function is in the prelude; it is called \\).

3.7 Use the functions `map` and `concat` instead of the list comprehension notation to define the following list:

```
[ (x,y+z) | x<-[1..10], y<-[1..x], z<-[1..y] ]
```

3.8 The simplification of fractions is not necessary if they are never compared to each other directly. Write a function `qEq` that can be used instead of `==`. This function delivers `True` if two fractions have the same value, even if the fractions are not simplified.

3.9 Write the four arithmetical functions for *complex numbers*. Complex numbers are numbers of the form $a + bi$ where a and b are real numbers and i is a 'number' with the property $i^2 = -1$. Hint: determine a formula for $\frac{1}{a+bi}$ by solving x and y from $(a{+}bi)*(x{+}yi) = (1{+}0i)$.

3.10 Write a function `stringInt` that turns a string into the corresponding integer. For example: `stringInt "123"` gives the value 123. Consider the string as a list of characters and determine what operator should be inserted between the elements. Should you start with that from the left or the right?

3.11 Define the function `curry` as the counterpart of `uncurry` from section 3.3.5. page 56

3.12 Write a search tree version of the function `search`, like `elemTree` is a search tree version of `elem`. Also give the type of this function.

3.13 The function `map` can be applied to functions. The result is again a function (with another type). There are no restrictions on the type of the functions which `map` is applied to. So you can apply it to the function `map` itself! What is the type of the expression `map map` ?

3.14 Give a definition of `until` that uses `iterate` and `dropWhile`.

3.15 Give a direct definition of the operator `<>` on lists. This definition should not make use operators like `<=` on lists. (If you want to test this definition using the Gofer interpreter, you should use a different name from `<`, because the operator `<` is already defined in the prelude.)

3.16 Write the function `length` as a call of `foldr`. (Clue: start on the right side with the number 0, and make that the operator which is applied to all elements of the list adds 1 to the intermediate result, regardless of the value of the list element.) What is the type of the function which is passed to `foldr`?

3.17 In section 3.1.4 two sorting methods were mentioned: the function `isort` based on `insert`, and the function `msort` based on `merge`. Another method works according to the next principle. Take a look at the first element of the list which is to be sorted. Next, take all elements which are smaller than this value. You are sure these values have to be stored *in front of* this element in the final result. Of course, they first have to be sorted (by a recursive call). The values in the list which are larger than the first element need to be appended (after being sorted recursively) behind this element. (This algorithm is known by the name *quicksort*). page 43
Write a function which operates according to this principle. Think of the base case yourself. What is the essential difference between this function and `msort`?

3.18 Look at the function `group` with the type

```
group :: Int -> [a] -> [[a]]
```

This function splits the given list in sub-lists (which are resulted in a list of lists), where the sub-lists have a given length. Only the last sub-list may be shorter, if necessary. An example application of the function is:

```
? group 3 [1..11]
[ [1,2,3], [4,5,6], [7,8,9], [10,11] ]
```

Write this function in the same way as the function `intString` in section 3.2.6, which means you will have to make a composition of a number of partial parametrizations of `iterate`, `takeWhile` and `map`. page 49

3.19 Given are trees of the next type:

```
data Tree2 a  =  Leaf2 a
              |  Node2 (Tree2 a) (Tree2 a)
```

Write a function `mapTree` and a function `foldTree` which operate on `Tree2`, like `map` and `foldr` do on lists. Also supply the type of these functions.

3.20 Write a function `depth`, which results the number of levels in a `Tree2`. Give a definition with induction, and an alternative definition where you use `mapTree` and `foldTree`.

3.21 Write a function `showTree`, which results an attractive representation in a string of a tree as defined on page 57. Each leaf must be put on a separate line (new line is expressed in Gofer by `"\n"`); leafs which are nested deeper need to be indented more than leafs on a lower level.

3.22 Suppose a tree b has depth n. What is the minimum and the maximum number of leafs b can have?

3.23 Write a function that given a sorted list delivers a search tree (i.e. a tree that gives a sorted list when you apply `labels` to it). Make sure the tree doesn't grow 'crooked', which is the case when you use `listToTree`.

Chapter 4

List Algorithms

4.1 Combinatorial Functions

4.1.1 Segments and sublists

Combinatorial functions operate on lists. They result a list of lists, where the specific characteristics of the elements are not used. The only thing combinatorial function can do, is remove elements, swap elements or count elements.

In these sections a number of combinatorial functions will be defined. Because they do not use the specific characteristics of their parameter-list, they are polymorphic functions:

```
inits, tails, segs ::        [a] -> [[a]]
subs, perms        ::        [a] -> [[a]]
combs              :: Int -> [a] -> [[a]]
```

To illuminate the effects of these functions the results of these functions applied to the list `[1,2,3,4]` is shown below:

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 more)		

The meaning of these six functions is probably already clear from the example:

- `inits` yields all *initial segments* of a list, meaning sequential segments of a list starting at the beginning. The empty list is also assumed to be a initial segment.
- `tails` yields all *tail segments* of a list: sequential fragments ending at the end. Here also the empty list is included.
- `segs` yields *all segments* of a list: initial segments and tail segments, but also sequential fragments from in between.
- `subs` yields al *subsequences* of a list. In contrast to segments these sequences need not to be sequential fragments of the original list. So there are more subsequences than segments.
- `perms` yields all *permutations* of a list. A permutation of a list contains the same elements, but this time in all possible orders.
- `combs n` yields all *combinations of* `n` *elements*, so all possible ways to choose `n` elements from a list. The order must be the same as in the original list.

These combinatorial functions can be defined recursively, so in the definition the cases `[]` and `(x:xs)` need to be handled separately. In the case of `(x:xs)` the function will be recursively called with the list `xs`.

There is a very simple way to find a clue for the definition of a function `f`. Just look at an example

to see what the result of `(x:xs)` is and identify the result of `f xs`. Next, try to find a way to combine `x` and `f xs` into the result of `f (x:xs)`

inits

With the description of the initial segments the empty list is also assumed to be an initial segment. Thus the *empty list* also has a initial segment: the empty list itself. De definition of `inits` for the case 'empty list' is therefore:

```
inits []   =  [ [] ]
```

For the case `(x:xs)` we look at the desired results of the list `[1,2,3,4]`.

```
inits [1,2,3,4]   =   [ [] , [1] , [1,2] , [1,2,3] , [1,2,3,4] ]
inits [2,3,4]     =   [ [] , [2] , [2,3] , [2,3,4] ]
```

From this it can be seen that the second until the fifth element of `inits [1,2,3,4]` are exactly the elements of `inits [2,3,4]`, only with an extra `1` in front. These four then only need to be completed with an empty list.

This mechanism is generally described in the second line of the definition of `inits`:

```
inits []      =  [ [] ]
inits (x:xs)  =  [] : map (x:) (inits xs)
```

tails

Just like with `inits` the empty list has one tail segment: the empty list. The result of `tails []` is therefore a list with only the empty list as an element.

For an indication for the definition of `tails (x:xs)` we first take a look at the example `[1,2,3,4]`:

```
tails [1,2,3,4]   =   [ [1,2,3,4] , [2,3,4] , [3,4] , [4] , [] ]
tails [2,3,4]     =   [ [2,3,4] , [3,4] , [4] , [] ]
```

With this function the second until the fifth element are exactly those of the recursive call. The only thing needed to be done is the addition of a first element ([1,2,3,4] in this example).

Using this concept the full definition can be written as:

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

The parentheses on the second line are essential: without them the typing would be incorrect due to the fact that `:` associates to the right.

segs

Again, the only segment of the empty list is the empty list. The result of `segs []` is, just as with `inits` and `tails`, a singleton-empty-list.

For a hint for the design of `segs (x:xs)`, we again apply the well-tried method:

```
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] ]
```

If put in the right order, it shows that the first seven elements of the desired result are exactly those of the recursive call. In the second part of the result (the lists starting with a 1) the initial segments of [1,2,3,4] can be identified (only the empty list is left out, since it is already in the result).

So the definition of `segs` can be assumed to be:

```
segs []      =  [ [] ]
segs (x:xs)  =  segs xs  ++  tail (inits (x:xs))
```

Another way of removing the empty list from the `inits` is:

```
segs []      =  [ [] ]
segs (x:xs)  =  segs xs  ++  map (x:) (inits xs)
```

subs

The empty list is the only subsequence of an empty list. In order to find the definition of `subs (x:xs)` we again examine the example:

```
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] , [] ]
```

The number of elements of `subs (x:xs)` (16 in this example) is exactly twice as big as the number of elements in the recursive call `subs xs`. The second half of the entire result is exactly equal to the result of the recursive call. In the first half these same 8 lists can be found, but this time preceded by a `1`.

So the definition can be formulated as:

```
subs []      =  [ [] ]
subs (x:xs)  =  map (x:) (subs xs)  ++  subs xs
```

The function is called recursively twice with the same parameter. This would be a waste of labour: it would be better to do the call just once, after which the result is used twice. This makes the algorithm a lot faster, since the function is called twice again in the recursive call of `subs xs`, and in that one again... Thus a much more efficient definition would be:

```
subs []      =  [ [] ]
subs (x:xs)  =  map (x:) subsxs  ++  subsxs
        where  subsxs = subs xs
```

4.1.2 Permutations and combinations

perms

A *permutation* of a list is a list with the same elements, but possibly in another order. The list of all permutations of a list can be defined very well using a recursive function.

The empty list has one permutation: the empty list. This is because it contains all the 0 elements and in the same order...

The more interesting case is of course the non-empty list `(x:xs)`. Again, we first examine the example:

```
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] ]
```

The number of permutations increases very fast: of a list with four elements there exist four times as many permutations as of those with three elements. In this example it is a little harder to recognize the result of the recursive call. This can only be done by grouping the 24 elements into 6 groups of 4. In every group there are lists with the same values as those of one list of the recursive call. The new element is inserted in all possible ways.

For example, the third group [[1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1]] always contains the elements [3,4,2], where the element 1 is added at the beginning, the second place, the third place, and at the end.

To insert an element in all possible ways in a list a helper-function can be written, which is also written recursively:

```
between           ::  a -> [a] -> [[a]]
between e []       =  [ [e] ]
between e (y:ys)   =  (e:y:ys)   :  map (y:) (between e ys)
```

In the definition of `perms (x:xs)` this function is partially parametrized with `x`, applied to all elements of the result of the recursive call. In the example this results in a list of six lists of four lists. However, the goal is there is resulted one long list of 24 lists. So, to the list of lists of lists `concat` must be applied, which does exactly that.

After all this the function `perms` can be defined as follows:

```
perms []      =  [ [] ]
perms (x:xs)  =  concat (map (between x) (perms xs))
         where  between e []     = [ [e] ]
                between e (y:ys) = (e:y:ys) : map (y:) (between e ys)
```

combs

As a last example of a combinatorial function the function `combs` is being treated. This function has, except for the list, also a number as a parameter:

```
combs :: Int -> [a] -> [[a]]
```

The objective is that the result of `combs n xs` contains all subsequences of `xs` with length `n`. So the function can be defined easily by

```
combs n xs  =  filter ok (subs xs)
        where  ok xs  =  length xs == n
```

However, this definition is not very efficient. Generally, the number of subsequences is quite large, so `subs` costs a lot of time, while most subsequences are thrown away by `filter`. A better definition can be obtained by defining `combs` directly, without using `subs`.

In the definition of `combs` the integer-parameter cases `0` and `n+1` are treated separately. In the case of `n+1` there are also two cases distinguished. So the definition would look like:

```
combs 0     xs     =  ...
combs (n+1) []     =  ...
combs (n+1) (x:xs) =  ...
```

The tree cases are treated below.

- To choose zero elements from a list there is only one possibility: the empty list. Therefore the result of `combs 0 xs` is the singleton-empty-list. It does not matter whether or not `xs` is empty.
- The pattern `n+1` means '1 or more'. It is impossible to choose at least one element from an empty list. Therefore the result of `combs (n+1) []` is the empty list: there is no solution. Not that there is an *empty list of solutions* and not *the empty list as the only solution* as in the previous case. An important difference!
- It is possible to choose `n+1` elements from the list of `x:xs`, provided that it is possible to choose `n` elements from `xs`. The solutions can be divided into two groups: lists containing `x`, and lists which do not contain `x`.
  - For lists which do contain `x`, there have to be chosen another `n` elements from `xs`. After that, `x` must be appended in front.
  - For lists not containing `x`, all `n+1` elements must be chosen from `xs`.

  In both cases `combs` can be called recursively. The results can be combined with `++`.

So the definition of `combs` will look like:

```
combs 0     xs     =  [ [] ]
combs (n+1) []     =  [ ]
combs (n+1) (x:xs) =  map (x:) (combs n xs)  ++  combs (n+1) xs
```

With this function the recursive calls cannot be combined, in contrast to `subs`, since the two calls have different parameters.

4.1.3 The @-notation

The definition of `tails` is a little cumbersome:

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

In the second line the parameter-list of the pattern is split into a head `x` and a tail `xs`. The tail is used with the recursive call, but the head and tail are also concatenated together again to the list `(x:xs)`. This is a waste of labour, because this list is also passed as a parameter.

Another definition of `tails` could be:

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

Now it is not necessary to rebuild the parameter list, because it is not split in two. But now, the function `tail` must be used explicitly in the recursive call. The nice thing about patterns was that that is not necessary anymore.

It would be ideal to unify the advantages of the two definitions. The parameter must be available both as a whole and as a head and tail. The name and the pattern are separated by the symbol `@`.

With the use of this feature the definition of `tails` becomes:

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

Although the symbol `@` may be used in operator-symbols, a stand-alone `@` is reserved for this feature.

With function already defined in this reader an @-pattern is very handy. For example in the function `dropWhile`:

```
dropWhile p []       = []
dropWhile p ys@(x:xs)
         | p x       = dropWhile p xs
         | otherwise = ys
```

This is the way `dropWhile` is really defined in the prelude.

4.2 Matrix calculus

4.2.1 Vectors and matrices

Matrix calculation is the part of mathematics that deals with linear images in multi-dimensional spaces. In this section the most important concepts of matrix calculus will be introduced. Furthermore it will be indicated how these concepts can be modeled as type or function in Gofer. The Gofer definition of the functions will be shown in the next to sections.

The generalization of a one-dimensional line, the two-dimensional plane and the three-dimensional space is the n-dimensional space.'indexspace, n-dimensional In an n-dimensional space every ‘point‘ can be identified by n numbers. Such an identification is called a *vector*. In Gofer a vector could be defined as an element of the type

```
type Vector  =  [Float]
```

The number of elements of a list determines the dimension of the space. To prevent confusion with other lists of floats, which are not meant to be vectors, it would be better to define a ‘protected type’ like described in section 3.4.3:

page 63

```
data Vector  =  Vec [Float]
```

To covert a list of numbers into a `vector` the constructor function `Vec` must be applied.

In stead of a *point* in the (n-dimensional) space it is also possible to interpret a vector as a *(directed) line segment* from the origin to that point. A useful function with the manipulation of vectors is the determination of the distance of a point to the origin,

```
vecLength    :: Vector -> Float
```

Of two vectors (in the same space) can be determined whether they are *perpendicular*, and more generally what *angle* they have with respect to each other:

```
vecPerpendicular :: Vector -> Vector -> Bool
vecAngle         :: Vector -> Vector -> Float
```

A vector can also be ‘multiplied’ with a number by multiplying all numbers in the list (all coordinates) by that number. Two vectors can be ‘added’ by adding all the coordinates:

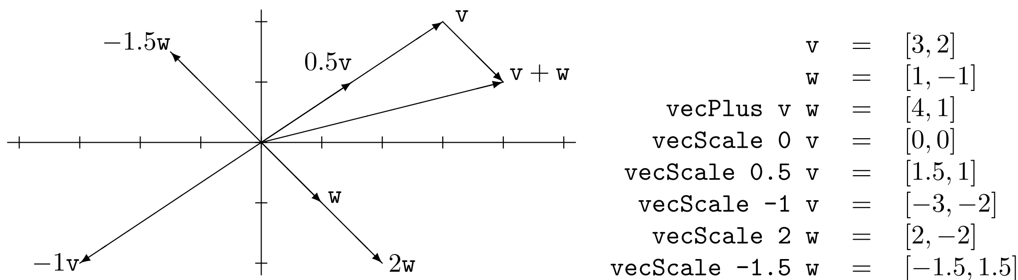
```
vecScale     :: Float  -> Vector -> Vector
vecPlus      :: Vector -> Vector -> Vector
```

In the next section these functions will be written. In the directed line segment interpretation of vectors these functions can be interpreted as:

- `vecScale`: the vector remains to point in the same direction, but will be ‘extended’ with a factor according the indicated number. If the absolute value of this number is < 1 the vector becomes shorter; if the number is < 0 the direction of the vector is inverted.

- `vecPlus`: the two vectors are drawn ‘head to tail’, and in this way they point to a new location (regardless of the order).

As an example we examine a few vectors in the 2-dimensional space:



Functions from vectors to vectors are called *mappings*. Extremely important are the *linear* mappings. These are mappings where every coordinate of the image vector is a linear combination of the coordinates of the original.

In the 2-dimensional space every linear mapping can be written as

$$f\ (x, y) \ = \ (\ a*x+b*y\ ,\ c*x+d*y\ )$$

where a, b, c en d can be chosen arbitrary. In mathematics, the n^2 numbers describing a linear mapping the n-dimensional space are written as an array of numbers with square brackets around it. For example:

$$\begin{pmatrix} \frac{1}{2}\sqrt{3} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2}\sqrt{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

describes a linear mapping in the 3-dimensional space (rotation over 30^{circ} around the z-axis). Such an array of numbers is called a *matrix* (plural: *matrices*).

In Gofer a matrix can be represented by a list of lists. Why not turn it into a protected (data)type at the same time:

```
data Matrix  =  Mat [[Float]]
```

Here it is needed to choose whether the lists are the rows or the columns of the matrix. In this reader is chosen for the rows, since every row matches one equation in the linear mapping.

If the column-representation is needed, there is a function converting rows into columns and vice versa. This process is called *transposing* of a matrix. The function performing this operation has the type:

```
matTransp  ::  Matrix -> Matrix
```

So, for example

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

The most important function on matrices, is the function which performs the linear mapping. This is called *applying* a matrix to a vector. The type of this function is:

```
matApply  ::  Matrix -> Vector -> Vector
```

The composition of two linear mappings is again a linear mapping. Two matrices mean two linear mappings; the composition of these mappings is again described by a matrix. The calculation of this matrix is called matrix *multiplication*. So there is a function:

```
matProd  ::  Matrix -> Matrix -> Matrix
```

Just as with function composition (the operator `(.)`) the function `matProd` is associative (so $A\times(B\times C) = (A\times B)\times C$). However, matrix multiplication is *not* commutative ($A\times B$ is not always equal to $B\times A$). The matrix product $A\times B$ is the mapping which first applies B, and then A.

The identity mapping is also a linear mapping. It is described by the *identity matrix.* This is a matrix with everywhere zeroes and a 1 on the diagonal. For each dimension there is a so called identity matrix, which is computed by the function

```
matId  ::  Int -> Matrix
```

Some linear mappings are em invertible. The inverse mapping (if it exists) is also linear, so it is described by a matrix:

```
matInv  ::  Matrix -> Matrix
```

The dimension of the image of a linear mapping need not to be the same as that of the original. For instance, an image of a 3-dimensional space onto a 2-dimensional plane is described by the matrix

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

So a mapping from p-dimensional space to a q-dimensional space has p columns and q rows. With the composition of mappings (matrix multiplication) these dimensions need to match. In the matrix product $A \times B$ the number of columns of A needs to be equal to the number of rows of B. Since number of columns of A denotes the dimension of the original of the mapping A it must be equal to the dimension of the image produced by the mapping B. The composition $A \times B$ has the same original as B, so as many columns as B' it has the same image of A, so it has as many rows as A. For example:

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

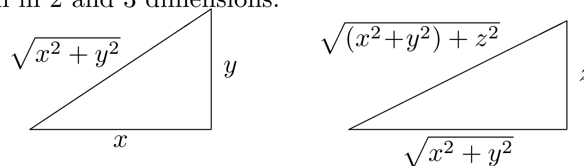
It is obvious the concepts of 'indentity matrix' and 'inverse' are only useful with square matrices, so matrices which have equal original and image spaces. And even with square matrices the inverse is not always defined. For example, the matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ has no inverse.

4.2.2 Elementary operations

In this an the next section the definitions will be provided for a number of functions on vectors and matrices.

The length of a vector

The length of a vector is determined according to the Pythagoras' theorem. The next pictures illustrate the situation in 2 and 3 dimensions:

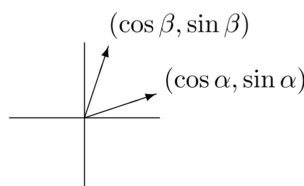


Generally speaking (in arbitrary dimension) the length of a vector can be calculated by the square root of the sum of the squared coordinates. Thus the function is:

```
vecLength (Vec xs) = sqrt (sum (map square xs))
```

Angle of two vectors

Assume two vectors with length 1. The ending points of these vectors are on the unit circle. If the angle of these vectors with the x axis are called α and β, the coordinates of the endpoints are $(\cos\alpha, \sin\alpha)$ and $(\cos\beta, \sin\beta)$.



The angle of two vectors with each other is $\beta - \alpha$. For the difference of two angles there is the following rule:

$$\cos(\beta - \alpha) \;=\; \cos\alpha\;\cos\beta \;+\; \sin\alpha\;\sin\beta$$

So in the case of two vectors the cosine of the enclosed angle equals the sum of the products of the corresponding coordinates (this is also the case in dimensions greater than 2). Because this formula is very important, it has been assigned a special name: the *inner product*. The value is computed by the function:

```
vecInprod (Vec xs) (Vec ys)  =  sum (zipWith (*) xs ys)
```

In order to determine the cosine of the angle of vectors with lengths other than 1 the inner product must be divided by the length. So, the angle can be calculated by:

```
vecAngle v w  =  arccos (vecInprod v w / (vecLength v * vecLength w))
```

The function `arccos` is the inverse of the cosine. If it is not built-in, it can be calculated with the function `inverse` in section 2.4.5.indexvector!inner product

page 33

The cosine of both 90° and −90° is 0. To determine whether two vectors are perpendicular, it is not needed to calculate the `arccos`: the inner product suffices. It is even not needed to divide by the lengths of the vectors, because zero is scale invariant:

```
vecPerpendicular v w  =  vecInprod v w == 0
```

The addition and extension of vectors

The functions `vecScale` and `vecPlus` are basic applications of the standard functions `map` and `zipWith`:

```
vecScale  ::  Float -> Vector -> Vector
vecScale k (Vec xs)  =  Vec (map (k*) xs)
vecPlus   ::  Vector -> Vector -> Vector
vecPlus (Vec xs) (Vec ys)  =  Vec (zipWith (+) xs ys)
```

Similar functions are useful on matrices too. It would be convenient to develop two functions which act like `map` and `zipWith`, which operate on the elements of *lists of* lists. These functions will be called `mapp` and `zippWith` (these are no standard functions).

To apply a function to all elements of a list of lists, it is needed to apply 'the applying to all elements of a list' to all elements of the big list. So `map f` must be applied to all elements of the big list:

```
mapp  :: (a->b) -> [[a]] -> [[b]]
mapp f  =  map (map f)
```

Alternatively:

```
mapp  =  map . map
```

For `zippWith` something similar holds:

```
zippWith :: (a->b->c) -> [[a]] -> [[b]] -> [[c]]
zippWith  =  zipWith . zipWith
```

These functions can be used in `matScale` and `matPlus`:

```
matScale  ::   Float -> Matrix -> Matrix
matScale k (Mat xss) =  Mat (mapp (k*) xss)
matPlus   ::  Matrix -> Matrix -> Matrix
matPlus (Mat xss) (Mat yss)  =  Mat (zippWith (+) xss yss)
```

Transposing matrices

A transposed matrix is a matrix of which the rows and and columns are swapped. This operation is also required on normal lists of lists (without the constructor `Mat`). Because of that we first design a function

```
transpose  ::  [[a]] -> [[a]]
```

After that `matTransp` is quite elementary:

```
matTransp  (Mat xss)  =  Mat (transpose xss)
```

The function `transpose` is a generalization of `zip`. Where `zip` zips *two* lists together into a list of *two pairs*, `transpose` zips a *list of* lists together into a list of *lists*.

This function can be defined recursively, but first let's take a look at an example. It must hold that

`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]]

If the list of lists consists of only one row, the function is simple: the row of n elements becomes n columns of each one element. So:

```
transpose [row]   =  map singleton row
                     where  singleton x = [x]
```

For the recursive case we assume that the transpose of everything except for the first line is already determined. So in the previous example that would be:

`transpose` [[4,5,6] , [7,8,9] , [10,11,12]] = [[4,7,10] , [5,8,11] , [6,9,12]]

How should the first row [1,2,3] be combined with this partial solution into the complete solution? The elements of it must be put in front of the recursive solution. So 1 will be put in front of [4,7,10], 2 will be put in front of [5,8,11], and 3 will be put in front of [6,9,12]. This can be easily achieved with `zipWith`:

```
transpose (xs:xss)  =  zipWith (:) xs (transpose xss)
```

With this the function `transpose` is defined. The only problem is that it cannot be applied to a matrix with zero rows, but that is a futile case anyhow.

Non-recursive matrix transposing

It is also possible to design a non-recursive definition of `transpose`, which leaves the 'dirty work' to the standard function `foldr`. This is because the definition has the form of

```
transpose (y:ys)  =  f y (transpose ys)
```

(with `f` being the partially parametrized function `zipWith (:)`). Functions of this form are a special case of `foldr` (see 'subsfoldr). As the function parameter of `foldr` it is possible to use `f`, that is `zipWith (:)`. The question remains what the 'neutral element' should be, that is, what is the result of `transpose []`.

An 'empty matrix' can be regarded as a matrix with 0 rows for each n elements. A transpose of that is a matrix with n empty lists. But what is the size of n? There are only zero rows available, so we cannot take a look at the length of the first row. Because we do not want to run the risk of choosing n too small, we choose n infinitily large: the transpose of a matrix with 0 rows of ∞ elements is a matrix with ∞ rows of 0 elements. The function `zipWith` takes care of the oversized infinitely long list, since it zips until one of the two lists has reached an end.

This somewhat complicated reasoning results in a very elegant definition of `transpose`:

```
transpose  =  foldr f e
       where  f = zipWith (:)
              e = repeat []
```

or even simply

```
transpose  =  foldr (zipWith (:)) (repeat [])
```

Functional programming is programming with functions...

Applying a matrix to a vector

A matrix is a representation of a linear mapping between vectors. The function `matApply` executes this linear mapping. For example:

`matApply` (`Mat` [[1,2,3] , [4,5,6]]) (`Vec` $[x, y, z]$) = `Vec` [$1x+2y+3z$, $4x+5y+6z$]

The number of *columns* of the matrix is equal to the number of coordinates of the *original* vector; the number of *rows* of the matrix equals the dimension of the *image* vector.

For each coordinate of the image vector only numbers from the corresponding row of the matrix are needed. It is therefore obvious to design `matApply` as the `map` of another function on the (rows of the) matrix:

```
matApply (Mat m) v  =  Vec (map f m)
```

The function `f` operates on one row of the matrix, and results one element of the image vector. For example on the second row:

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

The function `f` calculates the inner product of its parameter with the vector `v` ($[x, y, z]$ in the example). So the entire function `matApply` becomes:

```
matApply :: Matrix -> Vector -> Vector
matApply (Mat m) v  =  Vec (map f m)
                where  f row = vecInprod (Vec row) v
```

Two items to pay attention to in this definition:

- The constructor function `Vec` is applied at the right places to keep the type accurate. The function `vecInprod` expects two vectors. The parameter `v` is already a vector, but `row` is an ordinary list, which should first be made into a vector by applying `Vec`.
- The function `f` may, except for its parameter `row`, make use of `v`. Local functions are always allowed to use the parameters of the function they are defined within.

The identity matrix

In each dimension there is an identical image. This image is described by a square matrix with a 1 on the diagonal, and zeroes everywhere else. To design a function which results the right identity matrix for each dimension, it would be practical to first define an infinitily large identity matrix, so the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \ddots \end{pmatrix}$$

The first row of that matrix is an infinite list of zeroes with a one in front, so `1:repeat 0`. The other rows are determined by iteratively adding an extra 0 in front. This can be done by using the function `iterate`:

```
matIdent ::  Matrix
matIdent  =  Mat (iterate (0:) (1:repeat 0))
```

The identity matrix of dimension n can now be obtained by taking n rows of this infinite matrix, and taking n elements of each row:

```
matId :: Int -> Matrix
matId n  =  Mat( map (take n) (take n xss))
     where  (Mat xss) = matIdent
```

Matrix multiplication

The product of two matrices describes the image which is the composition of the two mappings belonging to the two matrices. To see how the product can be calculated, we apply the matrices $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)$ after each other on the vector $\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$. (We use the notation $\otimes$ to describe a matrix product and $\odot$ to describe the application of matrix to a vector. Keep in mind the difference between matrices and vectors.)

$$\begin{aligned} & \left(\begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} e & f \\ g & h \end{pmatrix} \right) \odot \begin{pmatrix} x \\ y \end{pmatrix} \\ = & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \left(\begin{pmatrix} e & f \\ g & h \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix} \right) \\ = & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} ex + fy \\ gx + hy \end{pmatrix} \\ = & \begin{pmatrix} a(ex{+}fy) + b(gx{+}hy) \\ c(ex{+}fy) + d(gx{+}hy) \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} (ae+bg)x + (af+bh)y \\ (ce+dg)x + (cf+dh)y \end{pmatrix}$$
$$= \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix}$$

So the product of two matrices can be calculated as follows: each element is the *inner product* of a *row* of the left matrix and a *column* of the right matrix. The length of a row (the number of columns) of the left matrix must equal the length of a column (the number of rows) of the right matrix. (This was already observed at the end of the former section).

The question remains how to design matrix multiplication as a function. Therefore we again take a look at the example of the former section:

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

The number of rows of the result is the same as the number of rows of the left matrix. Therefore we try to design `matProd` as `map` on the left matrix:

```
matProd (Mat a) (Mat b)  =  Mat (map f a)
```

Here the function `f` operates on one row of the left matrix, and results one row of the result. For example the second row:

```
f [2,1,0] = [2,5,8,5]
```

How are these numbers `[2,5,8,5]` calculated? By computing the inner product of `[2,1,0]` with all *columns* of the right matrix.

However, matrices are stored as *rows*. To obtain columns, it is needed to apply the `transpose`-function. So the final result is:

```
matProd (Mat a) (Mat b)  =  Mat (map f a)
                    where  f aRow  = map (vecInprod (Vec aRow)) bCols
                           bCols   = map Vec (transpose b)
```

The function `transpose` only operates on 'bare' lists of lists (so it does not work on matrices). It results another list of lists. With `map Vec` this list is turned into a list of vectors. Of all these vectors the inner product with the rows of `a` can be computed (where the rows of `a` are regarded as vectors).

4.2.3 Determinant and inverse

The use of the determinant

Only bijective (one-to-one) images are invertible. If there are images with more than one original, the inverse cannot be defined.

So if a matrix is not square, it cannot be inverted (because the image space has lower dimension (causing images with more than one original) or a higher dimension (causing images without an original)). But even square matrices do not always represent bijective images. For example, the matrix $\left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ maps every point to the origin, so it has no inverse. The matrix $\left(\begin{smallmatrix} 1 & 1 \\ 2 & 2 \end{smallmatrix}\right)$ maps every point onto the line $y = 2x$, and so this one has no inverse either. Also the matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ maps all points onto a line.

Only if the second row of a 2-dimensional matrix is no multiple of the first, the matrix can be inverted. So a matrix $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ is only invertible if $\frac{a}{c} \neq \frac{b}{d}$, or $ad - bc \neq 0$. This value is called the *determinant* of the matrix. If the determinant is zero, the matrix cannot be inverted; if it is non-zero, the matrix *can* be inverted.

The determinant of (square) matrices in higher dimension can also be calculated. For a 3×3 matrix it looks like:

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \det \begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \det \begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \det \begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

So you first start by splitting the matrix into a first row (a, b, c) and the rest $\binom{d\ e\ f}{g\ h\ i}$. Then you calculate the determinants of the 2×2- matrices obtained from the 'other rows' by leaving out one column at a time ($\binom{e\ f}{h\ i}$, $\binom{d\ f}{g\ i}$ and $\binom{d\ e}{g\ h}$). Those are multiplied by the corresponding elements from the first row. Finally you add them all, where each alternate term is negated. This mechanism applies also in higher dimensions than 3.

Definition of the determinant

We are now turning this informal description of the determinant into a function definition. We make use of the splitting into the first row of the matrix and the rest, so the definition looks like

```
det (Mat (row:rows))  =  ...
```

We have to leave out columns of the resting rows `rows`. To convert rows into columns they must be transposed. The definition becomes something like

```
det (Mat (row:rows))  =  ...
                where  cls = transpose rows
```

In all possible ways there must be left out one column of the list. This can be done with the combinatorial function `gaps` defined in exercise 4.5. The result is a list of n lists of lists. These lists have to be transposed back, and they need to be turned into little matrices with `Mat`:

page 83

```
det (Mat (row:rows))  =  ...
                where  cols = transpose rows
                       mats = map (Mat.transpose) (gaps cols)
```

Of all these matrices the determinant bust be calculated. This is of course done by calling the function recursively. The resulting determinants need to be multiplied by the corresponding elements of the first row:

```
det (Mat (row:rows))  =  ...
                where  cols  = transpose rows
                       mats  = map (Mat.transpose) (gaps cols)
                       prods = zipWith (*) row (map det mats)
```

The products `prods` which are resulted must be added, where each alternate term must be negated. This can be done by using the function `altsum` (meaning 'alternating sum') defined with the aid of an infinite list:

```
altsum xs  =  sum (zipWith (*) xs plusMinusOne)
       where  plusMinusOne  =  1 : -1 : plusMinusOne
```

So the function definition becomes:

```
det (Mat (row:rows))  =  altsum prods
                where  cols  = transpose rows
                       mats  = map (Mat.transpose) (gaps cols)
                       prods = zipWith (*) row (map det mats)
```

This can be simplified a little bit. This is because the back transposing of the matrices is not needed, because the determinant of a transposed matrix remains the same as the original matrix. Furthermore it is not necessary to name the temporary results (`cols`, `mats` en `prods`) The definition can now be written as:

```
det (Mat (row:rows)) =
  altsum (zipWith (*) row (map det (map Mat (gaps (transpose rows)))))
```

Because `det` is a recursive function, we must not forget to define a non-recursive basis case. For this the 2×2 case can be used, but it is even easier to define the 1×1 case:

```
det (Mat [[x]])  =  x
```

The final result, where some braces are left out by using function composition, looks like:

```
det  ::  Matrix -> Float
det (Mat [[x]])    =  x
det (Mat (row:rows)) =
  (altsum . zipWith (*) row . map det . map Mat . gaps . transpose) rows
```

The inverse of a matrix

The determinant is not only of use to determine whether there exists an inverse, but also to actually calculate the inverse. Of course the determinant must not be equal to zero.

The inverse of a 3 matrix can be calculated as follows: determine a matrix of nine 2×2 matrices, obtained by leaving out one row and one column from the matrix. Determine of each matrix the determinant, add alternatingly a minus sign, and divide everything by the determinant of the whole matrix (which must be $\neq 0$!).

An example is probably clearer. The inverse of $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$ is calculated as follows:

$$\frac{\begin{pmatrix}
+\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ \cancel{b} & e & h \\ \cancel{c} & f & i \end{pmatrix} & -\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ \cancel{c} & f & i \end{pmatrix} & +\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & e & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\
-\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & \cancel{e} & h \\ c & \cancel{f} & i \end{pmatrix} & +\det\begin{pmatrix} a & \cancel{d} & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & \cancel{f} & i \end{pmatrix} & -\det\begin{pmatrix} a & \cancel{d} & g \\ b & \cancel{e} & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\
+\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & e & \cancel{h} \\ c & f & \cancel{i} \end{pmatrix} & -\det\begin{pmatrix} a & d & \cancel{g} \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & f & \cancel{i} \end{pmatrix} & +\det\begin{pmatrix} a & d & \cancel{g} \\ b & e & \cancel{h} \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix}
\end{pmatrix}}{\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}$$

The striked out elements are only in this example to show which elements are left out; so actually there are nine 2×2 matrices.

Pay attention to which elements need to be left out: in the rth row of the big matrix each time the rth *column* of the matrices is left out, while in the kth column of the big matrix the kth *row* is left out!

The matrix inverse function `matInv` can now be written in the same fashion as `det`, meaning you will need to apply `gaps`, `Transpose`, `Mat`, etc. at the right place at the right time. It is left to the reader as an exercise to fill out the details (see exercise 4.8). page 83

4.3 Polynomials

4.3.1 Representation

A *polynomial* is the sum of *terms*, where each term consists of the product of a real number and a natural power of a variable.

$x^2 + 2x + 1$
$4.3x^3 + 2.5x^2 + 0.5$
$6x^5$
x
3

The highest power occurring is called the *degree* of the polynomial. So in the upper examples the degree is respectively 2, 3, 5, 1 and 0. It might be a little strange to call 3 a polynomial; however, the number 3 equals $3x^0$, so it is indeed the product of a number and a natural power of x.

You can compute with polynomials: you can add, subtract or multiply polynomials. For example, the product of the polynomials $x + 1$ and $x^2 + 3x$ equals $x^3 + 4x^2 + 3x$. However, dividing two polynomials by each other the result is not guaranteed to be a polynomial. It is very convenient that numbers are polynomials too: we can now compute that adding $x + 1$ and $-x$ results in the polynomial 1.

In this section the data type `Poly` will be designed, with which polynomials can be defined. In the next section a number of functions is defined, operating on this kind of polynomials:

```
pAdd  :: Poly -> Poly -> Poly
pSub  :: Poly -> Poly -> Poly
pMul  :: Poly -> Poly -> Poly
```

```
pEq   :: Poly -> Poly -> Bool
pDeg  :: Poly -> Int
pEval :: Float -> Poly -> Float
polyString :: Poly -> String
```

A possible representation for polynomials is 'function from float to float'. But the disadvantage of this method is that you cannot inspect two polynomials any more; you will have a function which can only be applied to values. It is also not possible to define an equality operator; thus it is not possible to test whether the product of the polynomials x and $x + 1$ equals the polynomial $x^2 + x$.

Thus it would be better to represent a polynomial as a data structure with numbers. It is obvious to represent a polynomial as a list of terms, where each term is characterized by a `Float` (the coefficient) and an `Int` (the exponent). So a polynomial can be represented as a list of two pairs. We immediately turn it into a *data*type with the following definition:

```
data Poly  =  Poly [Term]
data Term  =  Term (Float,Int)
```

Note that the identifiers `Poly` and `Term` are used both as identifier of the type and as the identifier of the (sole) constructor function. This is allowed, because it is always clear from the context which one is meant. The identifier `Poly` is a type in type declarations like

```
pEq  ::  Poly -> Poly -> Bool
```

but it is a constructor function in function definitions like

```
pDeg (Poly [])  =  ...
```

A number of examples of representations of polynomials 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 []`

Just as with the rational numbers in section 3.3.3 we again encounter the problem of non-unique representations of one polynomial. For example, the polynomial $x^2 + 7$ can be represented by the expressions:

page 54

```
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) ]
```

Just as with rational numbers it is necessary to 'simplify' the polynomial after performing operations on it. In this case, simplification consists of:

- sorting of the terms, so that the terms with the highest exponent are in front;
- unification of terms with equal exponents;
- removal of terms with zero coefficients.

An alternative method would be to *not* simplify the polynomials, but then it would be necessary to do more in the function `pEq` which compares polynomials.

4.3.2 Simplification

To simplify polynomials we design a function

```
pSimple  ::  Poly -> Poly
```

This function performs the three mentioned aspects of the simplification process, and can therefore be written as the function composition

```
pSimple (Poly xs) = Poly (simple xs)
               where  simple = removeZero . uniExpo . sortTerms
```

The task remains to write down the three composing functions. All three operate on lists of terms.

In section 3.1.4 a function was defined which sorts a list, provided that the values could be ordered, and the list was sorted from small to large. One could think of a more general sorting function, which is passed an extra parameter which is the criterion of in which order the elements must be put. This function would be very useful, because it could be used with 'has a larger exponent' as a

page 43

sorting criterion. Therefore we first design a function `sortBy`, which can be used with the writing of `sortTerms`.

Te general sorting function `sortBy` has the type:

```
sortBy  ::  (a->a->Bool) -> [a] -> [a]
```

Except for the list which should be sorted it has a function as a parameter. This parameter function results `True` if its first parameter must be placed *in front of* its second parameter. The definition of `sortBy` shows resemblance with that one of `isort` in section 3.1.4. The difference is that the extra parameter is used as comparison function instead of `<`. The definition becomes:

page 43

```
sortBy inFrontOf xs  =  foldr (insertBy inFrontOf) [] xs
insertBy inFrontOf e []  =  [e]
insertBy inFrontOf e (x:xs)
      | e `inFrontOf` x  =  e : x : xs
      | otherwise        =  x : insertBy inFrontOf e xs
```

Examples of the usage of `sortBy` are:

```
? sortBy (<) [1,3,2,4]
[1, 2, 3, 4]
? sortBy (>) [1,3,2,4]
[4, 3, 2, 1]
```

When sorting terms on the basis of their exponent `sortBy` can now be used. This function has as `inFrontOf` function a function which checks if the exponent is bigger:

```
sortTerms ::  [Term] -> [Term]
sortTerms  =  sortBy expoBigger
      where  Term(c1,e1) `expoBigger` Term(c2,e2)  =  e1>e2
```

The second function which is needed, is the function which unifies two terms with the same exponents. This function may assume that the terms are already sorted by exponent. So terms qith equal exponents are next to each other. The function does not do anything with lists of zero or one element. With lists having two or more elements there are two possibilities:

- the exponents of the first two elements are equal; the elements are unified, the new element is put in front of the rest, and the function is called again, so that the new element can be unified with even more elements.
- the exponents of the first two elements are P*not* equal; the first element will be unchanged in the result, at the rest it will take a closer look (maybe the second element equals the third).

We recognize this all in the definition:

```
uniExpo   :: [Term] -> [Term]
uniExpo []         =  []
uniExpo [t]        =  [t]
uniExpo (Term(c1,e1):Term(c2,e2):ts)
         | e1==e2     =  uniExpo (Term(c1+c2,e1):ts)
         | otherwise  =  Term(c1,e1) : uniExpo (Term(c2,e2):ts)
```

The third needed function is easy to make:

```
removeZero ::  [Term] -> [Term]
removeZero  =  filter coefNonZero
          where  coefNonZero (Term(c,e))  =  c/=0.0
```

As desired the three functions can be defined locally in `pSimple`:

```
pSimple (Poly xs) =  Poly (simple xs)
      where  simple  = vN . sE . sT
             sT  =  sortBy expoBigger
             sE []         =  []
             sE [t]        =  [t]
             sE (Term(c1,e1):Term(c2,e2):ts)
              | e1==e2     =  sE (Term(c1+c2,e1):ts)
              | otherwise  =  Term(c1,e1) : sE (Term(c2,e2):ts)
             vN  =  filter coefNonZero
             coefNonZero (Term(c,e))  =  c/=0.0
             Term(c1,e1) `expoBigger` Term(c2,e2)  =  e1>e2
```

The function `pSimple` removes *all* terms of which the coefficient is zero. So the zero polynomial

will be represented by `Poly []`, an empty list of terms. In that respect the zero polynomial differs from other polynomials which do not contain the variable. For example, the polynomial '3' is represented by `Poly [Term(3.0,0)]`.

4.3.3 Arithmetic operations

The addition of two polynomials is very simple. The lists of terms can just be concatenated. After that `pSimple` takes care of the sorting, unification and removal of zero terms:

```
pAdd  ::  Poly -> Poly -> Poly
pAdd (Poly xs) (Poly ys)  =  pSimple (Poly (xs++ys))
```

To subtract two polynomials, we add the first polynomial to the negated second:

```
pSub  ::  Poly -> Poly -> Poly
pSub p1 p2  =  pAdd p1 (pNeg p2)
```

The question remains how to compute the opposite of a polynomial: therefore the negative of each term must be calculated.

```
pNeg  ::  Poly -> Poly
pNeg (Poly xs)   =  Poly (map tNeg xs)
tNeg  ::  Term -> Term
tNeg (Term(c,e)) = Term(-c,e)
```

Polynomial multiplication is a little more complicated. It is needed to multiply each term of the first polynomial by each term of the other polynomial This asks for a higher order function: 'do something with each element of a list in combination with each element of another list'. This function will be called `cpWith` The `cp` stands for cross product, the `With` is for analogy with `zipWith`. If the first list is empty, there is nothing to combine. If the first row is of the form `x:xs`, the first element `x` needs to be combined with all elements in the second list, besides it is needed to determine the cross product of `xs` with the second list. This results in the definition:

```
cpWith  ::  (a->b->c) -> [a] -> [b] -> [c]
cpWith f [] ys     = []
cpWith f (x:xs) ys = map (f x) ys ++ cpWith f xs ys
```

The definition can directly be incorporated with the multiplication of polynomials:

```
pMul  ::  Poly -> Poly -> Poly
pMul (Poly xs) (Poly ys)  =  pSimple (Poly (cpWith tMul xs ys))
```

Here the function `tMul` is used, which multiplies two terms. As shows from the example $3x^2$ multiplied with $5x^4$ is $15x^6$, the coefficients must be multiplied, and the exponents added:

```
tMul  :: Term -> Term -> Term
tMul (Term(c1,e1)) (Term(c2,e2))  =  Term (c1*c2,e1+e2)
```

Because we simplify the polynomials each time we perform an operation, the term with the highest exponent will always be in front. Therefore the degree of a polynomial equals the exponent of the first term. Only for the zero polynomial another definition is needed.

```
pDeg  ::  Poly -> Int
pDeg (Poly [])            =  0
pDeg (Poly (Term(c,e):ts)) =  e
```

Two simplified polynomials are equal if all the terms are equal. Two terms are equal if the coefficients and the exponents are equal. All this is easily translated to functions:

```
pEq  :: Poly -> Poly -> Bool
pEq (Poly xs) (Poly ys)  =  length xs==length ys
                            && and (zipWith tEq xs ys)
tEq  :: Term -> Term -> Bool
tEq (Term(c1,e1)) (Term(c2,e2))  =  c1==c2 && e1==e2
```

The function `pEval` must evaluate a polynomial with a specific value for x. For that purpose all terms must be evaluated, and the results must be added:

```
pEval :: Float -> Poly -> Float
pEval w (Poly xs)   =  sum (map (tEval w) xs)
tEval :: Float -> Term -> Float
tEval w (Term(c,e)) =  c * w^e
```

As the last function we design a function to show the polynomial as a string. For this purpose the terms must be shown as a string, and between them needs to be a + sign:

```
polyString  :: Poly -> String
polyString (Poly [])     =  "0"
polyString (Poly [t])    =  termString t
polyString (Poly (t:ts)) =  termString t ++ " + "
                            ++ polyString (Poly ts)
```

When displaying a term we leave out the coefficient and the exponent if it is 1. If the exponent is 0, the variable is left out, but never the coefficient. The exponent is denoted by a ^-sign, just as with Gofer expressions. This results in the function

```
termString  :: Term -> String
termString (Term(c,0))    =  floatString c
termString (Term(1.0,1))  =  "x"
termString (Term(1.0,e))  =  "x^" ++ intString e
termString (Term(c,e))    =  floatString c ++ "x^" ++ intString e
```

The function `floatString` is not defined yet. With a lot of trouble it is possible to define such a function, but it is not necessary: in section **??** the function `show` is introduced. page ??

Exercises

4.1 What will happen with the order of the elements of `segs [1,2,3,4]` if the parameters of `++` in the definition of `segs` are reversed ?

4.2 Write `segs` as a combination of `inits`, `tails` and standard operations. The order of the elements in the result needs not to be the same as on page 67.

4.3 Given a list of `xs` with `n` elements, determine the number of elements of `inits xs`, `segs xs`, `subs xs`, `perms xs`, en `combs k xs`.

4.4 Write down a function `bins :: Int -> [[Char]]` determining all binary numbers (as strings of zeroes and ones) with given length. Compare this function with the function `subs`.

4.5 Design a function `gaps` listing all possibilities to remove one element from a list. For example:

`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]]

4.6 Compare the conditions concerning the dimensions of matrices for matrix multiplication with the type of the function composition operator `(.)`.

4.7 Verify that the recursive definition of a matrix determinant `Det` resembles the explicit definition for determinants of 2×2 matrices from section 4.2.3. page 77

4.8 Write the function `matInv`.

4.9 In section 4.2.2 the function `Transpose` was described as being the generalization of `zip`. In section 3.3.4 `zip` was written as `zipWith make2pair`. Write a function `crossprod` which is the generalization of `cp` like `transpose` is a generalization of `zip`. How can `length (crossprod xs)` be calculated without using `crossprod`? page 74 page 55

4.10 Another way of representing polynomials is a list of coefficients, so the exponents are not stored. The price to pay is that 'missing' terms like `0.0` must be stored. It is the handiest to put the terms with the smallest exponent in front. So for example:

$x^2 + 2x$	`[0.0, 2.0, 1.0]`
$4x^3$	`[0.0, 0.0, 0.0, 4.0]`
5	`[5.0]`

Design functions to determine the degree of a polynomial and to calculate the sum and the product of two polynomials in this representation.

Chapter 5

Code transformation

5.1 Efficiency

5.1.1 Time

Every call of a function when calculating an expression, costs a certain amount of time. Those function calls are counted by the interpreter: these are the *reductions* reported after the answer.

The number of needed reductions can be smaller when calling an expression for the second time. For example, take a look at the next case. The function `f` is a very complicated function, for example the faculty function. The constant `k` is defined involving a call of `f`. The function `g` uses `k`:

```
f x = product [1..x]
k   = f 5
g x = k + 1
```

The constant `k` is not calculated during the analysis of the file thanks to lazy evaluation. It is evaluated when it is used for the first time, for example when calling `g`:

```
? g 1
121
(57 reductions, 78 cells)
```

If after that `g` is called again, the value of `k` needs not to be calculated. The second call of `g` is therefore a lot quicker:

```
? g 1
121
(3 reductions, 8 cells)
```

So do not hesitate to put all kinds of complicated constant definitions in a file. Only when they are used they cost time. Functions defined by a higher order function which is partially parametrized, for instance

```
sum  =  foldr (+) 0
```

cost the second time they are used one reduction less:

```
? sum [1,2,3]
6
(9 reductions, 19 cells)
? sum [1,2,3]
6
(8 reductions, 17 cells)
```

After the second call the number of reductions remains the same.

5.1.2 Complexity analysis

When analyzing the usage of time of a function mostly the precise number of reductions is not needed. It is much more interesting to see how this number of reductions changes if the size of the inpt is changed.

For example, suppose we have three lists with respectively 10, 100 and 1000 elements (the figures from 1 in reversed order) The number of reductions of some functions on these lists are as follows:

elements	`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	$n{+}1$	$4n{+}2$	$3n{+}2$	$5n{+}2$	$n^2{+}3n{+}3$

The calculation of the `head` can always be done in two reductions. The head of a list needs only to be taken of the beginning. The function `last` however, needs more time if the list is longer. This is because the list has to be traversed to the end, and it takes longer when the list is longer. Also with `length`, `sum` and `++` the function of the needed number of reductions is a linear function of the length of the list. However with `isort` the number of reductions is a quadratic function of the length of the list.

It is not always that easy to determine the number of reductions as a function of the length of the parameter. Therefore in practice it suffices to indicate the *order* of this function: the needed time for `head` is *constant*, for `last`, `length`, `sum` and `++` is *linear*, and for `isort` it is *quadratic* in the length of the parameter. The order of a function is denoted by the letter $\mathcal{O}$. A quadratic function is written as $\mathcal{O}(n^2)$, a linear function with $\mathcal{O}(n)$, and a constant function with $\mathcal{O}(1)$.

The number of needed reductions for both `length` as `sum` is $\mathcal{O}(n)$. If the needed time for two functions has the same order, it can only differ a constant factor.

The following table provides some insight in the speed with which the different function increase:

orde	$n = 10$	$n = 100$	$n = 1000$	name
$\mathcal{O}(1)$	1	1	1	constant
$\mathcal{O}(\log n)$	3	7	10	logarithmic
$\mathcal{O}(\sqrt{n})$	3	10	32	
$\mathcal{O}(n)$	10	100	1000	linear
$\mathcal{O}(n \log n)$	30	700	10000	
$\mathcal{O}(n\sqrt{n})$	30	1000	31623	
$\mathcal{O}(n^2)$	100	10000	10^6	quadratic
$\mathcal{O}(n^3)$	1000	10^6	10^9	cubic
$\mathcal{O}(2^n)$	1024	10^{30}	10^{300}	exponentially

In this table 1 is taken for the constant factor, and as a basis for the logarithm 2. It does not matter for the order what basis are used, because due to ${}^g\log n = {}^2\log n / {}^2\log g$ this only differs a constant factor.

Keep in mind that, for instance a $\mathcal{O}(n\sqrt{n})$ algorithm is not always better than an $\mathcal{O}(n^2)$ algorithm to solve the same problem. If for instance the constant factor in the $\mathcal{O}(n\sqrt{n})$ algorithm is ten times as big as that of the $\mathcal{O}(n^2)$ algorithm, then the first algorithm will only be faster for $n > 100$. Sometimes algorithms with a lower complexity do more operations per step than algorithms with a higher complexity, and so we gave a much bigger constant factor. However, if n is large enough, algorithms with a lower complexity are always faster.

Mostly it is easy to determine the order of a needed number of reductions from the definition of the function. The order can be used as a measure for the *complexity* of the function. Here are some examples. The first example is about a non-recursive function. Then three examples will be discussed of recursive functions of which the parameter of the recursive call is slightly less (1 shorter or 1 less), so functions with definitions of the form

```
f (x:xs)   =  ... f xs ...
g (n+1)    =  ... g n  ...
```

Finally three functions of recursive functions of which the parameter of the recursive call is about half of the original, like in

```
f (Branch x le ri)  =  ... f le ...
g (2*n)             =  ... g n  ...
```

- The function is non recursive, and contains only calls of functions with complexity $\mathcal{O}(1)$. Example: `head`, `tail`, `abs`. (Remark: the time needed to to calculate the result of `tail` is linear, but to calculate the tail itself takes constant time). A number of times a constant factor remains a constant factor, so these functions have the same complexity $\mathcal{O}(1)$.

- The function does *one* recursive call of itself with a slightly decreased parameter. The final result is then determined in constant time. Example: `fac`, `last`, `insert`, `length`, `sum`. And also: `map f` and `foldr f`, where `f` has constant complexity. These functions have complexity $\mathcal{O}(n)$.
- The function does *one* recursive call of itself with a slightly decreased parameter. The final result will be calculated in linear time. Example: `map f` or `foldr f`, where `f` has linear complexity. A special case of this is the function `isort`; this function is defined as `foldr insert []`, where `insert` has linear complexity. These functions have complexity $\mathcal{O}(n^2)$.
- The function does *two* recursive calls of itself with a slightly decreased parameter. Example: the previous version of last version of `subs` in section 4.1.1. This kind of functions is incredibly awful: with each step in the recursion the number of reductions doubles. These functions have complexity $\mathcal{O}(2^n)$. page 69
- The function does *one* recursive call of itself with a parameter which is about twice as small. Then the final result is calculated in constant time. For example: `elemTree` in section 3.4.2. These functions have complexity $\mathcal{O}(\log n)$. page 59
- The function divides its parameter in two parts and recursively calls itself with *both* parts. The final result is calculated in constant time. For example: `size` in section 3.4.1. These functions have complexity $\mathcal{O}(n)$. page 58
- The function splits it parameter in two parts and calls itself recursively on both parts. The final result is then calculated in linear time. For instance: `labels` in section 3.4.2, `msort` in section 3.1.4. These functions have complexity $\mathcal{O}(n \log n)$. page 60 page 44

Summarized in a table:

parameter	num.rec.calls	afterwards	example	complexity
none	0	constant	`head`	$\mathcal{O}(1)$
smaller	1	constant	`sum`	$\mathcal{O}(n)$
smaller	1	linear	`isort`	$\mathcal{O}(n^2)$
smaller	2	arb.	`subs`	$\mathcal{O}(2^n)$
half	1	constant	`elemTree`	$\mathcal{O}(\log n)$
half	1	linear		$\mathcal{O}(n)$
half	2	constant	`size`	$\mathcal{O}(n)$
half	2	linear	`msort`	$\mathcal{O}(n \log n)$

5.1.3 Improving efficiency

The smaller the parameter of a recursive call, the shorter it takes to calculate the function. From the former follows that next to that there are still three other ways to improve the efficiency of a function:

- preferably do recursive calls on half of the original parameters than on a with one decreased parameter;
- preferably one than two recursive calls;
- keep the remaining work after the recursive call(s) preferably constant instead of linear.

With some functions these methods can indeed be applied to obtain a faster function. We take a look at an example for every method.

a. Small parameters

The operator++ is defined with induction to the left parameter:

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

The needed time to calculate `++` is therefore linear in the length of the left parameter. The length of the right parameter does not influence the time taken. If two lists have to be concatenated, but the order of the elements does not matter, then it would be more efficient to put the shortest of the two in front.

The operator is associative, meaning that the value of `(xs++ys)++zs` equals that of `xs++(ys++zs)`. But it *does* matter to the complexity which of the two expressions is calculated.

Suppose the length of `xs` equals x, the length of `ys` y and that of `zs` z. Then holds:

time for...	`(xs++ys)++zs`	`xs++(ys++zs)`
first `++`	x	x
second `++`	$x+y$	y
total	$2x+y$	$x+y$

So if `++` associates to the right it will be calculated faster than if it associates to the left. That is why `++` is defined in the prelude as a right associating operator (see also section 2.1.3). page 22

b. Halve i.s.o. decreasing (next linear)

Look at some functions where the work after the recursive call is linear. A recursive function calling itself twice with a parameter which is half as big (for example a left and right subtree) has complexity $\mathcal{O}(n \log n)$. So that is better than a function calling itself once with a parameter which is only a constant amount smaller. (for example the tail of a list), which has complexity $\mathcal{O}(n^2)$.

The sorting function `msort`

```
msort xs | lenXs <= 1  =  xs
         | otherwise   =  merge (msort ys) (msort zs)
   where ys     =  take half xs
         zs     =  drop half xs
         half   =  lenXs / 2
         lenXs  =  length xs
```

so it is more efficient than the function `isort`

```
isort  []     =  []
isort (x:xs)  =  insert x (isort xs)
```

With many functions on lists an algorithm like `isort` is the most obvious. It pays to always check if there is also a `msort`-like algorithm possible. Think of the credo

Divide and Conquer

meaning in this case: 'divide the parameter in two roughly equally big lists; apply the function recursively to the two parts, and combine the partial solutions'. In the function `msort` the dividing is done by the functions `take` and `drop`; the conquering happens by the function `merge`.

c. Halving i.s.o. decreasing (next constant)

Regard functions which call itself once and then calculate the result in constant time. An example is the involution function:

```
x ^ 0      =  1
x ^ (n+1)  =  x  *  x ^ n
```

This function has complexity $\mathcal{O}(n)$. Would it have been halved at the recursive call, the complexity would only be $\mathcal{O}(\log n)$. In the case of involution this is indeed possible:

```
x ^ 0        =  1
x ^ (2*n)    =  square (x^n)
x ^ (2*n+1)  =  x * square (x^n)
square x     =  x * x
```

In this definition the well-known rule $x^{2n} = (x^n)^2$ is used. In Gofer the cases 'even parameter' and 'odd parameter' can be distinguished with the help of patterns. In languages where this is not possible it can always be done with the function `even`.

d. Few recursive calls (halving parameter)

If you can choose, it would be better to do one recursive call than two. Therefore it is possible to find an element in a search tree in $\mathcal{O}(\log n)$ time:

```
search e Leaf                          =  False
search e (Branch x le ri)  |  e==x  =  True
                           |  e<x   =  search e le
                           |  e>x   =  search e ri
```

this is because the function is called recursively only on one half: the left sub-tree if `e<x`, or the right sub-tree if `e>x`. In a tree where the elements are not ordered, searching generally costs $\mathcal{O}(n)$ time, because a recursive call on the left *and* the right subtree is needed:

```
search e Leaf                             =  False
Search e (Branch x le ri)  | e==x         =  True
                           | otherwise    =  search e le || search e ri
```

if the element is to be found in the left tree you are lucky, but else you will need to call `search` again on the right tree.

e. Few recursive calls (decreased parameter)

The execution of one recursive call instead of two is really spectacular if the parameter does not halve but only decreases a little: it turns an exponential algorithm into a linear algorithm! So it would be a sin to do two recursive calls with the same parameter, like in `subs`:

```
subs []      =  [[]]
subs (x:xs)  =  map (x:) (subs xs)  ++  subs xs
```

Like in section 4.1.1 is pointed out, the recursive call can be put in a `where` clause, resulting in only a single execution of the recursive call:

page 69

```
subs []      =  [[]]
subs (x:xs)  =  map (x:) (subsxs)  ++  subsxs
         where  subsxs = subs xs
```

Although `subs` is not turned into a linear function because the `++` costs more than constant time, there is a considerable time gain.

If the two recursive calls do not have the same parameter, this method does not work. Still, sometimes an improvement is possible. Take a look at the function of Fibonacci for instance:

```
fib 0      =  0
fib 1      =  1
fib (n+2)  =  fib n + fib (n+1)
```

The function is called twice with a parameter which is not much smaller, at least not halve of it. The complexity of `fib` is therefore $\mathcal{O}(2^n)$.

An improvement of this is possible by writing an extra function, which does more than the desired. In this case we make a function `fib' n`, which results except for the value of `fib n` also the value of `fib (n-1)`. So the function results a pair. The function `fib` can be written by using `fib'`:

```
fib 0  =  0
fib n  =  snd (fib' n)
   where  fib' 1     =  (0,1)
          fib' (n+1) =  (b,a+b) where (a,b)=fib' n
```

The function contains onl one recursive call, and therefore it has complexity $\mathcal{O}(n)$.

f. Keep the combining work small

With 'divide and conquer' functions, which combine the result of two recursive calls into a final result it is important how much work the combining costs. If the combining costs constant time, then the complexity is $\mathcal{O}(n)$; does it cost linear time, then the complexity will be $\mathcal{O}(n \log n)$.

The function `labels`, which puts the elements of a tree in a list, has complexity $\mathcal{O}(n \log n)$:

```
labels Leaf               =  []
labels (Branch x le ri)   =  labels le ++ [x] ++ labels ri
```

That is because the 'conquer' operator is used, which costs linear time.

Even this can be improved, by writing an extra function, which does in fact more than asked for:

```
labelsBefore  ::  Tree a -> [a] -> [a]
labelsBefore tree xs   =  labels tree ++ xs
```

Given this function `labels` cna be written as

```
labels tree  =  labelsBefore tree []
```

In this way this does not gain any time, of course. But there is another possible definition for `labelsBefore`:

```
labelsBefore Leaf             xs  =  xs
```

```
labelsBefore (Branch x le ri) xs  =  labelsBefore le (x:labelsBefore ri xs)
```

This function does two recursive calls on the subtrees (like `labels`), but the 'conquering' costs only constant time (for the operator `:`). Therefore the complexity of this algorithm is only $\mathcal{O}(n)$, an improvement to the $\mathcal{O}(n \log n)$ of the original algorithm.

5.1.4 Memory usage

Except for the needed time an algorithm can also be judged by the needed memory space. The memory is used in three ways during a calculation:

- the storage of the program;
- the building of data structures with the help of constructor functions;
- the storage of the still to be evaluated sub-expressions.

The memory usage for the data structures(lists, tuples, trees etc.) are built with the use of constructor functions. The heap consists of *cells*, in which the information is stored. [1] With each call of the constructor operator `:` two new cells are being used. Also when using self defined constructor functions cells are used (one for each parameter of the constructor function).

Sooner or later the whole heap will be used up. At that moment the *garbage collector* does its job. All cells which are not needed anymore are reclaimed for reuse. When calculating the expression `length (subs [1..8])` the sub-rows of `[1..8]` are computed and added. As soon as the sub-row is added, the row itself is not important anymore. With garbage collection that heap space will be reclaimed for reuse.

If during the calculation of an expression some garbage collection(s) took place, they will be reported after finishing the calculation:

```
? length (subs [1..8])
256
(4141 reductions, 9286 cells, 1 garbage collection)
? length (subs [1..10])
1024
(18487 reductions, 43093 cells, 6 garbage collections)
?
```

It is also possible to see each separate garbage collection. For this the option `g` must be enabled:

```
? :set +g
```

When initiating each garbage collection the interpreter will now report `{{Gc:`. After finishing each garbage collection the reclaimed cells are reported:

```
? length (subs [1..10])
{{Gc:7987}}{{Gc:7998}}{{Gc:7994}}{{Gc:7991}}{{Gc:7997}}{{Gc:8004}}1024
(18487 reductions, 43093 cells, 6 garbage collections)
```

If even after garbage collection there is still not enough memory available to build a new cell, the calculation is canceled with the error

```
ERROR: Garbage collection fails to reclaim sufficient space
```

The only thing left to do is to open Gofer with a larger heap. Opening Gofer with a heap of 50000 cells is done, provided that there is enough memory for such a large heap, by typing:

```
% gofer -h50000
```

Significant improvement in the efficiency of heap usage is almost impossible. The built data structures are often just necessary. The only way to avoid the use of constructor functions, and therefore the usage of the heap, is to use as many `@` patterns as possible, as described in section 4.1.3. page 70
The function `tails` was defined in three ways:

- without patterns

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

- with conventional patterns

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

[1] Depending on the used computer each cell costs 4 or 8 bytes in memory.

- with @ patterns

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

When applying this functions to the list [1..10] the time and memory usage is:

without patterns	210 reductions	569 cells
with conventional patterns	200 reductions	579 cells
with @ patterns	200 reductions	559 cells

From this shows again that the definition with @ patterns combines the advantages of the others.

The stack

The stack will be used by the interpreter to store the expression which has to be computed. The intermediate results are also stored on the stack. Each 'reduction' results a new intermediate result.

The stack space needed to store the intermediate results can be larger than the expression to be calculated and the result For example take a look at the intermediate results in the calculation of `foldr (+) 0 [1..5]`, which calculates the sum of the numbers 1 until 5:

```
foldr (+) 0 [1..5]
1+foldr (+) 0 [2..5]
1+(2+foldr (+) 0 [3..5])
1+(2+(3+foldr (+) 0 [4..5]))
1+(2+(3+(4+foldr (+) 0 [5])))
1+(2+(3+(4+(5+foldr (+) 0 []))))
1+(2+(3+(4+(5+0))))
1+(2+(3+(4+5)))
1+(2+(3+9))
1+(2+12)
1+14
15
```

The real calculation of the + operators starts after the stack has built up the whole expression `1+(2+(3+(4+(5+0))))`. In most cases you would not be bothered with that, but it gets annoying if you start adding real long lists:

```
? foldr (+) 0 [1..5000]
12502500
? foldr (+) 0 [1..6000]
ERROR: Control stack overflow
```

If you would use `foldl` instead of `foldr` the figures are added in another way. But here also a lot of stack space is required:

```
foldl (+) 0 [1..5]
foldl (+) (0+1) [2..5]
foldl (+) ((0+1)+2) [3..5]
foldl (+) (((0+1)+2)+3) [4..5]
foldl (+) ((((0+1)+2)+3)+4) [5]
foldl (+) (((((0+1)+2)+3)+4)+5) []
((((0+1)+2)+3)+4)+5
(((1+2)+3)+4)+5
((3+3)+4)+5
(6+4)+5
10+5
15
```

Because of the lazy evaluation principle the calculation of the parameters is postponed as long as possible. This is also the case for the second parameter of `fold`. In this case this is a little annoying, because eventually the expression is still evaluated. The postponing has been good for nothing, and it only costs a lot of stack space.

Without lazy evaluation the calculation would have been as follows; as soon as possible the second parameter of `foldl` is calculated:

```
foldl (+) 0 [1..5]
```

```
foldl (+) (0+1) [2..5]
foldl (+) 1 [2..5]
foldl (+) (1+2) [3..5]
foldl (+) 3 [3..5]
foldl (+) (3+3) [4..5]
foldl (+) 6 [4..5]
foldl (+) (6+4) [5]
foldl (+) 10 [5]
foldl (+) (10+5) []
foldl (+) 15 []
15
```

Now the used stack space is constant during the whole calculation.

How is it possible to get the interpreter to evaluate a function (in this case the partially parametrized function `foldl (+)`) not lazy (eager)? Especially for this purpose there is a built-in function, `strict`, The effect of `strict` is as follows:

```
strict f x  =  f x
```

So this function gets a function and a value as a parameter, and applies the function to that value. This would be a very silly function if that is the only effect of `strict`, but `strict` does more: it calculates its second parameter not lazy. Such a function could never been written yourself; `strict` is therefore built in.

With the help of the function `strict` each desired function can be calculated non-strict. In the example it would be convenient to have `foldl` calculate its *second* parameter non-lazy. That is to say, the partially parametrized function `fold (+)` has to be non-lazy in its first parameter. This is possible, by applying the function `strict` to it; we therefore write `strict (foldl (+))`. Note the extra pair of parentheses: `fold (+)` has to be passed as a whole to `strict`.

In the prelude a function `foldl'` is defined, which does the same as `foldl`, but then with a non-lazy second parameter. To compare them, we first recall the definition of `foldl`:

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

In the function `foldl'` the function `strict` is used in the recursive call like described:

```
foldl' f e []      =  e
foldl' f e (x:xs)  =  strict (foldl' f) (f e x) xs
```

Utilizing `foldl'` the stack space is constant, which makes it possible to compute the sum of very long lists without hesitation:

```
? foldl (+) 0 [1..6000]
ERROR: Control stack overflow
? foldl' (+) 0 [1..6000]
18003000
```

For operators like `+` and `*`, which need the value of both parameters to calculate their result, it is therefore recommended tu use `foldl'` instead of `foldr` or `foldl`. This will prevent any shortages of stack space. Therefore the prelude defines:

```
sum      =  foldl' (+) 0
product  =  foldl' (+) 0
```

But for the operator `&&` `foldr` is still used:

```
and      =  foldr (&&) True
```

This is because lazy evaluation makes it possible to prevent calculation of the second parameter if it is not needed. This means that `and` stops the calculation as soon as a value `False` is encountered. When using `foldl'` it would always traverse the whole list.

5.2 Laws

5.2.1 Mathematical laws

Mathematical functions have the pleasant property of not depending on the context of the calculation. The value of $2+3$ is always 5, regardless of this expression is part of the expression $4\times(2+3)$

or for example $(2+3)^2$.

Many operators satisfy certain laws. For instance, for all numbers x and y it holds that $x+y = y+x$. Some mathematical laws are:

commutative *law* for +	$x + y = y + x$
commutative law for ×	$x \times y = y \times x$
associative law for +	$x + (y + z) = (x + y) + z$
associative law for ×	$x \times (y \times z) = (x \times y) \times z$
distributive law	$x \times (y + z) = (x \times y) + (x \times z)$
law for repeated involution	$(x^y)^z = x^{(y \times z)}$

These kind of mathematical laws can be used well to transform expressions into expression which have the same value. By that you can derive new laws out of old ones. The well-known remarkable product $(a + b)^2 = a^2 + 2ab + b^2$ can be derived from the former laws:

$(a{+}b)^2$
$=$ (definition squared)
$(a{+}b) \times (a{+}b)$
$=$ (distributive law)
$((a{+}b) \times a) + (a{+}b) \times b$
$=$ (commutative law for × (twice))
$(a \times (a{+}b)) + (b \times (a{+}b))$
$=$ (distributive law (twice))
$(a{\times}a + a{\times}b) + (b{\times}a + b{\times}b)$
$=$ (associative law for +)
$a{\times}a + (a{\times}b + b{\times}a) + b{\times}b$
$=$ (definition square (twice))
$a^2 + (a{\times}b + b{\times}a) + b^2$
$=$ (commutative law for ×)
$a^2 + (a{\times}b + a{\times}b) + b^2$
$=$ (definition '(2×)')
$a^2 + 2{\times}a{\times}b + b^2$

In each branch of mathematics new functions and operators are being designed, for which some mathematical laws apply. For example, in the propositional logic the following rules hold to 'calculate' with Boolean values:

commutative law for ∧	$x \wedge y = y \wedge x$
associative law for ∧	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
distributive law	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
de Morgans Law	$\neg(x \wedge y) = \neg x \vee \neg y$
Howards Law	$(x \wedge y) \rightarrow z = x \rightarrow (y \rightarrow z)$

The nice thing of laws is that you can apply a number of them after each other, without having to bother about the meaning of the intermediate steps. For instance, you can prove the law $\neg((a \vee b) \vee c) \rightarrow \neg d = \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$ using the stated laws:

$\neg((a \vee b) \vee c) \rightarrow \neg d$
$=$ (de Morgans law)
$(\neg(a \vee b) \wedge \neg c) \rightarrow \neg d$
$=$ (de Morgans law)
$((\neg a \wedge \neg b) \wedge \neg c) \rightarrow \neg d$
$=$ (Howards law)
$(\neg a \wedge \neg b) \rightarrow (\neg c \rightarrow \neg d)$
$=$ (Howards law)
$\neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$

Even if you would not know what the meaning of ∧, ∨ and ¬ is, you can accept the validity of the new law, provided that you accept the validity of the used laws. This is because each equality is augmented with a hint which states which law is applied. Those hints are very important. If they are omited, the doubting reader will have to fund out him or herself which law has been used. The presence of the hints contributes an important amount to the 'readability' of a deduction.

5.2.2 Gofer laws

Gofer functions have the same property as mathematical functions: a call of a function with the same parameter results always the same value. If you replace somewhere in an expression a sub-expression with another equivalent sub-expression, then it does not influence the final result.

For all kinds of functions it is possible to formulate laws which they satisfy. For instance, for all functions `f` and all lists `xs`:

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

This is no *definition* of `map` or `.` (these are defined before); it is a *law* the functions seem to satisfy.

Laws for Gofer functions can be used while designing a program. You can make programs easier to read or faster with them. In the definition of the determinant function on matrices for instance, (see section 4.2.3) the following expression occurs: page 77

```
(altsum . zipWith (*) ry . map det . map Mat . gaps . transpose) rys
```

By applying the law stated above it shows that this can also be written as

```
(altsum . zipWith (*) ry . map (det.Mat) . gaps . transpose) rys
```

By using the laws you can 'calculate' with entire programs. Programs are transformed in that way into other programs. Just as with calculating with numbers or propositions it is again not necessary to understand all intermediate (or even the final) programs. If the initial program is right, the laws hold, and you do not make any mistakes with the applying of the laws, then the final program is guaranteed to be the same as the initial program.

A number of important laws which hold for the Gofer standard functions are:

- function composition is associative, so

  ```
  f . (g . h)  =  (f . g) . h
  ```
- `map` distributes over `++`, so

  ```
  map f (xs++ys)  =  map f xs ++ map f ys
  ```
- the generalization of this to one *list of* lists instead of *two* lists:

  ```
  map f . concat  =  concat . map (map f)
  ```
- a `map` of a composition is the composition of the `map`s:

  ```
  map (f.g)  =  map f . map g
  ```
- If `f` is associative (so `` x`f`(y`f`z) `` = `` (x`f`y)`f`z ``), and `e` is the neutral element of `f` (so `` x`f`e `` = `` e`f`x `` = `x` for all `x`), then holds:

  ```
  foldr f e xs  =  foldl f e xs
  ```
- If the initial value of `foldr` equals the neutral element of the operator, the `foldr` over a singleton list is the identity:

  ```
  foldr f e [x]  =  x
  ```
- An element in front of a list can be swapped with `map`, provided that you put the application of the function to that element in front:

  ```
  map f . (x:)  =  (f x :) . map f
  ```
- Each usage of `map` can also be written as a call of `foldr`:

  ```
  map f xs  =  foldr g [] ys  where g x ys = f x : ys
  ```

Many of these laws match what would be called 'programming tricks' in imperative languages. The fourth law in the list would be described as 'combination of two loops'. In imperative languages the program transformations matching the laws are not unconditionally usable: you must always keep in mind that some functions could have unexpected 'side effects'. In functional languages like Gofer these laws may *always* be applied; this is due to the fact that functions have the same value regardless of their context.

5.2.3 Proving laws

From a number of laws it is intuitively clear they hold. The validity of other laws is not clear at first sight. Mainly in the second case, but also in the first, it is useful to *prove* the law. The definitions of functions and previously proven laws can be used. In section 5.2.1 the laws $(a+b)^2 = a^2+2ab+b^2$ page 92

and $\neg((a \vee b) \vee c) \to \neg d = \neg a \to (\neg b \to (\neg c \to \neg d))$ were already proven.

The proving of laws for Gofer functions looks much alike. It is very practical to name those laws, enabling you to refer easily to them later (when proving other laws). The formulation of a law, including the proof, could look like:

> **Law** *foldr over a singleton-list*
> If `e` equals the neutral element of the operator `f`, then
>
> `foldr f e [x] = x`

Proof:

```
foldr f e [x]
=   (notation list-summary)
foldr f e (x:[])
=   (def. foldr)
f x (foldr f e [])
=   (def. foldr)
f x e
=   (precondition of the law)
x
```

If a law proves the equality of two *functions*, this can be proven by proving that the result of the function is equal for all possible parameters. Both functions can thus be applied to a variable `x`, after which equality is shown. This is the case in the next law:

> **Law** *function composition is associative*
> For all functions `f`, `g` and `h` of the right type:
>
> `f . ( g . h ) = ( f . g ) . h`

Proof:

```
(f . (g.h)) x
=   (def. (.))
f ((g.h) x)
=   (def. (.))
f (g (h x))
=   (def. (.))
(f.g) (h x)
=   (def. (.))
((f.g) . h) x
```

The reading of such a proof is not always as exiting as you would want it to be. When thinking of a proof yourself, you will find out that the writing of a proof is hard. Usually, the first few steps are not as hard, but mostly halfway you will get stuck. It can then be useful to work from the other side for a while. To show that a proof is constructed in this way, we will write it in two columns from now on:

	`f . ( g . h )`	`( f . g ) . h`
`x`	`(f . (g.h)) x`	`((f.g) . h) x`
	= (def. `(.)`)	= (def. `(.)`)
	`f ((g.h) x)`	`(f.g) (h x)`
	= (def. `(.)`)	= (def. `(.)`)
	`f (g (h x))`	`f (g (h x))`

In the two columns of this lay-out the two sides of the law are shown to be equal to the same expression. Two things equal to the same expression, are of course also equal to each other, what was to be proven. In the first column of the scheme we also write to which parameter (`x`) the left and right function are being applied.

This method of proof can also be used to prove another law:

> **Law** *map after (:)*
> For all values `x` and functions `f` of the right type:

```
map f . (x:)  =  ((f x):) . map f
```

Proof:

	`map f . (x:)`	`((f x):) . map f`

`xs`	`(map f . (x:)) xs`
	= (def. `(.)`)
	`map f ((x:) xs)`
	= (section notation)
	`map f (x:xs)`
	= (def. `map`)
	`f x : map f xs`
		`( ((f x):) . map f) xs`
		= (def. `(.)`)
		`((f x):) (map f xs)`
		= (section notation)
		`f x : map f xs`

Just as with the previous proof this proof could of course have been written as one long deduction too. However in this two column notation it is more clear how the proof was made: the two sides of the law are proven to be equal to a third expression, and therefore to each other.

5.2.4 Inductive proofs

Functions ol lists are mostly defined inductively. The function is defined separately for `[]`. Then the function is defined for the pattern `(x:xs)`, where the function may be called recursively on `xs`.

In the proof of laws where lists play a part, induction can also be used. To do this, the law is proven separately for the case of `[]`. Next, the law is proven for a list of the form `(x:xs)`. In this prove one may assume the law already holds for the list `xs`. An example of a law which can be proven inductively is the distribution of `map` over `++`.

Law *map after* `++`
For all functions `f` and all lists `xs` and `ys` with the right type:

```
map f (xs++ys)  =  map f xs ++ map f ys
```

Proof with induction to `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 this proof the first part of the law is formulated for the list `xs`. This is called the *induction hypothesis*. In the second part the law is proven for the empty list: so here `[]` is used where the original law said `xs`. In the third case the law is proven with `(x:xs)` in stead of `xs`. Although the last line of the two columns is not the same expression, the equality *does* hold, since in this part the induction hypothesis may be assumed.

In the law 'map after function composition' two functions are said to be equal. To prove this equality, the left and the right hand side are applied to a parameter `xs`. After that the proof is finished with induction to `xs`:

Law *map after function composition*
For all compositionable functions `f` and `g`:

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

Proof with induction to `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 this proof the right hand side of the hypothesis is first simplified, before the real induction starts; if not this step would have been needed in both parts of the inductive proof. This also happens in the proof of the next law:

Law *map after concat*
For all functions `f`:

```
map f . concat  =  concat . map (map f)
```

This is a generalization of the distribution law of `map` over `++`.

Proof with induction to `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)`
	= (law of distribution)
	`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)`

Again, the expressions of the last lines are equal because the assumption of the induction hypothesis. In this proof are another rule is used, except for the definition of functions and notations, namely the previously proven distribution law of `map` over `++`.

The distinction `[]/(x:xs)` is sometimes not enough to prove a law. The same holds by the way for the definition of functions. In the proof of the next law *three* cases are distinguished: the empty list, a singleton list and a list with at least two elements.

Law *law of duality*
If `f` is an associative operator (so `` x`f`(y`f`z) `` = `` (x`f`y)`f`z ``), and `e` is the neutral element of `f` (so `f x e` = `f e x` = `x` for all `x`) then:

```
foldr f e  =  foldl f e
```

Proof with induction to `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 neutral element) x	foldl f e [x] = (def. foldl) foldl f (e‘f‘x) [] = (def. foldl) e‘f‘x = (e neutral 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 associative) (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 associative) foldl f (e‘f‘(x1‘f‘x2)) xs = (def. foldl) foldl f e ((x1‘f‘x2):xs)

The last two lines are equal because the list `(x1‘f‘x2):xs` is shorter than `x1:x2:xs`. So the validity of the law may be assumed for this list.

5.2.5 Improving efficiency

Laws can be used to transform functions into more efficient functions. Two expressions of which equality is proven, need not to be calculated equally fast; in that case a slower definition can be replaced by a faster one. In this section two examples of this technique will be discussed:

- the `reverse` function will be improved from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$;
- the function of Fibonacci, which already has been improved from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$ will be improved even more to $\mathcal{O}(\log n)$.

reverse

To improve the function `Reverse` we prove three laws:

- Next to the relation proven in the previous section between `foldr` and `foldl` there is another connection: the *second law of duality*:

```
foldr f e (reverse xs)  =  foldl (flip f) e xs
```

- The proof of this law does not succeed at once. Another law is needed, which can be proven separately with induction:

```
foldr f e (xs++[y])  =  foldr f (f y e) xs
```

- About the easiest provable law with induction is:

```
foldr (:) []  =  id
```

We start with the third law. After that a proof of the auxiliary law, and concluding with the second law of duality itself. Finally, we will improve the `reverse` function with this law.

Law *foldr with constructor functions*
If the constructor functions of lists are passed to `foldr`, that is `(:)` and `[]`, the result will be the identity:

```
foldr (:) []  =  id
```

Proof with induction to `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`

Law *auxiliary law for the next law*

```
foldr f e (as++[b])  =  foldr f (f b e) as
```

Proof with induction to `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)`

Law *second law of duality*
For all functions `f`, values `e` and lists `xs` of the right type:

```
foldr f e (reverse xs)  =  foldl (flip f) e xs
```

Proof with induction to `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])`
	= (auxiliary law as stated above)
	`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`

The last expressions in this proof are equal due to the induction hypothesis. The induction hypothesis may be assumed for *all* `e`, so also with `f x e` for `e`. (The induction hypothesis may however only be assumed for fixed variables to which the induction is done (`xs`); otherwise the whole induction principle collapses.)

The function `reverse` is defined in section 3.1.2 as page 41

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

Or the equivalent

```
reverse  =  foldr post []
     where  post x xs = xs++[x]
```

When it is defined in this way the function costs $\mathcal{O}(n^2)$ time, where n is the length of the to be reversed list. This is because the operator `++` in `post` costs $\mathcal{O}(n)$ time, and this should be multiplied by the $\mathcal{O}(n)$ of the recursion (regardless whether it is hidden in `foldr`).

But from the recently proven laws can be deduced that:

```
reverse xs
=   (def. id)
id (reverse xs)
=   (foldr with constructor functions)
foldr (:) [] (reverse xs)
=   (second law of duality)
foldl (flip (:)) [] xs
```

The new definition

```
reverse  =  foldl (flip (:)) []
```

costs only $\mathcal{O}(n)$ time. This is because the operator used for `foldr`, `flip (:)`, costs only constant time.

Fibonacci

Also laws where natural numbers play a role, can sometimes be proven with induction. In that case the law is proven separately for the case 0, and then for the pattern $n{+}1$, where the law for the case n may already be used. In some proofs another induction scheme is used, like $0/1/n{+}2$; or $1/n{+}2$ if the law is not needed to hold for the case 0.

Induction over natural numbers can well be used to improve the efficiency of the Fibonacci function. The original definition was:

```
fib 0      =  0
fib 1      =  1
fib (n+2)  =  fib n + fib (n+1)
```

The needed time to calculate `fib n` is $\mathcal{O}(2^n)$. In section 5.1.3 it was already improved to $\mathcal{O}(n)$, but thanks to the next law an even better improvement will be possible. page 89

Law *Fibonacci by involution*
If $p = \frac{1}{2} + \frac{1}{2}\sqrt{5}$, $q = \frac{1}{2} - \frac{1}{2}\sqrt{5}$, and $c = 1/\sqrt{5}$ then:

```
fib n  =  c * (p^n - q^n)
```

The values p and q are the solutions of the equation $x^2 - x - 1 = 0$. That is why $p^2 = p + 1$ and $q^2 = q + 1$. Also, $p - q = \sqrt{5} = 1/c$. Using these prototypes we prove the hypothesis.

Proof of 'Fibonacci by involution' with induction to n:

n	`fib` n	$c \times (p^n - q^n)$
0	`fib` 0 = (def. `fib`) 0 = (def. $\times$) $c \times 0$	$c \times (p^0 - q^0)$ = (def. involution) $c \times (1 - 1)$ = (property $-$) $c \times 0$
1	`fib` 1 = (def. `fib`) 1 = (def. /) $c \times (1/c)$	$c \times (p^1 - q^1)$ = (def. involution) $c \times (p - q)$ = (property c) $c \times (1/c)$
$n{+}2$	`fib` $(n{+}2)$ = (def. `fib`) `fib` n `+` `fib` $(n{+}1)$	$c \times (p^{n+2} - q^{n+2})$ = (property involution) $c \times (p^n p^2 - q^n q^2)$ = (property p en q) $c \times (p^n(1{+}p) - q^n(1{+}q))$ = (distribution $\times$) $c \times ((p^n{+}p^{n+1}) - (q^n{+}q^{n+1}))$ = (commutativity en associativity +) $c \times ((p^n{-}q^n) + (p^{n+1}{-}q^{n+1}))$ = (distribution $\times$) $c \times (p^n{-}q^n) + c \times (p^{n+1}{-}q^{n+1})$

The remarkable thing about this law is that although the square roots the final answer is again whole. This law can be used to create a $\mathcal{O}(\log n)$ version of `fib`. Involution can be done in $\mathcal{O}(\log n)$ time, by using the halving method. By defining `fib` by

```
fib n  =  c * (p^n - q^n)
   where  c = 1/root5
          p = 0.5 * (1+root5)
          q = 0.5 * (1-root5)
          root5 = sqrt 5
```

`fib` can also be calculated in logarithmic time. The final optimalization is possible by noting that $|q| < 1$, so q^n can be neglected, especially for large n. It suffices to round $c \times p^n$ to the nearest integer.

5.2.6 properties of functions

Except for improving the efficiency of some functions laws are also useful to gain more insight in the operation of some functions. For functions resulting a list it is for instance interesting to know how the length depends to the result of the parameter of the function. Below four laws are discussed about the length of the result of a function. After that three laws about the sum of the resulting list of a function will be discussed. The laws are used after that to be able to say something about the length of the result of combinatoric functions.

Laws about length

In this section the function `length` will be written as `len` (to save ink, and thus being more aware of the surroundings).

Law *length after (:)*
By adding one element at the front of a list the length will increase exactly with one:

```
len . (x:)  =  (1+) . len
```

The law follows almost directly from the definition. No induction is needed:

	`len . (x:)`	`(1+) . len`

`xs`	`(len . (x:)) xs`
	= (def. `(.)`)
	`len (x:xs)`
	= (def. `len`)
	`1 + len xs`
		`((1+) . len) xs`
		= (def. `(.)`)
		`1 + len xs`

Law *length after map*
By `mapping` a function over a list the length remains unchanged:

```
len . map f  =  len
```

The proof is with induction to `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`

Law *length after* `++`
The length of the concatenation of two lists is the sum of the lengths of those lists:

```
len (xs++ys)  =  len xs + len ys
```

Proof with induction to `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 = (associativity +) 1 + (len xs + len ys)

The next law is a generalization of this: in this law a list of lists is used, instead of two lists, and the operator `+` can therefore be changed for `sum`.

Law *length after concatenation*
The length of a concatenation of a list of lists is the sum of the lengths of all those lists:

```
len . concat  =  sum . map len
```

Proof with induction to `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) = (length after ++) 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)

Laws concerning sum

Just as with `len` there are two laws about `sum` describing the distribution over concatenation (of two lists or of a list of lists).

Law *sum after* `++`
The sum of the concatenation of two lists equals the added sums of the lists:

```
sum (xs++ys)  =  sum xs + sum ys
```

Proof with induction to `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 = (associativity +) x + (sum xs + sum ys)

Just as with `len` this law is used in te proof of the generalization to lists of lists.

Law *sum after concatenation*
The sum of the concatenation of a list of lists is the sum of the sums of those lists:

```
sum . concat  =  sum . map sum
```

Proof with induction to `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 after `++`)
	`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)`

There is no law for the `sum` of a `map` on a list, which was the case for `len`. This is clear from the next example: the sum of the squares of a number does not equal the square of the sum. But it is possible to formulate a law for the case when the `mapped` function is the function `(1+)`.

Law *sum after map-plus-1*
The sum of a list increased numbers is the sum of the original list plus the length of it:

```
sum (map (1+) xs)  =  len xs + sum xs
```

Proof with induction to `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)`
		= (`+` associative and commutative)
		`(1+x) + (len xs + sum xs)`

In exercise 5.4 this law is generalized to an arbitrary linear function. page 111

Laws about combinatoric functions

With the help of a number of the laws stated above laws can be proven about combinatoric functions from section 4.1. We prove for `inits`, `segs` and `combs` a law describing the number of elements of the result (see also exercise 4.3): page 67 page 83

```
len . inits    =  (1+) . len
len . segs     =  f    . len     where f n = 1 + (n*n+n)/2
len . combs k  =  (`choose` k) . len
```

Law *number of initial segments*
The number of initial segments of a lists is one more than the number of elements in the list:

```
len . inits  =  (1+) . len
```

Proof with induction to `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))`
	= (length after map)
	`1 + len (inits xs)`
		`1 + len (x:xs)`
		= (def. `len`)
		`1 + (1+len xs)`

Law *number of segments*
The number of segments of a list is a quadratic function of the number of elements of the list:

```
len . segs  =  f . len  where f n = 1 + (n*n+n)/2
```

Proof with induction to `xs`. We write n for `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`)	= (algebra)
	`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`)
	= (length after `++`)	$1+((1+n)^2+(1+n))/2$
	`len (segs xs) +`	= (remarkable product)
	`len (map (x:)(inits xs))`	$1+((1+2n+n^2)+(1+n))/2$
	= (length after map)	= (+ associative and commutative)
	`len (segs xs) + len (inits xs)`	$1+((n^2+n)+(2+2n))/2$
	= (number of initial segments)	= (distribution /)
	`len (segs xs) + 1 + len xs`	$1+(n^2+n)/2+1+n$

The definition of `choose` in section 1.2.2 was not complete. The complete definition is: page 4

```
choose n k  |  n>=k  =  fac n / (fac k * fac (n-k))
            |  n<k   =  0
```

This function plays a role in the next law.

Law *number of combinations*
The number of combinations of k elements from a list is the binomial coefficient 'length of the list `choose` k':

```
len . combs k  =  (`choose` k) . len
```

In the proof the usual mathematical notation $n!$ will be used for `fac n`, and $\binom{n}{k}$ for ``n `choose` k``. We write n for `len xs`. The proof is done with induction to `k` (with the cases `0` and `k+1`). The proof of the induction stap (case `k+1`) is done with induction again, this time to `xs` (with the

cases [] and x:xs). This induction structure matches that of the definition of combs.

	len . combs k	(‘choose‘ k) . len
k xs	len (combs k xs)	$\binom{\texttt{len xs}}{k}$
0 xs	len (combs 0 xs) = (def. combs) len [[]] = (def. len) 1 + len [] = (def. len) 1 + 0 = (property +) 1	$\binom{\texttt{len xs}}{0}$ = (def. choose) $\frac{n!}{(n-0)!*0!}$ = (def. fac en $-$) $\frac{n!}{n!*1}$ = (def. / en $*$) 1
k+1 []	len (combs (k+1) []) = (def. combs) len [] = (def. len) 0	$\binom{\texttt{len []}}{k+1}$ = (def. len) $\binom{0}{k+1}$ = (def. choose) 0
k+1 x:xs	len (combs (k+1) (x:xs)) = (def. combs) len (map (x:)(combs k xs) ++ combs (k+1) xs) = (length after ++) len (map (x:)(combs k xs)) + len (combs (k+1) xs) = (length after map) len (combs k xs) + len (combs (k+1) xs)	$\binom{\texttt{len (x:xs)}}{k+1}$ = (def. len) $\binom{n+1}{k+1}$ = (def. choose ($n \geq k$)) $\frac{(n+1)!}{((n+1)-(k+1))! * (k+1)!}$ = (numerator: def. fac; denominator: algebra) $\frac{(n+1)*n!}{(n-k)!*(k+1)!}$ = (numerator: algebra; denominator: def. fac ($n>k$)) $\frac{(k+1) * n!}{(n-k)! * (k+1) * k!} + \frac{(n-k) * n!}{(n-k) * (n-k-1)! * (k+1)!}$ = (divide ($n>k$)) $\frac{n!}{(n-k)! * k!} + \frac{n!}{(n-(k+1))! * (k+1)!}$ = (def. choose) $\binom{n}{k} + \binom{n}{k+1}$

The right column of the induction step proof is only valid for $n > k$. For $n = k$ the proof is:

$$\binom{n+1}{k+1} = 1 = 1+0 = \binom{n}{k} + \binom{n}{k+1}$$

For $n < k$ the proof is:

$$\binom{n+1}{k+1} = 0 = 0+0 = \binom{n}{k} + \binom{n}{k+1}$$

5.2.7 Polymorphism

The next laws hold for all functions f:

```
inits . map f  =  map (map f) . inits
segs  . map f  =  map (map f) . segs
subs  . map f  =  map (map f) . subs
perms . map f  =  map (map f) . perms
```

It is intuitively clear that these laws hold. Imagine for example calculating the initial segments of a list with inits, after calculating the f-value of all elements (by map f). You could have calculated the inits first too, and then have applied map f to *each* resulting initial segment afterwards. In

the last case (represented by the right hand side of the law) you will have to apply `f` to the elements of the list of lists, which explains the double `map`.

We prove the first of the mentioned laws; the others are equally difficult.

Law *initial segments after map*
For all functions `f` on lists:

```
inits . map f  =  map (map f) . inits
```

Proof with induction to `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 after function composition)
		`[] : map (map f.(x:)) (inits xs)`
		= (map after (:))
		`[] : map ((f x:).map f) (inits xs)`
		= (map after function composition)
		`[] : map (f x:)`
		`(map (map f)(inits xs))`

A proof as this one is roughly the same for *each* combinatoric function. That is, if `combinat` is a combinatoric function, then for all function `f`

```
combinat . map f  =  map (map f) . combinat
```

This is due to the definition of 'combinatoric function': functions from lists to lists of lists, which are not allowed to make use of the specific prototypes of the elements. Another way of putting it: combinatoric functions are polymorphic functions with the type

```
combinat  ::  [a] -> [[a]]
```

It is even true that the last law may be used as a *definition* of combinatoric functions. So: a function `combinat` is called 'combinatoric', if, for all functions `f`:

```
combinat . map f  =  map (map f) . combinat
```

With such a definition, which gives a clear description with the help of a law, you can usually do more than the somewhat vague description 'are not allowed to make use of the specific prototypes of the elements', used in section 4.1. page 67

There are laws which look like this 'law of the combinatoric functions'. In section 5.2.4 for example, page 96 the law 'map after concat' is proved. This law states that for all functions `f`:

```
map f . concat  =  concat . map (map f)
```

Of course this law can also be read the other way around. Then it says that:

```
concat . map (map f)  =  map f . concat
```

In this form the law shows resemblance with the law of the combinatoric functions. The only difference is that the 'double map' is on the other side. This is not very hard to understand, because the type of `concat` is:

```
concat  ::  [[a]] -> [a]
```

Normally the number of list-levels increases when using combinatoric functions, with `concat` this decreases. The double map must therefore be applied *before* `concat` is being applied; the single

map *after* that.

The function `concat` is no combinatoric function, for the sole reason that it does not satisfy the law for combinatoric functions. It still is a *polymorphic* function. In section 1.5.3 a polymorphic function was defined as 'a function with a type where type variables are being used'. Just as with the notion of 'combinatoric function' the concept 'polymorphic function' can be defined with the help of a law:

page 16

A function `poly` with lists is called a *polymorphic* function if for all functions `f`:

$$\texttt{poly . } \underbrace{\texttt{map (...(map f))}}_{n\ \texttt{map's}} = \underbrace{\texttt{map (...(map f))}}_{k\ \texttt{map's}} \texttt{ . poly}$$

Then this function will have the type:

$$\texttt{poly :: } \underbrace{\texttt{[...[a]...]}}_{n \text{ dimensional list}} \texttt{ -> } \underbrace{\texttt{[...[a]...]}}_{k \text{ dimensional list}}$$

All combinatoric functions are polymorphic, since the law for combinatoric functions is a special case of the law for polymorphic functions, with $n = 1$ and $k = 2$. The function `concat` is also polymorphic: the law 'map after concat' has the desired form with $n = 2$ en $k = 1$.

Also for other data structures than lists (e.g. tuple, or trees) the notion of 'polymorphic function' can defined with the help of a law. Then there will be an equivalent version of `map` on the data structure, which can be used in the law instead of `map`.[2]

5.2.8 Proofs of mathematical laws

In section 5.2.1 a number of mathematical laws is mentioned, like 'multiplication is associative'. These laws can also be proven. When proving a law where some kind of function plays a role, the definition of this function is needed. However, until now we have not given a definition of addition and multiplication; these functions were regarded as 'built in'.

page 92

Theoretically speaking it is not necessary that the numbers, the natural numbers, that is, are built in Gofer. They can be defined by means of a data declaration. In this section the definition of the type `Nat` (the natural numbers) will be given, for two reasons:

- to show the power of the data declaration mechanism (you can even define the natural numbers!);
- to define mathematical operators with induction, after which the mathematical laws can be proven with induction.

In practice you'd better not use the numbers defined in this way, because the mathematical operations will not be very efficient (compared to the built-in operators). Still, the definition is very useful to prove the laws.

The definition of a natural number with a data declaration is done according to a procedure thought of in the previous century by Giusseppe Peanp (although he did not use our notation). The data type `Nat` (for 'natural number') is:

```
data Nat  =  Zero
          |  Succ Nat
```

So a natural number is either the number `Zero`, or it is an encapsulation of a natural number by the constructor function `Succ`. Each natural number can be transformed by applying `Succ` a number of times to `Zero`. In this way we can define:

```
one    =  Succ Zero
two    =  Succ (Succ Zero)
three  =  Succ (Succ (Succ Zero))
four   =  Succ (Succ (Succ (Succ Zero)))
```

[2] The branch of mathematics applying this construction is called 'category theory'. In the category theory a function satisfying this law is called a 'natural transformation'.

This might be a somewhat crazy notation compared to 1,2,3 and 4, but when defining functions and proving laws you will not be bothered by it.

The function 'plus' can now be defined with induction to one of the two parameters, for example the left:

```
Zero   + y  =  y
Succ x + y  =  Succ (x+y)
```

In the second line the to be defined function is called recursively. This is permitted, because `x` is a smaller data structure than `Succ x`. With the help of the plus function the multiplication function can also be defined recursively:

```
Zero    * y  =  Zero
Succ x * y  =  y + (x*y)
```

Here the to be defined function is called recursively too with a smaller parameter. With the help of this function the involution function can be defined, this time with induction to the second parameter:

```
x ^ Zero     =  Succ Zero
x ^ Succ y  =  x * (x^y)
```

Having defined the operators, it is possible to prove the mathematical laws. This has to be done in the right order, because the proof of some laws requires other laws. All proofs are done with induction to one of the variables. Some are used that frequently they are named; some are only proven because they are needed in the proof of other laws. The proofs are not difficult. The only hard thing is, to not use a not yet proven law during the proving because 'it is of course correct' – you might as well quit immediately.

Maybe this is a little redundant, but it is kind of neat to see that the known laws indeed can be proved.

These are the laws we will prove:

1.	`x + Zero`	=	`x`	
2.	`x + Succ y`	=	`Succ (x+y)`	
3.	`x + y`	=	`y + x`	`+` is commutative
4.	`(x+y) + z`	=	`x + (y+z)`	`+` is associative
5.	`x * Zero`	=	`Zero`	
6.	`x * Succ y`	=	`x + (x*y)`	
7.	`x * y`	=	`y * x`	`*` is commutative
8.	`x * (y+z)`	=	`x*y + x*z`	`*` distributes left over `+`
9.	`(y+z) * x`	=	`y*x + z*x`	`*` distributes right over `+`
10.	`(x*y) * z`	=	`x * (y*z)`	`*` is associative
11.	`x ^ (y+z)`	=	`x^y * x^z`	
12.	`(x*y)^z`	=	`x^z * y^z`	
13.	`(x^y)^z`	=	`x ^ (y*z)`	repeated involution

Proof of law 1, with induction to `x`:

`x`	`x + Zero`	`x`

`Zero`	`Zero + Zero`
	= (def. `+`)
	`Zero`
		`Zero`

`Succ x`	`Succ x + Zero`
	= (def. `+`)
	`Succ (x+Zero)`
		`Succ x`

Proof of law 2, with induction to `x`:

`x`	`x + Succ y`	`Succ (x+y)`

`Zero`	`Zero + Succ y`
	= (def. `+`)
	`Succ y`
		`Succ (Zero+y)`
		= (def. `+`)
		`Succ y`

`Succ x`	`Succ x + Succ y`
	= (def. `+`)
	`Succ (x + Succ y)`
		`Succ (Succ x + y)`
		= (def. `+`)
		`Succ (Succ (x+y))`

Proof of law 3 (plus is commutative), with induction to `x`:

x	x + y	y + x
Zero	Zero + y = (def. +) y	y + Zero = (law 1) y
Succ x	Succ x + y = (def. +) Succ (x+y)	y + Succ x = (law 2) Succ (y+x)

Proof of law 4 (plus is associative), with induction to x:

x	(x+y) + z	x + (y+z)
Zero	(Zero+y) + z = (def. +) y + z	Zero + (y+z) = (def. +) y + z
Succ x	(Succ x + y) + z = (def. +) Succ (x+y) + z = (def. +) Succ ((x+y) + z)	Succ x + (y+z) = (def. +) Succ (x + (y+z))

Proof of law 5, with induction to x:

x	x * Zero	Zero
Zero	Zero * Zero = (def. *) Zero	Zero
Succ x	Succ x * Zero = (def. *) Zero + (x*Zero)	Zero = (def. +) Zero+Zero

Proof of law 6, with induction to x:

x	x * Succ y	x + (x*y)
Zero	Zero * Succ y = (def. *) Zero	Zero + Zero*y = (def. *) Zero+Zero = (def. +) Zero
Succ x	Succ x * Succ y = (def. *) Succ y + (x*Succ y) = (def. +) Succ (y + (x*Succ y))	Succ x + (Succ x * y) = (def. *) Succ x + (y + (x*y)) = (def. +) Succ (x + (y + (x*y))) = (+ associative) Succ ((x + y) + (x*y)) = (+ commutative) Succ ((y + x) + (x*y)) = (+ associative) Succ (y + (x + (x*y)))

Proof of law 7 (* is commutative), with induction to x:

x	x * y	y * x
Zero	Zero * y = (def. *) Zero	y * Zero = (law 5) Zero
Succ x	Succ x * y = (def. *) y + (x*y)	y * Succ x = (law 6) y + (y*x)

Proof of law 8 (* distributes left over +), with induction to x:

x	x * (y+z)	x*y + x*z
Zero	Zero * (y+z) = (def. *) Zero	Zero*y + Zero*z = (def. *) Zero + Zero = (def. +) Zero
Succ x	Succ x * (y+z) = (def. *) (y+z) + (x*(y+z))	(Succ x*y) + (Succ x*z) = (def. *) (y+x*y) + (z + x*z) = (+ associative) ((y+x*y)+z) + x*z = (+ associative) (y+(x*y+z)) + x*z = (+ commutative) (y+(z+x*y)) + x*z = (+ associative) ((y+z)+x*y) + x*z = (+ associative) (y+z) + (x*y + x*z)

Proof of law 9 (* distributes right over +):

	(y+z) * x = (* commutative) x * (y+z) = (law 8) x*y + x*z	y*x + z*x = (* commutative) x*y + x*z

Proof of law 10 (* is associative), with induction to x:

x	(x*y) * z	x * (y*z)
Zero	(Zero*y) * z = (def. *) Zero * z = (def. *) Zero	Zero * (y*z) = (def. *) Zero
Succ x	(Succ x*y) * z = (def. *) (y+(x*y)) * z = (law 9) (y*z) + ((x*y)*z)	Succ x * (y*z) = (def. *) (y*z) + (x*(y*z))

Proof of law 11, with induction to y:

y	x^(y+z)	x^y * x^z
Zero	x^(Zero+z) = (def. +) x^z	x^Zero * x^z = (def. ^) Succ Zero * x^z = (def. *) x^z + Zero*x^z = (def. *) x^z + Zero = (law 1) x^z
Succ y	x ^ (Succ y+z) = (def. +) x ^ (Succ (y+z)) = (def. ^) x * x^(y+z)	x^Succ y * x^z = (def. ^) (x*x^y) * x^z = (* associative) x * (x^y * x^z)

The proof of law 12 and law 13 will be left as an exercise to the reader.

Exercises

5.1 Which version of segs is more efficient: the function of section 4.1.1 or that from exercise 4.1? page 68 page 83

5.2 Take a look at the (third) scheme in section 5.1.2. Mark at the optimalization methods **b** to **f** page 87

in section 5.1.3 at which line of the scheme the algorithms are before and after optimalization. page 87

5.3 The function `concat` can be defined by

```
concat xss  =  fold (++) [] xss
```

Here for `fold` one of the three functions`foldr`, `foldl` or `foldl'` can be used. Discuss the effect of the choice for the efficiency, both for the case that `xss` contains only finite lists, as for the case there are also infinite lists.

5.4 In section 5.2.6 the law page 103

```
sum (map (1+) xs)  =  len xs + sum xs
```

was proven. Formulate the same law for an arbitrary linear function instead of `(1+)`, so

```
sum (map ((k+).(n*)) xs)  =  ...
```

Prove the formulated law.

5.5 Prove the next law:

Law *fold after concatenation*
Let `op` be an associative operator, then:

```
foldr op e . concat  =  foldr op e . map (foldr op e)
```

5.6 Design a function `g` and a value `e` with

```
map f  =  foldr g e
```

Prove the equality for the found `g` and `e`.

5.7 Prove the following law:

```
len . subs  =  (2^) . len
```

5.8 Prove that `subs` is a combinatorial function.

5.9 Prove law 12 and law 13 from section 5.2.8. page 107

Appendix A

Exercises

A.1 Complex numbers

Problem

In this exercise two types will be defined with which *complex numbers* can be represented. Next a number of functions have to be defined, working on values of this type. All things which have to be defined are marked in the next text by a •.

The types

A complex number is a number of the form $a + bi$, where a and b are real numbers, and i is a value with the property $i^2 = -1$. This representation is called the *cartesian* representation: a complex number can be interpreted as a location (a, b) in the plane.

- Define a type `Cart` which represents a complex number as a tuple of two real numbers. Turn it into a 'protected type' like described on page 90.

Each point (a, b) in the plane is described uniquely by the distance r to the origin, and the angle φ of the x axis with the line from the origin to the point (φ in radials, $0 \leq \varphi \leq 2\pi$). Thus, a complex number can also be described by the real numbers r and φ. This is called the *polar coordinates* representation.

- Define a type `Polar` with which complex numbers are represented in polar coordinates. (This representation is the same as `Cart`, so it is sensible to use protected types).

Naming

It is important that you name all functions exactly as described below. The programs will be automatically tested by a program which assumes that these functions are indeed called that way. Next to that you may (optionally) define some extra functions. The names of the functions which have to be defined are systematically composed.

Conversion functions

- Write four conversion functions with which integers and real numbers from both representations of complex numbers can be converted to each other. Also write two functions converting both representations to each other:

```
float'cart    ::  Float -> Cart
float'polar   ::  Float -> Polar
int'cart      ::  Int   -> Cart
int'polar     ::  Int   -> Polar
cart'polar    ::  Cart  -> Polar
polar'cart    ::  Polar -> Cart
```

Basic functions

In the Cartesian representation complex numbers can be added by adding the x and y coordinates pairwise. They can also be multiplied by removing the parenthesis in $(a + bi) * (c + Tu)$, and using the property that $i^2 = -1$. For the division of cartesian complex numbers, see exercise 3.7.

- Write the four mathematical functions for `Cart`:

```
cart'add   ::  Cart -> Cart -> Cart
cart'sub   ::  Cart -> Cart -> Cart
cart'mul   ::  Cart -> Cart -> Cart
cart'div   ::  Cart -> Cart -> Cart
```

In the polar representation complex numbers can be multiplied to multiply their distance to the origin (as real numbers), and adding the φs. Do devide the complex numbers: divide the distances and and subtract the angles. Adding and subtracting `Polar` complex numbers is not easy; the easiest way to Th that is to convert them to `Cart`, perform the operation and finally transforming them back.

- Write the four mathematical functions for `Polar`

```
polar'add   ::  Polar -> Polar -> Polar
polar'sub   ::  Polar -> Polar -> Polar
polar'mul   ::  Polar -> Polar -> Polar
polar'div   ::  Polar -> Polar -> Polar
```

Involution with a natural number as power can be executed by repeated multiplication.

- Write the involution function for both representations:

```
cart'power   ::  Cart -> Int -> Cart
polar'power  ::  Polar -> Int -> Polar
```

Other Functions

The nth power root can be calculated the easiest in the polar representation: The real $\sqrt[n]{\ }$ is performed on the distance to the origin, and the angle φ is divided by n. (Check that if you raise the outcome to the n-th power, you will get the original back). But there are more solutions: all numbers with angle $\frac{\varphi}{n} + \frac{k*2\pi}{n}$ for $k \in \{0..n-1\}$ satisfy. So there are always n solutions of the complex n-th power root.

For the function in the Cartesian representation you can convert back and forth to the polar representation.

- Write two functions which both result all the solutions to the n-th power root of a complex number. Also write two functions which calculate the square root.

```
cart'root   ::  Int -> Cart  -> [Cart]
polar'root  ::  Int -> Polar -> [Polar]
cart'sqrt   ::         Cart  -> [Cart]
polar'sqrt  ::         Polar -> [Polar]
```

The exponential function can be calculated by the sequence

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

using this you can define the sine and cosine for the complex numbers, using the formulas

$$\begin{aligned} \cos x &= (e^{ix} + e^{-ix})/2 \\ \sin x &= (e^{ix} - e^{-ix})/2i \end{aligned}$$

- Write the functions

```
cart'exp   ::  Cart  -> Cart
polar'exp  ::  Polar -> polar
```

  by adding ten terms of the sequence. Next write the functions

```
cart'sin   ::  Cart  -> Cart
polar'sin  ::  Polar -> Polar
cart'cos   ::  Cart  -> Cart
polar'cos  ::  Polar -> Polar
```

An application

- Finally, write two functions which take three complex numbers a, b and c and result the two complex solutions to $ax^2 + bx + c = 0$ (using the abc-formula):

```
cart'sol2   ::  Cart  -> Cart  -> Cart  -> [Cart]
polar'sol2  ::  Polar -> Polar -> Polar -> [Polar]
```

A.2 Texts

A value of the type `String` can be used excellently to store a sequence of characters. However, such a text does not really preserve the two dimensional character of a text. When manipulating texts mostly the two dimensions are of some importance. For instance, when laying out a news paper you will need operations like 'put this text next to that text, and the whole above this picture'.

This exercise consists of the designing of a number of functions which operate on a type `Tekst`[1] The second part of the exercise consists of the application of these functions in a concrete problem: making a calendar of a given year.

A two dimensional text will be represented as alist of Strings: each string of the text is one element of this list. The type definition is:

```
type Tekst = [[Char]]
```

We are going to always keep the text rectangular, so all the lines of the text have to be equally long. The functions working on texts may assume this property without further checking, but they have to maintain this property.

Functions on texts

Next are the types of functions working on texts, with a short description. The exercise is to write these functions.

```
empty :: Tekst
```

It is always useful to have a basic element available: `empty` is a text with 0 lines.

```
asChars :: Tekst -> [Char]
```

The function `asChars` can be used to prepare a `Tekst` for display on the screen. For this purpose, all the lines are concatenated, separated by newline characters.

```
width   :: Tekst -> Int
height  :: Tekst -> Int
```

These functions determine the width, respectively the height of a text.

```
maxWidht   :: [Tekst] -> Int
maxHeight  :: [Tekst] -> Int
```

These functions determine the maximum width, respectively the height of a number of texts.

```
makeWide  :: Int -> Tekst -> Tekst
makeHigh  :: Int -> Tekst -> Tekst
```

These functions result a new text with the given width, respectively height. If the given text is to wide (high), the end of the lines (lines of the end) are truncated; if the text is not wide (high) enough, then spaces (empty lines) are appended.

```
tekst :: Int -> Int -> Tekst -> Tekst
```

This function makes that a text gets the given width (first parameter) and height (second parameter).

```
nextTo :: Tekst -> Tekst -> Tekst
```

This function puts two texts next to each other, and makes one big text of it. You may assume the text are of equal height

```
above :: Tekst -> Tekst -> Tekst
```

This function places one text above the other, and makes one large text of it. You may assume the texts are equally wide.

```
row    :: [Tekst] -> Tekst
stack  :: [Tekst] -> Tekst
```

These functions put a list of texts next to, respectively above each other. The texts in the lists are not guaranteed to be of same height respectively width!

```
group :: Int -> [a] -> [[a]]
```

[1] This is no typing error; `Text` with an 'x' is already defined in the prelude (although it is *not* the implementation of this exercise).

This function groups the elements of a list in sub-lists of shown length. The last sub-list may be shorter, if desired. For example `group 2 [1..5]` results the list of lists `[[1,2],[3,4],[5]]`.

```
hblock :: Int -> [Tekst] -> Tekst
vblock :: Int -> [Tekst] -> Tekst
```

These functions make a number of small texts into one large text, where `Int` denotes the number of small text next to each other (with `hblock`), respectively above each other (with `vblock`).

Calendar functions

Write, using the text manipulation functions, two functions:

```
month :: Int -> Int -> Tekst
year  :: Int -> Tekst
```

which produce the calendar of a given month, respectively year. The functions need not to work for years before 1753.

The function `month` gets as a first parameter a year, and as a second parameter the number of a month. The resulting text contains as a first line the name of the month and the year, and below that in vertical columns the weeks, preceded by the abbreviations of the days, starting with Monday.

An example application is:

```
? asChars (month 1994 10)
October 1994
Mo     3  10 17 24 31
Tu     4  11 18 25
We     5  12 19 26
Th     6  13 20 27
Fr     7  14 21 28
Sa  1  8  15 22 29
Su  2  9  16 23 30
```

The function `year` gives the calendar of a whole year, where the calendars of January, February and March have to be next to each other, below that the calendars of April, May and June, etc:

```
? asChars (year 1994)

January 1994             February 1994            March 1994
Mo     3  10 17 24 31    Mo     7  14 21 28       Mo     7  14 21 28
Tu     4  11 18 25       Tu  1  8  15 22          Tu  1  8  15 22 29
We     5  12 19 26       We  2  9  16 23          We  2  9  16 23 30
Th     6  13 20 27       Th  3  10 17 24          Th  3  10 17 24 31
Fr     7  14 21 28       Fr  4  11 18 25          Fr  4  11 18 25
Sa  1  8  15 22 29       Sa  5  12 19 26          Sa  5  12 19 26
Su  2  9  16 23 30       Su  6  13 20 27          Su  6  13 20 27
April 1994               May 1994                 June 1994
Mo     4  11 18 25       Mo     2  9  16 23 30    Mo     6  13 20 27
Tu     5  12 19 26       Tu     3  10 17 24 31    Tu     7  14 21 28
We     6  13 20 27       We     4  11 18 25       We  1  8  15 22 29
Th     7  14 21 28       Th     5  12 19 26       Th  2  9  16 23 30
Fr  1  8  15 22 29       Fr     6  13 20 27       Fr  3  10 17 24
Sa  2  9  16 23 30       Sa     7  14 21 28       Sa  4  11 18 25
Su  3  10 17 24          Su  1  8  15 22 29       Su  5  12 19 26
July 1994                August 1994              September 1994
Mo     4  11 18 25       Mo  1  8  15 22 29       Mo     5  12 19 26
Tu     5  12 19 26       Tu  2  9  16 23 30       Tu     6  13 20 27
We     6  13 20 27       We  3  10 17 24 31       We     7  14 21 28
Th     7  14 21 28       Th  4  11 18 25          Th  1  8  15 22 29
Fr  1  8  15 22 29       Fr  5  12 19 26          Fr  2  9  16 23 30
Sa  2  9  16 23 30       Sa  6  13 20 27          Sa  3  10 17 24
Su  3  10 17 24 31       Su  7  14 21 28          Su  4  11 18 25
October 1994             November 1994            December 1994
Mo     3  10 17 24 31    Mo     7  14 21 28       Mo     5  12 19 26
Tu     4  11 18 25       Tu  1  8  15 22 29       Tu     6  13 20 27
We     5  12 19 26       We  2  9  16 23 30       We     7  14 21 28
```

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

A.3 Formula manipulation

Problem

In this exercise two types will be defined with which *formulas can be manipulated.* The most important functions which have to be written are *symbolic differentiation* and calculating the so-called *additive normal form.* The parts of the exercise are marked with a •.

The type 'Expression'

In the reader a data structure is given to describe a *polynomial*, and a number of functions to manipulate polynomials. A polynomial is a special case of a *mathematical expression.* A mathematical expression is described by this data definition:

```
data Expr  =  Con Int
           |  Var Char
           |  Expr :+: Expr
           |  Expr :-: Expr
           |  Expr :*: Expr
           |  Expr :^: Expr
```

In this definition `Con`, `Var`, `:+:`, `:-:`, `:*:` and `:^:` are the constructors. With this data definitions we sort of define a small programming language. In this language an expression is a constant integer, a variable (which has to consist of *one* letter), or two expressions added, subtracted, multiplied or an expression raised to the power of another expression. The language does not support division.

Examples of mathematical expressions with their representation as a data structure are:

expression	data structure
$3x + 5$	`Con 3 :*: Var 'x' :+: Con 5`
$(x + y)^2$	`(Var 'x' :+: Var 'y') :^: Con 2`
3	`Con 3`

- Define the type `Expr` (just copy it). Use exactly the same names as in the example, since the 'customer' wishes to use them in that way. Supply the four constructor operators of the same priority as the built in operators `+`, `-`, `*` and `^`.

Evaluations of an expression

An *environment* maps letters to numbers:

```
type Env = [(Char,Int)]
```

Given an environment you can determine the value of an expression (if all variables from the expression are defined in the environment).

- Write a function
  ```
  eval :: Expr -> Env -> Int
  ```
  which calculates the value of an expression in a given environment.

Example of a use of this function:

```
? eval  (Con 3 :*: Var 'x' :+: Con 5)  [('x',2),('y',4)]
11
```

If the expression contains variables which are not in the environment the function needs not to work if these variables are really needed for the result. However, if with multiplication the left parameter has the value zero, or with involution the right, then the function still must give a result, even if the other parameter contains variables which are not bound in the environment. For example:

```
? eval  (Con 0 :*: Var 'x')  []
0
? eval  (Con 3 :*: Var 'x')  []
{...}
```

Symbolic differentiation

For each expression the *derived* expression can be determined with differentiation 'to' a certain variable. For instance, the derivative of the expression $3x^2$ to the variable x should be $6x$ (or $3 * 2 * x^1$, or any other equivalent form). For the differentiation of an expression the well-known laws hold for derivative of sum and product. The derivative of each variable other than the variable 'to which' is being differentiated, is zero.

- Write a function

  ```
  diff :: Expr -> Char -> Expr
  ```

  which computes the derived function when differentiating to a given variable. For the differentiation of an expression containing `:^:`, there is no law. In that case we therefore proceed as follows: evaluate the exponent in an empty environment (and just pray it does not contain any variables), and use the law $(f(x)^n)'$ is $n * f(x)^{n-1} * f'(x)$.

  The resulting expression needs not to be simplified.

The type 'Polynomial'

The data structure for polynomials from the reader can be generalized to *represent polynomials with more variables*. A 'polynomial' is still a sequence of 'terms'. A 'term' no more consists of a coefficient and an exponent, but of a coefficient (an integer) and a sequence of 'factors'. Each 'factor' consists of a variable identifier (a character) and an exponent (an integer).

- Define a type `Poly` with which these kind of polynomials can be represented.

Simplifying polynomials

- Write a function which simplifies a polynomial.

Just as in the reader three things must be done:
1. the terms must be sorted, so $x^2 + xy + x^2$ becomes $x^2 + x^2 + xy$;
2. terms with equal factors have to be unified, so $x^2 + 2x^2$ becomes $3x^2$;
3. terms with zero coefficients have to be removed, so $0x^2 + x$ becomes x.
Before all this can be done, we first have to simplify the individual terms. This looks very much like the simplification of complete polynomials:
1. the factors have to be sorted, for instance alphabetic on the occurring variable, so x^2yx becomes x^2xy;
2. factors with equal variables have to be unified, so x^2x becomes x^3;
3. factors with exponent zero have to be removed, so x^0y becomes y.
If you do it properly, you can use the same (higher order) function for both simplifications.

The additive normal form

Almost every mathematical expression can be written as a polynomial in more variables. For example the expression $(x + y)^4$ equals $x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$. If this polynomial is simplified, this is called the *additive normal form* of the expression.

- Write a function

  ```
  anf :: Expr -> Poly
  ```

  which calculates the additive normal form of an expression. In the case of a `:^:`-expression you may, just as with symbolic differentiation, evaluate the exponent first. It is sensible to write auxiliary functions which can multiply factors, terms and polynomials, just as with the functions in the reader.

A.4 Predicate logic

This exercise is about *predicates* from the *predicate logic*. Soon a data structure will be defined which describes a predicate, and the exercise is to write a number of functions on them. For the design of these functions some knowledge of the predicate logic is needed. Hopefully you are already familiar with some basics of logic. If not, you should be.

Let me recall the terminology: like using the operator + is called 'addition', and sing the operator × is called 'multiplication', using logical operators have also names:

expression with $\wedge$	conjunction
expression with $\vee$	disjunction
expression with $\rightarrow$	implication
expression with $\leftrightarrow$	equivalence
expression with $\neg$	negation
expression with $\forall$	universal quantification
expression with $\exists$	existential quantification

These words will be used in the exercise without any further explanation.

The types

If you figure out of what a predicate consists, it shows that there are *relations* between *objects*. In this exercise these relations and objects being used play no rôle; we are only interested in their logical interconnection. Therefor we will denote relations and objects by a name. Common names in predicate logic are a, b, x, y (objects) and P and Q (relations).

In this exercise we (as usual in computer science) permit names to consist of more than one character. For clarity there are two type declarations, so that these can be modified later if desired:

```
type RelName  =  String
type ObjName  =  String
```

In predicate logic relations are for instance written as $C(x, y)$ (interpreted as 'x is a child of y'). So it seems like a relation is a relation name applied to a number of objects. We therefore define the type:

```
type Relation  =  (RelName, [ObjName])
```

So, the Gofer expression which represents $C(x, y)$ equals `("C", ["x","y"])`.

The type that we are going to use to describe predicates is an extension of the type `Prop` in section ??. page ??

```
data Pred  =   Con Bool
           |   Rel Relation
           |   And [Pred]
           |   Or  [Pred]
           |   Imp Pred Pred
           |   Eqv Pred Pred
           |   Not Pred
           |   All ObjName Pred
           |   Exi ObjName Pred
```

The propositional variables in the type `Prop` are replaced by relations. Another difference is that we added constructors to describe quantifications. Furthermore the representation of conjunctions and disjunctions is modified: instead of two parameters we gave these operators a list of parameters. This is done because the operators $\wedge$ and $\vee$ are associative, so we do not have to make artificial discrimination between $a \wedge (b \wedge c)$ and $(a \wedge b) \wedge c$.

In the next examples the predicates will be written as mathematical formulas (with italic characters). In stead of that you should read the representation as a Gofer data structure. For example: $P(x) \wedge \neg Q(x)$ stands for `And [ Rel ("P",["x"]), Neg (Rel ("Q",["x"])) ]`.

The functions

And now... The moment you have been waiting for: a description of the functions which have to be designed. Keep in mind you assign *exactly* the same name and the same type to the function as specified below, since the submitted exercises are tested automatically. It is allowed to write some auxiliary functions, of which you can choose the name freely.

free Object names in a predicate which are not bound by a quantifiers are called *free variables*. The function `free` has to determine which variables are free in a predicate:

```
free  ::  Pred -> [ObjName]
```

closed A predicate without free variables is called *closed*. The function `closed` has to determine if this property is satisfied.

```
closed  ::  Pred -> Bool
```

rename The meaning of a quantification does not depend on the name used in a quantification. For example, $\forall_x\langle P(x)\rangle$ and $\forall y\langle P(y)\rangle$ are completely equal. The function `rename` executes Such a renaming of the bound variables:

```
rename  ::  ObjName -> Pred -> Pred
```

If this function is applied to a quantification the name bound by the quantification will be changed in the given name. Other predicates remain unaltered.

objsubst Sometimes it is necessary to replace objects in a predicate by other objects. The goal is to replace only the *free occurrences*. If for instance in the predicate $P(x) \wedge \forall_x\langle Q(x)\rangle$ the object x is replaced by the object y then the predicate $P(y) \wedge \forall_x\langle Q(x)\rangle$ will be the result. So the x bound by the symbol $\forall$ is not replaced.

The exercise is to write a function

```
objsubst  ::  [(ObjName,ObjName)] -> Pred -> Pred
```

which substitutes all free occurrences of the variables in the left side of the list of pairs in a given predicate (second parameter) for the corresponding right side. If the list contains more than one element, the substitution has to take place at the same time. So if x is to be replaced by y and y by x, the predicate $P(x,y)$ is transformed into $P(y,x)$ and not into $P(x,x)$.

There is a complication which slightly complicates that substitution. For instance, take a look at the predicate $P(x)\wedge\forall_y\langle Q(x,y)\rangle$. If you would replace the object x by y the predicate $P(y)\wedge\forall_y\langle Q(y,y)\rangle$ would be the result. The first parameter of Q, which was free, is suddenly bound by the $\forall$. That was not ment to happen. The desired result is $P(y) \wedge \forall_z\langle Q(y,z)\rangle$. Because the replace-object (y) was bound to be bound we renamed the variable of the $\forall$ before the replacement. (The function `rename` is very useful here!)

Which name you choose (z, for instance) does not matter. A condition is though, the name should not yet occur free in the predicate (x would have been a bad choice) and not equal to the replace-objects. IT might be practical to design a function which comes up with an arbitrary name which is not in a given list of exceptions.

relsubst Next to substitution of objects by objects there exists also substitution of predicates for relations. First, take a look at an example. In the predicate $P(a,b) \wedge Q(c)$ the relation $P(x,y)$ can be replaced by $R(x) \wedge S(y)$. The result is the predicate $R(a) \wedge S(b) \wedge Q(c)$.

What happens in this example is that each relation p in the original predicate will be replaced by the predicate with the R and the S, where x and y are being replaced by the originally encountered a and b.

The function to be written is

```
relsubst  ::  Relation -> Pred -> Pred -> Pred
```

The first parameter is the relation which must be replaced (in the example $P(x,y)$). The second parameter is the predicate which it is replaced by (in the example $R(x) \wedge S(y)$). The third parameter is the predicate where the replacement should take place (in the example $P(a,b) \wedge Q(c)$). Take a close look at the way the variables x and y in this example (which

were summarized in the second component of the `Relation` parameter) are being replaced by a and b (which were behind p in the `Pred`-parameter).

Also with this substitution we again have to pay attention to undesired bindings. An example which demonstrates this is the call

`relsubst` $(Q(x))\ \ (R(x) \wedge S(y))\ \ (\forall_y \langle Q(y) \wedge T(y)\rangle)$

If you would execute this substitution without any hesitation then the predicate $\forall_y \langle R(y) \wedge S(y) \wedge T(y)\rangle$ would be the result. Here the y in $S(y)$ is bound erroneously. The desired result is $\forall_z \langle R(z) \wedge S(y) \wedge T(z)\rangle$. (Study this example closely; it is designed in such a way that all problems are demonstrated in it).

prenex In a course predicate logic it is oftenly desired to write predicates in *prenex* form, that is to say all quantifiers together in front. We are going to automate this process too. The type of the function is

```
prenex  ::  Pred -> Pred
```

For an exact specification you can use the material of a predicate logic course. (Hint: you might need the function `rename`.)

The operator $\leftrightarrow$ can be 'removed' by using the equality between $P \leftrightarrow Q$ and $P \rightarrow Q \wedge Q \rightarrow P$.

dnv Each predicate without quantifiers has a *disjunctive normal form*, short *dnv*. The dnv is a formula consisting of a huge disjunction, of which the elements are conjunctions, of which the elements are relations or negations of relations.

The type of the function is

```
dnv  ::  Pred -> Pred
```

This function is not needed to work on predicates which are quantified

You can find the dnv by using truth tables, but it is much easier to find the dnv by formula manipulation. You will need a case distinction:

- Is this formula a disjunction? Then write all sub-formulas (recursively) in dnv, and unify all these disjunctions into one big disjunction.
- Is the formula a conjunction, for instance $P \wedge Q$? Then recursively find all the dnvs of the sub-formulas. This for instance results in $(P_1 \vee P_2) \wedge (Q_1 \wedge Q_2)$. You can make this into *one* disjunction by removing the parenthesis.
- Is the formula an implication or equivalence? Then first write it as a conjunction or disjunction.
- Is the formula a negation? Then move it 'inward'.

prenexdnv This function converts a predicate to its prenex form, and converts the inner part to its dnv:

```
prenexdnv  ::  Pred -> Pred
```

simple By all that manipulating some very complicated formulas may be created. The function `simple` has to be able to simplify with the following rules:

- A conjunction containing both $P(x)$ as $\neg P(x)$ is *False*;
- A disjunction containing both $P(x)$ as $\neg P(x)$ is *True*;
- A conjunction containing *False* is *False*;
- A disjunction containing *True* is *True*;
- The constant *True* can be removed from conjunctions;
- The constant *False* can be removed from disjunctions;
- An empty conjunction is *True*;
- An empty disjunction is *False*;
- A double negation dissolves;
- Quantifications over variables which are not used disappear;
- A conjunction containing a conjunction can be written as one huge conjunction;
- Dito for a disjunction.

Easier testing

To simplify the testing of your program there are two functions available:

```
showPred  ::  Pred -> String
parsePred ::  String -> Pred
```

with which a predicate can be converted in a more readable representation, and vice versa. These functions are in a file of which the location will be supplied later. You can add this file to your own function definitions by the Gofer command `:a`. Read the comments in this file for a description of how a predicate parsed by `parsePred` should be written.

Guidelines for submission

Write a Gofer source defining the specified types (`RelName`, `ObjName`, `Relation` en `Pred`) and all desired functions: `free`, `closed`, `rename`, `objsubst`, `relsubst`, `prenex`, `dnv`, `prenexdnv`, en `simple`. For every function also specify the type. The functions have to be named identically, because the programs are tested automatically Do *not* put the functions `showPred` and `parsePred` in your program.

A.5 The class 'Set'

Problem

In this exercise a class is defined where operations on *sets* are described. Next two implementations of set operations will be modeled as an instance of this class.

All parts of the exercise are tagged by a •. Name all the desired functions, types etc. exactly as they are described in this exercise (including capitals). Of course you are allowed to make up your own names for parameters and auxiliary functions.

Sets

A type is a *set*-type if the functions `intersection`, `union` and `complement` can be defined on it, and the constants `empty` and `universe` are available.

- Define a class `Set`, containing the mentioned functions. Put a default definition for `universe` in the definition.

First implementation

Lists seem to be a useful implementation of sets. However, the problem is that the complement function cannot be defined. What is the complement of an empty set? For sets of natural numbers you still could pick the list `[0..]`, but how do you define this for sets of arbitrary type...

We therefore choose another approach. For the to be represented set either *all* elements of the set are summarized or all the elements *not* in the list are summarized. These two cases are distinguished by applying the constructor function `Only`, respectively `Except` on the list.

- Give a data declaration for `ListSet`, with which sets of an arbitrary type can be stored.
- Make the defined type an instance of `Set`. Do not forget to supply a condition for the object type of the set.
- Make the type an instance of `Eq`. Keep in mind that the number of times an elements occurs in the set does not matter, in contrast to lists.
- Make the type an instance of `Ord`. The rôle of the $\leq$ operator is played by the *subset* relation.

Second implementation

Another possible implementation of sets are functions with `Bool` result. An element is member of a given set, if the function representing the set results `True` if it is applied to that element.

- Give a `type` declaration for `FunSet`, with which sets can be represented.
- Make `FunSet` an instance of `List`.
- Describe in your own words (as a comment with the program) why `FunSet` cannot be made an instance of `Ord`.
- Also describe in the comments why the second implementation also has advantages to the first. Show what kind of sets cannot be represented as `ListSet` but can be represented as `FunSet`.

More functions on sets

By using the functions declared in the class `Set`, it is possible to write *overloaded* functions operating on sets, regardless their implementation.

- Write a function `difference` which determines the difference between two sets. Also supply the type of this function.
- Write a function `symmDifference` which computes the symmetrical difference of two sets.

Appendix B

ISO/ASCII table

<table>
<tr><th></th><th>0*16+...</th><th>1*16+...</th><th>2*16+...</th><th>3*16+...</th><th>4*16+...</th><th>5*16+...</th><th>6*16+...</th><th>7*16+...</th></tr>
<tr><td>...+0</td><td>0 NUL</td><td>16 DLE</td><td>32 SP</td><td>48 0</td><td>64 @</td><td>80 P</td><td>96 ‘</td><td>112 p</td></tr>
<tr><td>...+1</td><td>1 SOH</td><td>17 DC1</td><td>33 !</td><td>49 1</td><td>65 A</td><td>81 Q</td><td>97 a</td><td>113 q</td></tr>
<tr><td>...+2</td><td>2 STX</td><td>18 DC2</td><td>34 "</td><td>50 2</td><td>66 B</td><td>82 R</td><td>98 b</td><td>114 r</td></tr>
<tr><td>...+3</td><td>3 ETX</td><td>19 DC3</td><td>35 #</td><td>51 3</td><td>67 C</td><td>83 S</td><td>99 c</td><td>115 s</td></tr>
<tr><td>...+4</td><td>4 EOT</td><td>20 DC4</td><td>36 $</td><td>52 4</td><td>68 D</td><td>84 T</td><td>100 d</td><td>116 t</td></tr>
<tr><td>...+5</td><td>5 ENQ</td><td>21 NAK</td><td>37 %</td><td>53 5</td><td>69 E</td><td>85 U</td><td>101 e</td><td>117 u</td></tr>
<tr><td>...+6</td><td>6 ACK</td><td>22 SYN</td><td>38 &</td><td>54 6</td><td>70 F</td><td>86 V</td><td>102 f</td><td>118 v</td></tr>
<tr><td>...+7</td><td>7 BEL</td><td>23 ETB</td><td>39 ’</td><td>55 7</td><td>71 G</td><td>87 W</td><td>103 g</td><td>119 w</td></tr>
<tr><td>...+8</td><td>8 BS</td><td>24 CAN</td><td>40 (</td><td>56 8</td><td>72 H</td><td>88 X</td><td>104 h</td><td>120 x</td></tr>
<tr><td>...+9</td><td>9 HT</td><td>25 EM</td><td>41)</td><td>57 9</td><td>73 I</td><td>89 Y</td><td>105 i</td><td>121 y</td></tr>
<tr><td>...+10</td><td>10 LF</td><td>26 SUB</td><td>42 *</td><td>58 :</td><td>74 J</td><td>90 Z</td><td>106 j</td><td>122 z</td></tr>
<tr><td>...+11</td><td>11 VT</td><td>27 ESC</td><td>43 +</td><td>59 ;</td><td>75 K</td><td>91 [</td><td>107 k</td><td>123 {</td></tr>
<tr><td>...+12</td><td>12 FF</td><td>28 FS</td><td>44 ,</td><td>60 <</td><td>76 L</td><td>92 \</td><td>108 l</td><td>124 |</td></tr>
<tr><td>...+13</td><td>13 CR</td><td>29 GS</td><td>45 -</td><td>61 =</td><td>77 M</td><td>93]</td><td>109 m</td><td>125 }</td></tr>
<tr><td>...+14</td><td>14 SO</td><td>30 RS</td><td>46 .</td><td>62 ></td><td>78 N</td><td>94 ^</td><td>110 n</td><td>126 ~</td></tr>
<tr><td>...+15</td><td>15 SI</td><td>31 US</td><td>47 /</td><td>63 ?</td><td>79 O</td><td>95 _</td><td>111 o</td><td>127 DEL</td></tr>
</table>

Gofer notation for special tokens in a string is done by typing a special code after a backslash:

- the *name* of the special token, for instance `"\ESC"` for the escape token;
- the *number* of the special token, for instance `"\27"` for the escape token;
- the octal number, e.g. `"\o33"` for the escape token;
- the hexadecimal number, e.g. `"\x1B"` for the escape token;
- by the similar letter four columns further to the right, e.g. `"\^["` for the escape sign;
- one of the following codes: `"\n"` (newline), `"\b"` (backspace), `"\t"` (tab), `"\a"` (alarm), `"\f"` (formfeed), `"\""` ("-symbool), `"\'"` ('-symbool), en `"\\"` (\-symbool)

Appendix C

Gofer manual page

Synopsis

`gofer` Functional programming language
`gofc` Gofer-to-C compiler
`gofcc` Compile C program generated by `gofc`

`gofer` [*options*] [*files*]
`gofc` [*options*] [*files*]
`gofcc` *file*`.c`

File parameters

File parameters can be of three types:

Gofer scripts File containing definitions of functions, operators, types, etc. Comments are enclosed by `{-` and `-}`, or by `--` and end-of-line. Files ending in `.hs`, `.has`, `.gs` , `.gof` and `.prelude` are always treated as Gofer scripts.

Literate scripts File in which everything is comment except lines that start with a `>`-symbol. Files ending in `.lhs`, `.lgs`, `.verb` and `.lit` are always treated as literate scripts.

Project files File containing names of files and options that are used as parameter to Gofer. Project filenames should be preceded by a `+`-symbol and a space.

Filenames that cannot be classified according to their suffix are treated as Gofer script, or as literate script if the `+l` option is used (see below).

Options

Below is a list of toggles that can be switched on by `+`*letter* or off by `-`*letter*. In the list the default settings are shown.

`+s` Print number of reductions/cells after evaluation
`-t` Print type after evaluation
`-d` Show dictionary values in output expressions
`+f` Terminate evaluation on first error
`-g` Print number of recovered cells after garbage collection
`+c` Test conformality for pattern bindings
`-l` Literate scripts as default
`+e` Warn about errors in literate scripts
`-i` Apply `fromInteger` to integer literals
`+o` Optimise use of `(&&)` and `(||)`
`-u` Catch ambiguously typed top-level variables
`-.` Print dots during file analysis
`+w` Always show which files are loaded
`+1` Overload singleton list notation
`-k` Show ‘kind’-errors in full

Other options are:

`-h`*number*	Set heap size in cells (default 100000)
`-p`*string*	Set prompt string (default `?`)
`-r`*string*	Set 'repeat last expression' command (default `$$`)

Commands

Commands that can be typed to the interpreter are:

expression	Evaluate expression
`:quit`	Quit interpreter
`:?`	Display this list of commands
`:load` *files*	Load scrips from specified files
`:also` *files*	Read additional files
`:project` *file*	Use project file
`:edit` *file*	Edit file and reload if necessary
`:type` *expression*	Print type of expression
`:set` *options*	Set options (as in commandline)
`:info` *names*	Describe named functions, types etc.
`:find` *name*	Edit file containing definition of *name*
`:names` *pattern*	List names currently in scope
`!`*command*	Shell escape
`:cd` *directory*	Change working directory

Compiler

Gofer programs can be compiled using `gofc`. One of the scripts should contain a definition for a function `main Dialogue`. If you don't know what a 'Dialogue' is, you can use the definition

```
main = interact f
```

where `f` is a function `f::String->String`. The program `gofc` generates a C program, using the last filename with `.c` appended. The generated C program can be compiled and linked with `gofcc`, which generates `a.out`.

Environment variables

GOFER Name of standard prelude (script that is always loaded before all user files)

EDITOR Name of the editor to be used by the `:edit` command

EDITLINE Description how the editor can be called with specified linenumber and filename, e.g. `"vi +%d %s"`.

Files

In this list, '. . . ' is Gofer directory which is currently `/packages/gofer-2.28`.

`.../lib/standard.prelude`	Standard prelude
`.../lib/*.prelude`	Alternative preludes
`.../lib/demos/*`	Example programs
`.../lib/runtime.o`	Runtime system linked by `gofcc`
`.../include/*.h`	Files included by `gofcc`
`.../doc/*`	User manuals (LaTeX-source)

Appendix D

Gofer standard functions

Operator priorities

```
infixl  9  !!
infixr  9  .
infixr  8  ^
infixl  7  *
infix   7  /, ‘div‘, ‘quot‘, ‘rem‘, ‘mod‘
infixl  6  +, -
infix   5  \\
infixr  5  ++, :
infix   4  ==, /=, <, <=, >=, >
infix   4  ‘elem‘, ‘notElem‘
infixr  3  &&
infixr  2  ||
infixr  0  $
```

Functions on Booleans and characters

```
otherwise   ::  Bool
not         ::  Bool   -> Bool
(&&), (||)  ::  Bool   -> Bool -> Bool
and, or     ::  [Bool] -> Bool
any, all    ::  (a->Bool) -> [a] -> Bool

isAscii, isControl, isPrint, isSpace           ::  Char -> Bool
isUpper, isLower,   isAlpha, isDigit, isAlnum  ::  Char -> Bool

minChar, maxChar   ::  Char
toUpper, toLower   ::  Char -> Char
ord                ::  Char -> Int
chr                ::  Int  -> Char
```

Numerical Functions

```
even, odd  ::  Int -> Bool
gcd, lcm   ::  Int -> Int -> Int
div, quot  ::  Int -> Int -> Int
rem, mod   ::  Int -> Int -> Int

pi  ::  Float
sin, cos, tan     ::  Float -> Float
asin, acos, atan  ::  Float -> Float
log, log10        ::  Float -> Float
exp, sqrt         ::  Float -> Float
atan2             ::  Float -> Float -> Float
```

```
truncate          ::  Float -> Int

subtract       ::  Num a          =>  a -> a -> a
(^)            ::  Num a          =>  a -> Int -> a
abs            ::  (Num a, Ord a) =>  a -> a
signum         ::  (Num a, Ord a) =>  a -> Int
sum, product   ::  Num a          =>  [a] -> a
sums, products ::  Num a          =>  [a] -> [a]
```

Polymorphic functions

```
undefined ::  a
id        ::  a -> a
const     ::  a -> b -> a
asTypeOf  ::  a -> a -> a
($)       ::  (a->b) -> (a->b)
strict    ::  (a->b) -> (a->b)

(.)       ::  (b->c)     -> (a->b) -> (a->c)
uncurry   ::  (a->b->c)  -> ((a,b)->c)
curry     ::  ((a,b)->c) -> (a->b->c)
flip      ::  (a->b->c)  -> (b->a->c)
until     ::  (a->Bool)  -> (a->a) -> a -> a
until'    ::  (a->Bool)  -> (a->a) -> a -> [a]
```

Functions on tuples

```
fst   ::  (a,b)   -> a
snd   ::  (a,b)   -> b
fst3  ::  (a,b,c) -> a
snd3  ::  (a,b,c) -> b
thd3  ::  (a,b,c) -> c
```

Functions on lists

```
head   ::  [a] -> a
last   ::  [a] -> a
tail   ::  [a] -> [a]
init   ::  [a] -> [a]
(!!)   ::  [a] -> Int -> a

take, drop           ::  Int       -> [a] -> [a]
splitAt              ::  Int       -> [a] -> ([a],[a])
takeWhile, dropWhile ::  (a->Bool) -> [a] -> [a]
takeUntil            ::  (a->Bool) -> [a] -> [a]
span, break          ::  (a->Bool) -> [a] -> ([a],[a])

length           ::            [a] -> Int
null             ::            [a] -> Bool
elem, notElem    ::  Eq a  =>  a   -> [a] -> Bool
maximum, minimum ::  Ord a =>  [a] -> a
genericLength    ::  Num a =>  [b] -> a

(++)      ::  [a] -> [a]  -> [a]
iterate   ::  (a->a) -> a -> [a]
repeat    ::  a           -> [a]
cycle     ::  [a]         -> [a]
```

```
copy      ::  Int -> a    -> [a]
reverse   ::  [a]         -> [a]
nub       ::  Eq a => [a] -> [a]
(\\)      ::  Eq a => [a] -> [a] -> [a]

concat    ::  [[a]] -> [a]
transpose ::  [[a]] -> [[a]]

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

foldl     ::  (a->b->a) -> a -> [b] -> a
foldl'    ::  (a->b->a) -> a -> [b] -> a
foldr     ::  (a->b->b) -> b -> [a] -> b
foldl1    ::  (a->a->a) ->      [a] -> a
foldr1    ::  (a->a->a) ->      [a] -> a
scanl     ::  (a->b->a) -> a -> [b] -> [a]
scanl'    ::  (a->b->a) -> a -> [b] -> [a]
scanr     ::  (a->b->b) -> b -> [a] -> [b]
scanl1    ::  (a->a->a) ->      [a] -> [a]
scanr1    ::  (a->a->a) ->      [a] -> [a]

insert    ::  Ord a =>  a -> [a] -> [a]
sort      ::  Ord a =>       [a] -> [a]
qsort     ::  Ord a =>       [a] -> [a]
merge     ::  Ord a =>       [a] -> [a] -> [a]

zip   ::  [a] -> [b]                                    -> [(a,b)]
zip3  ::  [a] -> [b] -> [c]                             -> [(a,b,c)]
zip4  ::  [a] -> [b] -> [c] -> [d]                      -> [(a,b,c,d)]
zip5  ::  [a] -> [b] -> [c] -> [d] -> [e]               -> [(a,b,c,d,e)]
zip6  ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f]        -> [(a,b,c,d,e,f)]
zip7  ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a,b,c,d,e,f,g)]

zipWith   ::  (a->b->c)                -> [a]->[b]->[c]
zipWith3  ::  (a->b->c->d)             -> [a]->[b]->[c]->[d]
zipWith4  ::  (a->b->c->d->e)          -> [a]->[b]->[c]->[d]->[e]
zipWith5  ::  (a->b->c->d->e->f)       -> [a]->[b]->[c]->[d]->[e]->[f]
zipWith6  ::  (a->b->c->d->e->f->g)    -> [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7  ::  (a->b->c->d->e->f->g->h) -> [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip     ::  [(a,b)] -> ([a],[b])
```

Functions on strings

```
ljustify    ::  Int -> String -> String
cjustify    ::  Int -> String -> String
rjustify    ::  Int -> String -> String
space       ::  Int -> String

words     ::  String   -> [String]
lines     ::  String   -> [String]
unwords   ::  [String] -> String
unlines   ::  [String] -> String
layn      ::  [String] -> String

type ShowS  = String -> String

showChar   ::            Char   -> ShowS
showString ::            String -> ShowS
```

```
shows      ::  Text a => a      -> ShowS
show       ::  Text a => a      -> String
show'      ::             a     -> String

openfile   ::  String -> String
```

Type classes

```
class Eq a where
    (==), (/=) ::  a -> a -> Bool

class Eq a => Ord a where
    (<), (<=), (>), (>=) ::  a -> a -> Bool
    max, min             ::  a -> a -> a

class Ord a => Ix a where
    range   ::  (a,a) -> [a]
    index   ::  (a,a) -> a -> Int
    inRange ::  (a,a) -> a -> Bool

class Ord a => Enum a where
    enumFrom       ::  a -> [a]                -- [n..]
    enumFromThen   ::  a -> a -> [a]           -- [n,m..]
    enumFromTo     ::  a -> a -> [a]           -- [n..m]
    enumFromThenTo ::  a -> a -> a -> [a]      -- [n,n'..m]

class (Eq a, Text a) => Num a where
    (+), (-), (*), (/) ::  a -> a -> a
    negate             ::  a -> a
    fromInteger        ::  Int -> a

class Text a where
    showsPrec      ::  Int -> a   -> ShowS
    showList       ::         [a] -> ShowS
```

Instances of type classes

```
instance  Eq   ()
instance  Ord  ()
instance  Text ()
instance  Eq   Bool
instance  Ord  Bool
instance  Text Bool
instance  Eq   Char
instance  Ord  Char
instance  Ix   Char
instance  Enum Char
instance  Text Char
instance  Eq   Int
instance  Ord  Int
instance  Ix   Int
instance  Enum Int
instance  Num  Int
instance  Text Int
instance  Eq   Float
instance  Ord  Float
instance  Enum Float
instance  Num  Float
```

```
instance  Text Float
instance  Eq a            => Eq   [a]
instance  Ord a           => Ord  [a]
instance  Text a          => Text [a]
instance  (Eq a, Eq b)    => Eq   (a,b)
instance  (Ord a, Ord b)  => Ord  (a,b)
instance  (Text a, Text b) => Text (a,b)
```

Input/output functions

```
type Dialogue    =  [Response]  -> [Request]
type SuccCont    =                 Dialogue
type StrCont     =  String      -> Dialogue
type StrListCont =  [String]    -> Dialogue
type FailCont    =  IOError     -> Dialogue

stdin, stdout, stderr, stdecho ::  String

done            ::                                               Dialogue
readFile        ::  String ->           FailCont -> StrCont     -> Dialogue
writeFile       ::  String -> String -> FailCont -> SuccCont    -> Dialogue
appendFile      ::  String -> String -> FailCont -> SuccCont    -> Dialogue
readChan        ::  String ->           FailCont -> StrCont     -> Dialogue
appendChan      ::  String -> String -> FailCont -> SuccCont    -> Dialogue
echo            ::  Bool   ->           FailCont -> SuccCont    -> Dialogue
getArgs         ::                      FailCont -> StrListCont -> Dialogue
getProgName     ::                      FailCont -> StrCont     -> Dialogue
getEnv          ::  String ->           FailCont -> StrCont     -> Dialogue

abort     ::            FailCont
exit      ::            FailCont
interact  ::            (String->String) -> Dialogue
run       ::            (String->String) -> Dialogue
print     ::  Text a => a                -> Dialogue
prints    ::  Text a => a -> String      -> Dialogue
```

Appendix E

Literature

Text books with comparable material

- Richard Bird en Philip Wadler: *Introduction to functional programming.* Prentice-Hall, 1988.
- Richard Bird en Philip Wadler: *Functioneel programmeren, een inleiding.* Academic service, 1991.
- A.J.T. Davie: *An introduction to functional programming systems using Haskell.* Cambridge university press, 1992.
- Ian Hoyler: *Functional programming with Miranda.* Pitman, 1991.
- Hudak en Fasel: 'a gentle introduction to Haskell'. *ACM sigplan notices* **27**, 5 (may 1992) pp.T1–T53.

Text books in which another programming language is used

- Rinus Plasmeijer en Marko van Eekelen: *Functional programming and parallel graph rewriting.* Addison-Wesley, 1993.
- Åke Wikström: *Functional programming using standard ML.* Prentice-Hall, 1987.
- Chris Reade: *Elements of functional programming.* Addison-Wesley, 1989.
- Roger Bailey: *Functional programming with Hope.* Ellis-Horwood, 1990.
- Abelson en Sussman: *Structure and interpretation of computer programs.* McGraw-Hill, 1985.

Language descriptions

- Hudak, Peyton-Jones, Wadler *et.al.*: 'Report on the programming language Haskell: a non-strict purely functional language, version 1.2.' *ACM sigplan notices* **27**, 5 (may 1992) pp.R1–R164.
- Mark P. Jones: *Introduction to Gofer 2.20.* Oxford Programming Research Group, 1992.

Implementation of functional programming languages

- Mark P. Jones: *The implementation of the Gofer functional programming system.* Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.
- Field en Harrison: *Functional programming.* Addison-Wesley, 1989.
- Simon Peyton-Jones: *The implementation of functional programming languages.* Prentice-Hall, 1987.
- Simon Peyton-Jones en David Lester: *Implementing functional languages: a tutorial.* Prentice-Hall, 1992.

Advanced material

- Johan Jeuring en Erik Meijer (eds): *Advanced functional programming.* Springer, 1985 (LNCS 925).
- Colin Runciman: *Applications of functional programming.* Caombridge university press, 1995.
- *Proceedings of the ...th conference on Functional Programming and Computer Architecture (FPCA).* Springer, 1992–1995.

Background, underlying theory

- Henk Barendregt: *The lambda-calculus: its syntax and semantics.* North-Holland, 1984.
- Richard Bird: *An introduction to the theory of lists.* Oxford Programming Research Group report PRG–56, 1986.
- Luca Cardelli en P.Wegner: 'On understanding types, data abstraction and polymorphism.' *Computing surveys* **17**, 4 (1986).
- R. Milner: 'A theory of type polymorphism in programming.' *J.computer and system sciences* **17**, 3 (1978).

Magazines

- *Journal of functional programming* (1991–). Cambridge University Press.
- `comp.lang.functional`. Usenet newsgroup.

Historical material

- Alonzo Church: 'The calculi of lambda-conversion'. *Annals of mathematical studies* **6**. Princeton university, 1941; Kraus reprint 1971.
- Haskell Curry *et al.*: *Combinatory logic, vol. I.* North-Holland, 1958.
- A. Fraenkel: *Abstract set theory.* North-Holland, 1953.
- M. Schönfinkel: 'Über die Bausteine der mathematischen Logik.' *Mathematischen Annalen* **92** (1924).

Index

++, 39
., 27
&&, 11, 49

abcFormula, 10, 13, 17
    definitie, 9, 10
abcFormula', 10, 32
abs, 7, 10, 55, 86
    definitie, 10
AChar, 63
actual parameters, 11
addition
    fractions, 55
    numbers, 7
    polynomial, 82
    vectors, 74
after, 3, 27, 34
    definitie, 27
altsum, 6, 78
    definitie, 78
and, 9, 26, 33, 40, 42, 92
    definitie, 26
angle of vectors, 73
AnInt, 63
apostrophe, 21
arccos, 74
    definitie, 33
arcsin
    definitie, 33
'as'-pattern, 71
association
    of :, 38

back quote, 21
between, 70
    definitie, 69
bins, 6
Bool (type), 14–17, 45, 62
box, 4, 64

Char (type), 15, 45, 46, 50
choose, 4, 5, 17, 104, 105
    definitie, 4, 9, 104
chr, 46, 47, 63
Church, Alonzo, 1
class
    Eq, 17, 56, 59
    Num, 17
    Ord, 17, 44, 46, 59
Clean, 1
combination, 67, 70
combinatorial function, 67–71
combs, 70, 103–105
    definitie, 70
combs n, 67
comments, 13
comparing
    lists, 39
Complex (type), 54
complex numbers, 64
complexity, 86
comprehension, 51
concat, 40, 52, 63, 64, 69, 94, 106, 107, 111
    definitie, 40
concatenation, 39
constant, 5
constructor functions, 57
control structure, 26
copy, 49
    definitie, 49
cos, 3, 16, 33
cp, 8, 83
cpWith, 82
    definitie, 82
cross product, 82
crossprod, 8, 83
cubicRoot, 3, 34, 35
curry, 56, 64
    definitie, 64
Curry, Haskell, 1

data definition, 57
datatype, 57
day, 29, 30
    definitie, 29, 30
daynumber, 29, 30
    definitie, 30
dayofweek, 41
degree, 79
deleteTree, 61
    definitie, 60
denominators, 29, 30
    definitie, 29
depth, 65
Det, 83
det, 6, 78, 79
    definitie, 78
determinant, 77
diff, 31, 33, 35, 37
    definitie, 31
digitChar, 47

definitie, 47
digitValue, 4, 47
definitie, 47
distance, 54
div, 22
divisible, 29, 30, 51, 55
definitie, 29
divisors, 48, 55
drop, 40, 41, 43, 45, 53, 88
definitie, 41
dropWhile, 43, 64, 71
definitie, 43, 71

e
definitie, 9
eager evaluation, 48
elem, 41, 42, 59, 64
definitie, 41, 42
elem'
definitie, 59
elemBoom
definitie, 59
elemTree, 59, 64, 87
empty list, 37
enumFrom, 49
enumFromTo, 38
definitie, 38
Eq (class), 17, 56, 59
eq, 39
equality
on lists, 39
evaluation
lazy, 85
even, 8, 9, 12, 16, 19, 27, 88
definitie, 12
exp, 7, 9, 18, 33

fac, 4, 6, 9, 23, 87, 104, 105
definitie, 4, 6, 9, 12
fib, 89, 100, 101
definitie, 89, 100
filter, 25, 29, 38, 42, 43, 50–52, 55, 64, 70, 81
definitie, 25, 42
flexDiff, 31
definitie, 31
Float (type), 15, 17, 28, 30, 31, 33, 45, 53, 54, 80
floatString, 83
fold, 5, 91
foldl, 4, 9, 26, 43, 91, 92, 94, 98, 111
definitie, 43
foldl', 6, 9, 92, 111
definitie, 92
foldr, 4, 5, 8, 9, 26, 33, 38, 41–44, 63–65, 75, 91, 92, 94, 98–100, 111
definitie, 26, 42
foldTree, 5, 65
for (keyword), 26

formal parameters, 11
Fraction (type), 63
fractionString, 55
from, 48, 49
definitie, 48
fromInteger, 7
fst
definitie, 53
function
as operator, 21
combinatorial, 67–71
complexity, 86
on lists, 38
polymorphic, 16
functions
polymorphic, 16

gaps, 6, 7, 78, 79, 83
garbage collector, 90
gcd, 7, 55
definitie, 55
ge, 39
Gofer, 2
goodEnough, 32
greatest common divisor, 55
group, 5, 65
gt, 39
guards, 10

Haskell, 2
head, 12, 13, 16, 38, 40, 51, 86, 87
definitie, 11, 40
higher order function, 26

id, 17
definitie, 16
identity matrix, 73
improve, 32
inductive definition, 12
infix (keyword), 23
infixl (keyword), 23
inFrontOf, 81
init, 40
definitie, 40
initial segment, 67, 68
inits, 67, 68, 83, 103, 105, 106
definitie, 68
inner product, 74
insert, 44, 45, 60, 64, 87, 88
definitie, 44
insertTree, 60
definitie, 60
Int (type), 5, 14–17, 23, 28, 45, 50, 80
integrate, 35
interpreter, 2
interval-notation, 38
IntOrChar (type), 62
intString, 50, 55, 65, 83
definitie, 50
inverse, 33–35, 37, 74

definitie, 33
`is...`
definitie, 46
`isort`, 45, 60, 64, 81, 86–88
definitie, 44
`isSpace`, 47
`isZero`
definitie, 25
`iszero`
definitie, 10
`iterate`, 49–51, 64, 65, 76
definitie, 49

`join`, 61, 62

keyword
`for`, 26
`infix`, 23
`infixl`, 23
`type`, 54
`where`, 1, 10, 11, 13, 14, 27, 28, 31, 89
`while`, 26

`labels`, 60, 65, 87, 89, 90
definitie, 60, 89
`labelsBefore`, 89
`largest`, 61, 62
`last`, 40, 55, 86, 87
definitie, 40
law, 93
`layn`, 47
lazy evaluation, 48, 85
`Leaf`, 57, 58, 62
`leap`, 30
definitie, 30
`len`, 101–103
`length`, 3, 5, 9, 13, 14, 16, 38, 41, 45, 49, 57, 58, 64, 70, 86, 87, 101
definitie, 13, 41
linear mapping, 72
`lines`, 47
Lisp, 1
list, 37–52
comparing, 39
concatenating, 39
empty, 37
equality, 39
interval notation, 38
singleton, 37
summary, 37, 38
type, 37
list-comprehension, 51
`listToTree`, 60, 65
definitie, 60
`ln`
definitie, 33
local definitions, 10
`log`, 7

`map`, 5, 9, 17, 23–26, 38, 41, 42, 47, 48, 51, 52, 56, 59, 64, 65, 69, 70, 73–75, 77, 94, 96, 97, 101, 103, 106, 107
definitie, 25, 42
`mapp`, 74
definitie, 74
mapping, 72
composition, 72
identity, 73
`mapTree`, 5, 65
`Mat`, 74, 75, 78, 79
`matApply`, 75, 76
definitie, 76
`matId`
definitie, 76
`matIdent`, 76
definitie, 76
`matInv`, 79, 83
`matPlus`, 74
definitie, 74
`matProd`, 72, 77
definitie, 77
matrix, 71–79
applying to a vector, 75
determinant, 77
identity, 73
inverse, 79
multiplication, 77
transposing, 75
`matScale`, 74
definitie, 74
`matTransp`, 72, 74
definitie, 74
MacCarthy, John, 1
`merge`, 44, 45, 64, 88
definitie, 44
Miranda, 1
ML, 1
`mod`, 22
`months`, 30
definitie, 30
`msort`, 5, 64, 87, 88
`multiple`, 51
multiplication
fractions, 55
matrices, 72
numbers, 7
polynomial, 82

`na`, 3
`Nat` (type), 107
`ne`, 39
`negative`
definitie, 10
Neuman, John von, 1
`Node`, 5, 57, 58
`not`, 8, 9, 27, 42
`notElem`, 41
definitie, 41, 42

null, 5, 9
Num (class), 17
numberSol, 18

odd, 27
  definitie, 9
O-notation, 86
op, 5
operator, 5, 21
  ++, 39
  ., 27
  &&, 11, 49
  as function, 21
or, 41, 42, 59
  definitie, 42
Ord (class), 17, 44, 46, 59
ord, 46, 47
order, 86

pAdd, 8
pair, 52–53
pEq, 80
perms, 67, 69, 70
  definitie, 70
permutation, 67, 69–70
pEval, 82
pi, 18
  definitie, 5, 9
plus, 3, 23, 24
pMul, 8
Point (type), 54
Poly (type), 79, 80
polymorphic function, 16
polymorphic functions, 16
polymorphic type, 16, 37
polymorphism, 16
polynomial, 79–83
  addition, 82
  degree, 79, 82
  multiplication, 82
  simplification, 80
positive
  definitie, 10
predefined, 6
prime, 29, 30, 48, 50
  definitie, 29
primenums
  definitie, 51
primes, 29
  definitie, 29
primitive functions, 6
primPlusInt, 6
  definitie, 6
priority, 21–22
product, 4, 6, 9, 26, 33
  definitie, 26, 92
pSimple, 81, 82

qAdd, 63
qDiv, 63
qEq, 64
qMul, 63
qsort, 5
qSub, 63

Rat, 63
Rational numbers, 54
recursive definition, 12
rem, 22, 28–30, 55
repeat, 49, 64, 75, 76
  definitie, 49
reserved keywords, 6
Reverse, 98
reverse, 3, 41, 50, 98, 99
  definitie, 100
root, 33, 35
  definitie, 32, 33
round, 7, 63

Scheme, 1
Schönfinkel, M., 1
se, 39
search, 4, 56, 59, 64
searchTree, 4
segment, 67, 68
segs, 6, 67, 68, 83, 103, 104, 110
  definitie, 68
show, 83
showTree, 65
signum, 7, 10, 55
  definitie, 10
simplification
  fraction, 55
  polynomial, 80
simplify, 55
  definitie, 55
sin, 3, 7, 16, 24, 31, 33
singleton, 8
singleton list, 37
size, 87
  definitie, 58
snd
  definitie, 53
sort, 3, 44, 46
  definitie, 60
sortBy, 81
  definitie, 81
sorting
  by an ordering, 81
sortTerms, 81
splitAt, 53
  definitie, 53
sqrt, 3, 7–11, 18, 19, 30, 31, 53, 73, 101
sqrt3, 3
square, 9, 18, 24, 33, 51, 73, 88
  definitie, 9, 25
st, 39
strict, 92
  definitie, 92

String (type), 115
string, 45
stringInt, 64
subs, 6, 67, 69, 70, 83, 87, 89, 111
  definitie, 69
subsequence, 67, 69
Succ, 107
successor, 23
  definitie, 23, 25
sum, 3, 6, 9, 15, 16, 26, 30, 33, 38, 49, 73, 74, 78, 83, 86, 87, 102, 103
  definitie, 16, 26, 92
sum', 12, 13, 15
  definitie, 12, 13
sums, 3, 9

tail, 12, 13, 18, 38, 40, 68, 71, 86, 90, 91
  definitie, 11, 40
tail segment, 67, 68
tails, 67, 68, 70, 71, 83, 90, 91
  definitie, 68, 70, 71
take, 9, 30, 40, 41, 43, 45, 49, 53, 76, 88
  definitie, 30, 41
takeWhile, 5, 43, 48, 50, 65
  definitie, 43
takewhile, 48
tan, 16
Tekst (type), 115
Term (type), 80
Text (type), 115
tMul, 82
toUpper
  definitie, 47
Transpose, 79, 83
transpose, 7, 75, 77, 78, 83
  definitie, 75
transposing, 72, 75
Tree (type), 58, 59, 62
Tree2 (type), 65
tuple, 52–56
Turing, Alan, 1
type
  Bool, 14–17, 45, 62
  Char, 15, 45, 46, 50
  Complex, 54
  Float, 15, 17, 28, 30, 31, 33, 45, 53, 54, 80
  Fraction, 63
  Int, 5, 14–17, 23, 28, 45, 50, 80
  IntOrChar, 62
  Nat, 107
  Point, 54
  Poly, 79, 80
  polymorphic, 37
  String, 115
  Tekst, 115
  Term, 80
  Text, 115
  Tree, 58, 59, 62
  Tree2, 65
  vector, 71
type (keyword), 54
type definitions, 54
type variable, 16

uncurry, 56, 64
  definitie, 56
unlines, 47
until, 26, 27, 32–34, 45, 49, 64
unwords, 47

Vec, 71, 75, 76
vecInprod, 74, 76, 77
  definitie, 74
vecLength, 74
  definitie, 73
vecPerpendicular
  definitie, 74
vecPlus, 72, 74
vecScale, 71, 74
vector, 71–79
  addition, 74
vector (type), 71

weekday, 29, 30
  definitie, 41
where, 13
where (keyword), 1, 10, 11, 13, 14, 27, 28, 31, 89
while (keyword), 26
words, 47

zero, 4, 33, 34
  definitie, 33
zip, 56, 75, 83
  definitie, 56
zippWith, 74
zipWith, 56, 74, 75, 78, 82
  definitie, 56

Appendix F

Answers

1.1 The meanings of *gofer* are: 1. message boy; 2. waffle. The meanings of *gopher* are: 1. burrowing rat-like animal in N America; 2. rascal; 3. victim. By the way: the name of the programming language has no meaning in this sense, it is an abbreviation of **go**od **f**or **e**quational **r**easoning.

1.2

`=>`	reserved symbol (shown in list)
`3a`	nothing (names always start with a letter)
`a3a`	name
`::`	reserved symbol (shown in list)
`:=`	operator [constructor]
`:e`	nothing (instruction is not reserved)
`X_1`	name [constructor]
`<=>`	operator
`a'a`	name
`_X`	nothing (names always start with a letter)
`***`	operator
`'a'`	nothing (names always start with a letter)
`A`	name [constructor]
`in`	reserved keyword (shown in list)
`:-<`	operator [constructor]

1.3 `4000.02`, `60.0`, `200000.0`

1.4 `x=3` defines a constant (for instance in a `where` clause). `x==3` is the Boolean expression (with the value `True` or `False`).

1.5 The two functions are:

```
numberSol a b c  |  d>0  =  2
                 |  d==0 =  1
                 |  d<0  =  0
                    where  d  =  b*b-4.0*a*c
numberSol' a b c  =  1 + signum (b*b-4.0*a*c)
```

1.6 You can de-activate parts of your program by commenting it out, even if this part contains comments itself.

1.7 Ask the interpreter for the type, by for instance typing `:type tail`. Specify the types yourself by:

```
tail  ::  [a] -> [a]
sqrt  ::  Float -> Float
pi    ::  Float
exp   ::  Float -> Float
(^)   ::  Num a => a -> Int -> a
(/=)  ::  Eq a  => a -> a -> Bool
numberSol :: Float -> Float -> Float -> Int
```

1.8 Respectively `False` and `True`. In C the first expression is false, coded in C as `0`. The second expression however, means in C '`x` is devided by three', with the value `2` and the side effect that `x` is assigned this value. (In C inequality is written as `!=`).

1.9 'Syntax error' means 'this is no expression'. With a syntax error the symbols of a formula are not in the right order. With a type error there is an expression, but the types of the parameters do not match those of the operator or function being used.

1.10 `3::Int` and `even::Int->Bool`, so `even 3::Bool`. You have to check if the parameter 'fits' the function; if so, the result is of the type as specified in the function. With polymorphic functions it is more or less the same: `head::[a]->a` and `[1,2,3]::[Int]`; so if fits, because `[Int]` is a special case of `[a]`. But then the `a` should be equal to `Int`. So the type of `head [1,2,3]` is not `a`, but `Int`.

1.11

a. Variables in a patterns are bound to a *natural* number (so not to an arbitrary integer). So the pattern `x+2` stands for 'a number being at least 2'.

b. The operator `*` only makes a pattern according to the definition if a pattern is to the righ, and a (constant) natural number to the left.

c. Patterns satisfying this rule are easily inverted by the interpreter: the pattern n`*x` needs to be divided by the parameter it is called with, and the pattern `x+`n needs to subtract n of the actual parameter. Patters like `(x+2)*(y+2)` would have to check all (infinitely many) combinations of values for `x` and `y`.

d. When calling a function the inverse function of the pattern is determined. The square root function would look like:

```
sqrt (x*x)  =  x
```

Of course, this concerns a hypothetical function. Gofer has no built-in oracle...

1.12

a. when calling `fac (-3)` first `fac (-4)` will be calculated, which will need the value of `fac (-5)`, ad infinitum... The 'base value' 0 is never reached, and an infinitely long calculation will be the result.

b. The ordering of which parameter is simpler, should be limited to below, and the base case should handle this limit.

1.13 In a list there is also an order of the elements, and the number of times an element occurs.

1.14

a.

```
x ^ 0       =  1
x ^ (2*n)   =  square (x^n)
x ^ (2*n+1) =  square (x^n) * x
square x    =  x*x
```

b. The new definition reaches the final answer much faster:

```
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
```

For higher powers than 10 the time gain is even more.

1.15 A list of lists, where each elements contains three copies of the same element:

```
[ [0,0,0], [1,1,1], [3,3,3], [6,6,6], [10,10,10] ]
```

2.1 The expression `x+h` in its whole is a parameter to `f`. If there would be no parentheses, `f` would only be applied to `x`, because function application has the highest priority. That is why `f x` needs no parentheses. Because `/` is of higher priority than `-`, the left parameter of `/` needs parentheses too.

2.2

- The first pair of parentheses is redundant, because function application associates to the left. The second pair however, is needed because otherwise the first `plus` would get four parameters.
- Both pairs of parentheses are redundant. The first because parentheses around separate parameters are not needed (although in this case there has to be a space between `sqrt` and `3.0`, because otherwise it would denote the identifier `sqrt3`). The second pair is redundant because function application has a higher priority than addition (you could not have known that, but you could have tried it).
- The parentheses surrounding the numbers are redundant. The parentheses around the operator are not, because the operator is used in prefix notation here (as a kind of function).
- Parentheses are redundant, because multiplication has a higher priority from itself then addition.
- Parentheses are not redundant, because addition has a lower priority than multiplication.
- The second pair of parentheses is redundant, because `->` associates to the right. The first pair *is* needed, because here it means a function with a function parameter. and not a function with three parameters.

2.3 By the right association a^{b^c} means the same as $a^{(b^c)}$: First calculate b^c, then raise a to that power. If involution would associate to the left, then a^{b^c} would be equal to $(a^b)^c$, but that can be written without parentheses already as $a^{b \cdot c}$. So association to the right saves parentheses.

2.4 The operator `.` is indeed associative, because

```
   ((f.g).h) x
=  (f.g) (h x)
=  f (g (h x))
=  f ((g.h) x)
=  (f.(g.h)) x
```

2.5 Because parentheses are needed in expressions like `0<x && x<10`.

2.6 `valueIn0`, `plus`, `diff`.

2.7 Because `na` associates to the right, it is allowed to surround the right arrow and its parameters with some extra parentheses. This results in:

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

From this it shows that `after` is a function with only one parameter of type `b->c` and a result of the type `(a->b)->(a->c)`. In the next definition the function `na` has indeed only one parameter:

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

2.8

2.9 For the function f with $f\ x = x^3 - a$ holds $f'\ x = 3x^2$. The formula for `improve b` in section 2.4.5 is in this case simplified in the following way: page 33

$$
\begin{aligned}
& b - \frac{f\ b}{f'\ b} \\
= \ & b - \frac{b^3-a}{3b^2} \\
= \ & b - \frac{b^3}{3b^2} + \frac{a}{3b^2} \\
= \ & \tfrac{1}{3}(2b + a/b^2)
\end{aligned}
$$

Then the function `cubicRoot` would be:

```
cubicRoot x  =  until goodEnough Improve 1.0
```

```
where  Improve y     =  (2.0*b + a/(b*b)) / 3.0
       goodEnough y  =  y*y*y ~= x
```

2.10 By making use of the lambda notation the function `f` needs not to be named separately:

```
inverse g a  =  zero (\x -> g x - a)
```

2.11 Because the simplification process of `improve b` needs the function which should be inverted to be accessible in a symbolic way. If the function which should be inverted is a parameter, this representation is not known; the only thing left is to try the function in (many) locations; this is done in `zero`.

2.12 . . .

3.1 Write `[1,2]` as `1:(2:[])` and apply the definition of `++` three times:

```
   [1, 2] ++ []
=  (1:(2:[])) ++ []
=  1 : ((2:[]) ++ [])
=  1 : (2 : ([]++[]))
=  1 : (2 : [])
=  [1, 2]
```

3.2 `concat = foldr (++) []`

3.3 Expressions 4, 5 and 6 give `True`.

3.4 The function `box` puts its parameter as singleton in a list if it satisfies `p` and otherwise delivers the empty list:

```
box x  |  p x        =  [x]
       |  otherwise  =  []
```

3.5 `repeat = iterate id where id x = x`

3.6 . . .

3.7 . . .

3.8 `qEq (x,y) (p,q) = x*q==p*y`

3.9 The result of $(a{+}bi){*}(c{+}di)$ can be computed by eliminating the parenthesis: $ac{+}adi{+}bci{+}bdi^2$. Because of $i^2 = -1$ this equals $(ac{-}bd) + (ad{+}bc)\, i$. For the solution of $1/(a{+}bi)$ we solve x and y from $(a{+}bi) * (x{+}yi) = (1{+}0i)$. This results in two equations with two unknowns: $ax - by = 1$ en $ay + bx = 0$. The first equation gives $y = (-b/a)x$. Substituting in the first equation gives $ax + \frac{b^2}{a}x = 1$ and thus $x = \frac{a}{a^2+b^2}$ and $y = \frac{-b}{a^2+b^2}$.

```
type Complex = (Float,Float)
cAdd, cSub, cMul, cDiv :: Complex -> Complex -> Complex
cAdd (a,b) (c,d)  =  (a+c, b+d)
cSub (a,b) (c,d)  =  (a-c, b-d)
cMul (a,b) (c,d)  =  (a*c-b*d, a*d+b*c)
cDiv (a,b) (c,d)  =  cMul (a,b) (c/k, -d/k)
                       where k = c*c+d*d
```

3.10 The value of `stringInt ['1','2','3']` is $((0 \star \text{'1'}) \star \text{'2'}) \star \text{'3'}$, in which the operator $\star$ is the operation ‘multiply the left value with 10 and add the `digitValue` of the next character’. You start at the left side and thus you can use `foldl`:

```
stringInt  =  foldl f 0
       where  f n c  =  10*n + digitValue c
```

In this case `foldr` can not be used, because the operator is not associative. Apart from that, this is also an example in which the types of the two parameters of the operator are not the same. That is allowed, just look at the type of `foldl`.

3.11

```
curry :: ((a,b)->c) -> (a->b->c)
curry f x y  =  f (x,y)
```

3.12 Just like `search searchTree` fails if the wanted value is not in the collection, because they is no definition for `searchTree Leaf z`. If you want, you can add a line that delivers a special value depending on the application. The special value could even be an extra parameter to `searchTree`.

```
searchTree      ::  Ord a  =>  Tree (a,b) -> a -> b
```

```
searchTree (Tree (x,y) li re) z  |  z==x  =  y
                                 |  z<x   =  searchTree li z
                                 |  z>x   =  searchTree re z
```

3.13 The function `map` transform a function into a function from *lists* of its domain and codomain respectively. As `map` has the type `(a->b) -> ([a]->[b])`, the type of `map map` has brackets surrounding domain and codomain:

```
map map  ::  [a->b] -> [[a]->[b]]
```

3.14 `until p f x = hd (dropWhile p (iterate f x))`

3.15 The operator `<` on lists can be defined with the following recursive definition:

```
xs     < []      =  False
[]     < ys      =  True
(x:xs) < (y:ys)  =  x<y || (x==y && xs<ys)
```

The order in which the first two lines are put matters. As the lines are tried in order, the comparison of two empty lists will use the first line which indeed the intention.

3.16 A definition of `length` with the help of `foldr` is:

```
length  =  foldr op 0
    where  x ‘op‘ n  =  n+1
```

We start with the initial result 0, and the operator `op` adds for each element one to the intermediate result. The parameter `n` is the number of elements in the remaining part of the list, which is already counted by the recursive call in the definition of `foldr`. The parameters of the operator `op` do not have the same type: the left parameter is an arbitrary type (the type of the list elements does not matter), the right parameter and the result are of type `Int`. So the type of the function which is passed to `foldr` is: `a -> Int -> Int`. It is indeed the case that the function `foldr` accepts functions with the type `a -> b -> b`.

3.17 The function `qsort` can be defined as:

```
qsort  ::  Ord a  =>  [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<=x) xs)
               ++ [x] ++
               qsort (filter (>x) xs)
```

The difference with `msort` is that the function takes two arbitrary sub-lists, which have to be merged after the recursive sorting, while `qsort` invests its time in the selection of elements which are smaller (or larger), so that they can be simply concatenated after they are sorted.

3.18 The function `group` can be defined as follows:

```
group :: Int -> [a] -> [[a]]
group n  =  takeWhile (not.null)
            . map (take n)
            . iterate (drop n)
```

As a parameter of `takeWhile` we use `not.null` instead of `/=[]`, because usage of `/=` needs the lists to be comparable (an instance of Eq). The function `null` is defined in the prelude by

```
null [] = True
null xs = False
```

3.19 The function `mapTree` applies a given function to all elements which are stored in the leafs:

```
mapTree :: (a->b) -> Tree2 a -> Tree2 b
mapTree f (Leaf2 x)   =  Leaf2 (f x)
mapTree f (Node2 p q) =  Node2 (mapTree f p) (mapTree f q)
```

The function `foldTree` applies a given operator in each `Node`:

```
foldTree :: (a->a->a) -> Tree2 a -> a
foldTree op (Leaf2 x)    =  x
foldTree op (Node2 p q)  =  op (foldTree op p) (foldTree op q)
```

3.20 The inductive version:

```
depth (Leaf2 x)   =  0
depth (Node2 p q)  =  max (depth p) (depth q)
```

The version using `map` and `fold`:

```
depth  =  foldTree max . mapTree 0
```

3.21 We write an auxiliary function, with as an extra parameter the depth (so the number of prefixed spaces) of the tree.

```
showTree  =  showTree' 0
showTree' n Leaf  =  copy n ' ' ++ "Leaf\n"
showTree' n (Node x p q) = showTree' (n+5) p
                          ++ copy n ' ' ++ show' x ++ "\n"
                          ++ showTree' (n+5) q
```

3.22 If the tree is entirely crooked, it contains the minimum number of leaves $n+1$ A completely filled tree has 2^n leaves.

3.23 We divide the list in two (almost) equal parts and a separate element. Then we recursively apply the function to the two halves. That gives us two trees with approximately the same depth. Those trees are combined into one tree with the separate element as the root:

```
makeTree []  =  Leaf
makeTree xs  =  Node x (makeTree as) (makeTree bs)
         where  as     = take k xs
                (x:bs) = drop k xs
                k      = length xs / 2
```

4.1 Not only the first seven and the last four elements are swapped. Due to the effect in the recursive call the other seven elements are also ordered differently:

$$\texttt{segs}\ [1,2,3,4] = [\ [1]\ ,\ [1,2]\ ,\ [1,2,3]\ ,\ [1,2,3,4]\ ,\ [2]\ ,\ [2,3]\ ,\ [2,3,4]\ ,\ [3]\ ,\ [3,4]\ ,\ [4]\ ,\ []\ ]$$

4.2 `segs = ([]:) . filter (/=[]) . concat . map inits . tails`

4.3 $(n+1)$, $1+(n^2+n)/2$, 2^n, $n!$, $\binom{n}{k}$.

4.4 Everywhere the function `subs` chooses for '`x` in the result' `bins` chooses for 'a 1 on the result'. Everywhere the function `subs` chooses *not* to incorporate `x` in the result, `bins` chooses for 'a 0 in the result':

```
bins 0      =  [ [] ]
bins (n+1)  =  map ('0':) binsn  ++  map ('1':) binsn
        where  binsn = bins n
```

4.5 It (`gaps`) can be solved both recursively as with the use of other functions:

```
gaps []      =  [ ]
gaps (x:xs)  =  xs  :  map (x:) (gaps xs)
gaps' xs     =  zipWith (++) (init(inits xs)) (tail(tails xs))
```

4.6 A matrix describing a mapping from a p-dimensional space has p columns and q rows. If we write $M(p,q)$ for such a matrix, the 'type' of a matrix multiplication becomes:

$$(\times) :: M(q,r) \rightarrow M(p,q) \rightarrow M(p,r)$$

The order of the variables is the same as in the type of function composition:

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

(Matrices cannot be typed with respect to their dimension in Gofer. This is because lists have the same type, regardless of their length. They can be distinguished by using tuples instead of lists, but then we will need to provide a set of functions for every dimension.)

4.7
$$\begin{aligned}
& \det \binom{a\ b}{c\ d} \\
= \ & \texttt{altsum}\ [\ a * \texttt{det}(d)\ ,\ b * \texttt{det}(c)\ ] \\
= \ & \texttt{altsum}\ [\ a * d,\ b * c\ ] \\
= \ & a * d - b * c
\end{aligned}$$

4.8 It is the easiest to use an example to see which functions must be performed after each other: So the function is:

```
matInv (Mat m)  =  ( Mat
                   . zippWith (*) pmmp
                   . mapp ((/d).det.Mat)
                   . map (gaps.transpose)
```

$$\begin{matrix}147\\258\\369\end{matrix}$$

$\downarrow$ `transpose`

$$\begin{matrix}123\\456\\789\end{matrix}$$

$\downarrow$ `gaps`

$$\begin{bmatrix}\begin{matrix}???\\456\\789\end{matrix}, \begin{matrix}123\\???\\789\end{matrix}, \begin{matrix}123\\456\\???\end{matrix}\end{bmatrix}$$

$\downarrow$ `map transpose`

$$\begin{bmatrix}\begin{matrix}?47\\?58\\?69\end{matrix}, \begin{matrix}1?7\\2?8\\3?9\end{matrix}, \begin{matrix}14?\\25?\\36?\end{matrix}\end{bmatrix}$$

$\downarrow$ `map gaps`

$$\begin{bmatrix}\begin{bmatrix}\begin{matrix}???\\?58\\?69\end{matrix}, \begin{matrix}?47\\???\\?69\end{matrix}, \begin{matrix}?47\\?58\\???\end{matrix}\end{bmatrix},\\ \begin{bmatrix}\begin{matrix}???\\2?8\\3?9\end{matrix}, \begin{matrix}1?7\\???\\3?9\end{matrix}, \begin{matrix}1?7\\2?8\\???\end{matrix}\end{bmatrix},\\ \begin{bmatrix}\begin{matrix}???\\25?\\36?\end{matrix}, \begin{matrix}147\\???\\36?\end{matrix}, \begin{matrix}147\\25?\\???\end{matrix}\end{bmatrix}\end{bmatrix}$$

$\downarrow$ `mapp ((/d).det.Mat)`

$$\begin{array}{l}[\,[D_1/d,\ D_2/d,\ D_3/d]\,,\\ \ \ [D_4/d,\ D_5/d,\ D_6/d]\,,\\ \ \ [D_7/d,\ D_8/d,\ D_9/d]\,]\end{array}$$

$\downarrow$ `zippWith (*) pmmp`

$$\begin{array}{l}[\,[+D_1/d,\ -D_2/d,\ +D_3/d]\,,\\ \ \ [-D_4/d,\ +D_5/d,\ -D_6/d]\,,\\ \ \ [+D_7/d,\ -D_8/d,\ +D_9/d]\,]\end{array}$$

```
                  . gaps
                  . transpose
                  ) m
          where   d = det (Mat m)
                  pmmp = iterate tail plusMinusOne
```

4.9 The function `cp` combines the elements of a *list of* lists in all possible ways. We write the functions with induction. If there is only one list, the elements form the 'combinations of one element'. So:

```
crossprod [xs]  =  map singleton xs
```

The function `singleton` can be written as `(:[])`. For the recursive case we look at an example, for instance the list [[1,2] , [3,4] , [5,6]]. A recursive call of `crossprod` on the tail results in the list [[3,5], [3,6], [4,5], [4,6]]. These elements must be combined in all possible ways with [1,2]. This results in the definition:

```
crossprod (xs:xss)  =  cpWith (:) xs (crossprod xss)
```

The definition of the base case can be written differently too:

```
crossprod [xs]  =  cpWith (:) xs [[]]
```

What the `crossprod` of an empty list should be, is not exactly clear. But if we define

```
crossprod []  =  [[]]
```

the definition of a list with one element equals that one of a list with more elements:

```
crossprod [xs]  =  cpWith (:) xs (crossprod [])
```

With this the definition has got the structure of `foldr`, so it can also be written as:

```
crossprod  =  foldr (cpWith (:)) [[]]
```

The length of the `crossprod` of a list of lists is the product of the lengths of the lists:

```
lengthCrossprod xs  =  product (map length xs)
```

That is why it is called *cross product.*

4.10 If we take care that the tails are never filled with zeroes, the degree is one less than the length of the list (except for the 0-polynomial):

```
pDeg []  =  0
pDeg xs  =  length xs - 1
```

You will then need a simplification function, removing tails with zeroes:

```
pSimple []      =  []
pSimple (x:xs)  | x==0.0 && tail==[]  =  []
                | otherwise           =  x : tail
           where tail  =  pSimple xs
```

To add two polynomials corresponding terms can be added, after which the result is simplified:

```
pAdd xs ys  =  pSimple (zipWith (+) xs ys)
```

Defining `pMul` is done by induction to the first polynomial. If this is the 0 polynomial, the result will be the 0 polynomial. Otherwise every term of the second polynomial needs to be multiplied by the first element (the constant term) of the first polynomial. The rest of the first polynomial is multiplied by the second polynomial, after which all exponents are increased by one. The latter can be done by putting a 0 in front of the result The two resulting polynomials can be added using `pAdd`. The result needs not to be simplified, since `pAdd` takes care of that:

```
pMul [] ys  =  []
pMul (x:xs) ys  =  pAdd (map (x*) ys)
                        (0.0 : pMul xs ys)
```

5.1 The version from the question is more efficient, because `map (x:) (inits xs)` is shorter than `segs xs`, and `++` is linear in its left parameter, but constant in its right.

5.2 The optimalizations are as follows:

	before optimalization		after optimalization	
alg.	kind	complexity	kind	complexity
b.	smaller/1/linear	$\mathcal{O}(n^2)$	half/2/linear	$\mathcal{O}(n \log n)$
c.	smaller/1/const	$\mathcal{O}(n)$	half/1/const	$\mathcal{O}(\log n)$
d.	half/2/const	$\mathcal{O}(n)$	lalf/1/const	$\mathcal{O}(\log n)$
e.	smaller/2/const	$\mathcal{O}(2^n)$	smaller/1/const	$\mathcal{O}(n)$
f.	half/2/linear	$\mathcal{O}(n \log n)$	half/2/const	$\mathcal{O}(n)$

5.3 The operator `++` is the most efficient if it computer right associating (see section 5.1.3.a). This is done with `foldr`. The use of `foldl` is less efficient, because `++` will be calculated left associating. If one of the lists is infinite the result of both `foldr` as `foldl` will be the infinite list. The lists after the infinite list will never be seen, but the infinite list itself will. When using `foldl'` there will never be an answer, because the infinite list will have to be concatenated with the lists coming after that (non-lazy), before you will see te result. That can take quite a long time. . .

page 87

5.4

Law *sum after map-linear-function*
For the sum of a linear function applied to a list of numbers the following law holds:

```
sum (map ((k+).(n*)) xs)  =  k * len xs + n * sum xs
```

Proof with induction to `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` and `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` and `sum`)
		`k*(1+len xs) + n*(x+sum xs)`
		= (distribution `*`)
		`k + k*len xs + n*x + n*sum xs`
		= (`+` commutative)
		`k+n*x + k*len xs + n*sum xs`

5.5 First prove

```
foldr op e (xs++ys)  =  (foldr op e xs) ‘op‘ (foldr op e ys)
```

in the same way as ‘sum after `++`’. After that the proof is the same as ‘sum after concatenation’.

5.6 Assume `e` to be the empty list, and define `g x ys = f x : ys`. Proof with induction to `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)`

5.7 Proof with induction to `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`
		= (property `+`)
		`1 + 0`

`x:xs`	`len (subs (x:xs))`
	= (def. `subs`)
	`len (map (x:)(subs xs) ++ subs xs)`
	= (length after `++`)
	`len (map (x:)(subs xs)) + len (subs xs)`
	= (length after 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)`

5.8

Law *subsequences after map*
For all functions `f` on lists:

```
subs . map f  =  map (map f) . subs
```

Proof with induction to `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 after `++`)
		`map (map f) (map (x:) (subs xs))`
		`++ map (map f) (subs xs)`
		= (map after function composition)
		`map (map f.(x:)) (subs xs)`
		`++ map (map f) (subs xs)`
		= (map after (:))
		`map ((f x:).map f) (subs xs)`
		`++ map (map f) (subs xs)`
		= (map after function composition)
		`map (f x:) (map (map f)(subs xs))`
		`++ map (map f) (subs xs)`

5.9 Proof of law 12, with induction to z:

z	(x*y)^z	x^z * y^z
Zero	(x*y)^Zero = (def. ^) Succ Zero = (def. +) Succ Zero + Zero	x^Zero * y^Zero = (def. ^) Succ Zero * Succ Zero = (def. *) Succ Zero + Zero*Succ Zero = (def. *) Succ Zero + Zero
Succ z	(x*y)^Succ z = (def. ^) (x*y) * (x*y)^z	x^Succ z * y^Succ z = (def. ^) (x*x^z) * (y * y^z) = (* associative) ((x*x^z)*y) * y^z = (* associative) (x*(x^z*y)) * y^z = (* commutative) (x*(y*x^z)) * y^z = (* associative) ((x*y)*x^z) * y^z = (* associative) (x*y) * (x^z * y^z)

Proof of law 13, with induction to y:

y	(x^y)^z	x^(y*z)
Zero	(x^Zero)^z = (def. ^) (Succ Zero)^z = (law 13a, follows) Succ Zero	x^(Zero*z) = (def. *) x^Zero = (def. ^) Succ Zero
Succ y	(x^Succ y)^z = (def. ^) (x*(x^y))^z = (law 12) x^z * (x^y)^z	x^(Succ y*z) = (def. *) x^(z+y*z) = (law 11) x^z * x^(y*z)

Proof of law 13a, used in the proof above, with induction to x:

x	(Succ Zero)^x	Succ Zero
Zero	(Succ Zero)^Zero = (def. ^) Succ Zero	Succ Zero
Succ x	(Succ Zero)^(Succ x) = (def. ^) Succ Zero * (Succ Zero)^x = (def. *) (Succ Zero)^x + Zero*(Succ Zero)^x = (def. *) (Succ Zero)^x + Zero = (law 1) (Succ Zero)^x	Succ Zero