Haskell-Tutorial

Damir Medak
Gerhard Navratil

Institute for Geoinformation
Technical University Vienna

February 2003

There are numerous books on functional programming. These books are good, but usually intended to teach a student about everything their writers had ever learned.

Thus, the reader is confused in the very beginning: Either with the amount of information given, or by the absence of concrete, useful examples applying the theoretical material. Usually, the reader is soon curious about using abstract concepts in programming but has no idea how to start.

We would like to express here the fundamentals of functional languages for beginners. The idea is to empower the reader to really do some coding, producing curiosity for the theoretical background on some later stage. Some familiarity with basic mathematics is assumed.

We have to admit, however, that we ignore two aspects, which might be important for some readers:

- We do dot care about efficiency! We want to specify functions and create models. We do not aim at fast or powerful software but at easy-to-understand models.
- We define no user interface. We do not care about interaction with a user, with other programs or with files. We use the language for modelling only and use the command line for testing.

1 Prerequisites

It is useful to sit in front of the computer while reading this tutorial because then all examples can be tested immediately. It is therefore useful to install Hugs (98) and an adequate editor (e.g. UltraEdit).

1.1 How to install Hugs?

Reading about any programming language is boring if the reader cannot experiment with it. For Haskell, the solution is called Hugs (Haskell User's Gofer System). The latest version can be found in the Internet (www.Haskell.org).

	DOS-version	Windows-version
Command line	*dir* \HUGS.EXE %F	*dir* \WINHUGS.EXE %F
Working directory	%P	%P
Windows program?	Not checked	Checked

Table 1: Parameters to add Hugs as a tool in UltraEdit

The latest version is from November 2002 at the moment (February 2003). Versions for Microsoft Windows, generic Unix, Linux, MacOS X and MacOS 9 are available. We will always relate to the Windows version when talking about the user interface. Read the documentation for the other operating systems. The Windows file is a Windows installer package (msi) and the installation starts immediately after a double-click. It is recommended to read the help file to avoid errors and get additional information.

- Download the file with the recent Hugs-version (at the moment 'hugs98-Nov2002.msi') to your hard disc.
- Run the installation program.
- Check if you already have an editor installed. If you do not have a good one installed yet, do it now.
- If using UltraEdit set the editor option in Hugs (Options/Set ...) to: `UEdit32 %s/%d/1`. If you use a different editor replace "UEdit32" with the name of the editor of your choice and the parameters by the parameters necessary to open the file with the editor and jump to a specified line number.

Table 1 provides the necessary commands for adding Hugs as a tool to UltraEdit. The name *dir* in the command line must be replaced by the directory where Hugs is installed.

1.2 What are we going to learn in the tutorial?

This tutorial is a hard way to learn functional programming, because there are no easy ways to learn anything useful. (If you prefer some other formal method, take any book on first order logic, read a chapter or two, and you will come back.)

Easy exercises will make you comfortable with Hugs (comments and recommendations are welcome!). We will start with simple tasks like the definition of functions and data types and then move on to more elaborate methods like object oriented programming and modularization.

2 Basic Types, or "How to bring Haskell to work for me at all?"

Test the following functions in Hugs - save them in a file, load them into Hugs, and test them.

2.1 Print "Hello, world" on the command line

This is not a big deal, but every language can do it; Haskell has the operation `putstr` to write a string to the screen. Open Hugs and write the following on the command line (the line with a question mark)[1]:

```
putstr "Hello"
```

The result should look like this:

```
Hello
(2 reductions, 7 cells)
?
```

The first line is the result of the command. The second line provides some information on the evaluation of the command. The interpreter replaces the functions with easier functions until each part is a simple step with a direct implementation (similar to other interpreter languages like the original Basic). These replacements are called reductions. The cells are a value for the amount of memory used for the reductions and printing the result.

Open the editor (type ':e' at the command line) and type the same command in the editor window (use the editor-commands 'new file' and 'save file as'). We write it as a function, i.e. we provide a name (`f` in this case) and separate it from the function definition by the character '='. Save the file and give it the extension '.hs'.

```
f = putstr "Hello"
```

Load the file in HUGS[2] and run the function. Running a function is easy in Hugs. You only have to type the name of the function (here: `f`) and the parameters (none in this case) at the command line. The result is:

```
? f
Hello
(3 reductions, 15 cells)
?
```

The difference to the solution above is that the number of reductions is 3 (instead of 2) and the number of cells used is 15 (instead of 7). The number of reductions increased by one because the interpreter must replace the `f` by `putstr "Hello"`. This is an additional reduction which also requires some memory.

2.2 Writing functions in Haskell

In a functional language, everything (even a program) is a function. Thus, writing a program is actually writing a function. You can write many functions in a single file, that is, you can have many programs in a single file. Each function then may use other functions to calculate the result. There are many

[1] "Hello" is a string. Like Pascal Haskell uses double quotes to define start and end of strings. Single characters are marked by normal quotes: String "abc"; Characters 'a', 'b', 'c'

[2] In the Windows-version of Hugs you can use the script manager to load additional files - just say 'Add script', select the file, and click 'Ok' two times. If you use the DOS-version you have to change the current path (using ':cd ...') and load the file using ':l "FileName.Ext"'.

predefined functions available like in other programming languages. The files containing these functions can be found in 'hugs98\lib'.

Writing a function consists of two parts. First we have to define the types for the arguments and the result. A function may have no arguments. This is called a constant function. An example for this is the function `pi` which returns the value of π and needs no arguments to do this. Other functions may have one or more arguments. However, each function has a single result, e.g.:

```
increment :: Int -> Int
```

The name of our new function is `increment`. The '::' separates the name of the function from the type definition(s). If there is more than one type, the types are separated by `->`. The last type in that list is the type of the result[3].

The second part of writing a function is the concrete implementation, e.g.:

```
increment x = x + 1
```

The function increment takes a single argument of type `Int` (an integer number) returning an `Int` as well. To test its functionality, we type "`increment 2`" (without the quotes) on the Hugs-prompt resulting in something like

```
3 :: Int
```

If Hugs writes `3` without the type, use "Options/Set..." and check "show type after evaluation". If that field is unchecked, the interpreter only writes the result without specifying the type.

Let us take another example. Like other programming languages Haskell has a predefined data type for Boolean values. An important operator for Boolean values is the logical and. Although Haskell already knows that function we will implement it again. One solution would be to write is in the following way:

```
and1 :: Bool -> Bool -> Bool
and1 a b = if a == b then a
                     else False
```

Here we use a branch (`if-then-else`) to split the calculation method for the result. If the parameters are equal (either both True or False) we return the value of one of the parameters. If the parameters are not equal we return False. Another method would be the use of pattern matching. The logical 'and' has special a case where the result is True. In all other eases the result is False. We know which parameters the special case has. We can write these parameters in a separate line and add the result. For all other cases we write a second line:

```
and2 :: Bool -> Bool -> Bool
and2 True True = True
and2 x    y    = False
```

In our example we did not use the parameters for calculating the result of the second line. It is not necessary to know the values of the parameters because we have a constant result. In Haskell we can write '_' instead of providing names for these parameters. The advantage for the reader is that he knows the number of the parameters and also knows that the result is independent of the values of these parameters. The second line would then look like the following:

```
and2 _ _ = False
```

[3]If writing a Pascal-function this would read like: function Increment (i : Integer) : Integer

2.3 Calculate roots of quadratic equations (learn about `where` and `if then else`)

The first step for writing functions is the definition of the problem. We have to define the signature of the solution.

First, remember that the function you are writing is actually a program - use sketches, notes, books, and previously written examples while programming. It is not the activity of typing - it is a work of art.

A quadratic equation looks like $a_2x^2 + a_1x + a_0 = 0$. The equation has two roots (or one if the roots are equal). The mathematical solution looks like the following:

$$x_1 = \frac{-b}{2a} - \frac{\sqrt{b^2 - 4ac}}{2a}, \quad (1)$$

$$x_2 = \frac{-b}{2a} + \frac{\sqrt{b^2 - 4ac}}{2a}. \quad (2)$$

A function calculating the roots has a triple of values as an input and a tuple of values as an output. If we use the type `Float` for the values, the definition in Haskell is:

```
roots :: (Float, Float, Float) -> (Float, Float)
```

The name of the function is `roots`. The '::' separates the name of the function from the type definitions. The characters '`->`' separate the different type definitions. The last type definition is the result (output) of the function while the others are parameters (input). Thus the type definition `(Float,Float,Float)` is the parameter of our function. It is a representation of a triple of `Float` values. The result of the function is `(Float, Float)`, a pair of `Float` values.

The next step is the definition of the function roots:

```
roots (a,b,c) = (x1, x2) where
  x1 = e + sqrt d / (2 * a)
  x2 = e - sqrt d / (2 * a)
  d  = b * b - 4 * a * c
  e  = - b / (2 * a)
```

The result of the function is a pair `(x1,x2)`. Both values, `x1` and `x2` are defined by local definitions (after the where-clause). The local definitions calculate parts of the result. In general it is useful to create a local definition for a partial calculation if the result of that calculation is needed more than once. The result of the local definitions `d` and `e`, for example, is necessary for both parts of the solution (`x1` and `x2`). The solutions themselves could have been written directly in the tuple. However, defining them as local definitions improves the readability of the code because the symmetry of the solution is clearly visible.

The formulas used are the well-known formulas for the calculation of the roots for quadratic equations.

This function works, but it takes no care for negative square roots. We can test that with two polynomial triples `p1` and `p2`:

```
p1, p2 :: (Float, Float, Float)
p1 = (1.0, 2.0, 1.0)
p2 = (1.0, 1.0, 1.0)
```

For p1 the result is

```
? roots p1 (-1.0,-1.0) :: (Float,Float)
(94 reductions, 159 cells)
```

For p2, however, the output is

```
? roots p2
( Program error: {primSqrtFloat (-3.0)}
(61 reductions,183 cells)
```

The notation `Program error:` shows that there was an error at run-time. The bracket above that text is visible because Haskell started to print the result before the error occurred. The bracket is the beginning of the text for the resulting pair. The text after the error notation specifies the error. Here, the text says, that the function `primSqrtFloat` cannot handle the parameter -3.0 which is negative and, therefore, has no real square root. The function `primSqrtFloat` is the hardcoded implementation of the function `sqrt` for the type `Float`.

We have to modify the code to handle exceptions like the non-existence of real numbers as the roots:

```
roots (a,b,c) = if d < 0 then error "sorry" else (x1, x2)
  where x1 = e + sqrt d / (2 * a)
        x2 = e - sqrt d / (2 * a)
        d  = b * b - 4 * a * c
        e  = - b / (2 * a)
```

The function now tests the value if the local definition for `d` and writes an user-defined error if `d` is negative. Otherwise the function calculates the roots like before.

There are several explanations about this solution:

- The `if-then-else`-construct works like "if the condition is fulfilled (True) then do the first operation (after `then`), in all other cases do the second Operation (after `else`)".
- `where` is part of the previous line, because it is indented and the next four lines must be equally indented because they are on the same level.
- How is it possible to calculate `sqrt d` in the line `x1` in case that `d < 0.0`? The answer: It is never calculated if `d` is smaller than zero, because functional programs evaluate only those expressions that are necessary for the result:
  - The program needs `d`, calculates `d`, if it is `< 0.0`, the program gives the error message and ends.
  - If `d` is not `< 0.0`, it calculates `x1`, needs `e` for the calculation, calculates `e`, calculates `x2` and ends.

Functions should be tested for behavior. If the interpreter does not complain with an error message, our functions satisfied all syntax rules (e.g., all opened brackets match the closed brackets). If there is an error message when loading or

running a functions and it contains the word `type`, something is certainly wrong with the type structure of the function: It does not match the specification in the signature, etc. Finally, a syntactically correct function needs not be correct semantically: lt may give wrong results or no results at all. We should test new functions with some (easy) examples.

2.4 Some predefined types

Table 2 provides an overview on the most important predefined types in Haskell.

Bool	True, False
Int	-100, 0, 1, 2,...
Integer	-33333333333, 3, 484305638045783294578945 7,...
Float/Double	-3.22425, 0.0, 3.0,...
Char	'a' , 'z', 'A', 'Z , ';', ...
String	"Medak", "Navratil", ...

Table 2: Predefined types

2.5 Summary

- A function has a signature: A specification declaring the types of argument(s) and result
- The indentation matters:
  1. Top level declarations start in the first column and their end is the next declaration starting in the first column.
  2. A declaration can be broken at any place and continued in the next line provided that the indent is larger than the indent of the previous line.
  3. If the keyword `where` is followed with more than one declaration, all of them must be on the same level of indentation.
- When writing, be clear as much as possible (use descriptive names for functions). For example, a function name called `matrixMult` will be easier to understand than `mM`.
- Comments are nevertheless useful - everything written after '`--`' is a comment
- Functions are tested with examples we easily can check by hand.

Exercise: Extend the program for complex numbers (a complex number is a pair of real numbers).

3 Lists, or "How something so simple can be useful?"

In the previous exercise we met the data type pair `(x1,x2)` and triple `(a,b,c)` (or, in general, tuples). Pairs have exactly two elements and triples have exactly three. We may have also tuples with four, five and more elements, but the number of elements will always be fixed. The elements of tuples may have different types, however. For example, the first element of a pair may be a name (a string) and the second the age (an number).

What is a list? A list provides a way to deal with an arbitrary number of elements of the same type. A list can have no elements at all: empty list denoted as `[]`. It can contain any type, but all elements must be of the same type. In other words: A list is a one-dimensional set of elements of the same type (numbers, strings, Boolean values...). Examples for lists are:

`[]`	... empty list
`[1, 2]`	... list with elements of type integer
`[True,False]`	... list with elements of type Boolean
`[(1, 2), (2, 3)]`	... list of tuples of Integers
`[[1, 2], [2, 3, 4]]`	... list of Integer-lists

`Strings` are lists, too:

```
"Name" = ['N' , 'a' , 'm', 'e']
```

Access on list elements is provided on the first element (the head of the list) and all other elements (the tail of the list) only. Therefore, if we need access to other elements we have to read the list recursively. The following function, for example, provides access to the n^th element of a list:

```
nthListEl 1 1 = head 1
nthListEl 1 n = nthListE1 (tail 1) (n-1)
```

3.1 Recursion: The fundamental principle of functional programming, or "Who stole loops and counters"

The standard faculty function (!), defined in mathematics in terms of itself:

$$fact\ n = \begin{cases} 1 & : \quad n = 0 \\ n * fact(n-1) & : \quad n \neq 0 \end{cases} \tag{3}$$

Faculty is a rather simple example for mathematical induction - Zero-case and induction step (successor case):

```
1) p(0),
2) p(n) => p(n+1)
```

We can translate this syntax easily to Haskell:

```
fact 0 = 1                  -- zero step
fact n = n * fact (n-1)     -- induction step
```

How does this program work? Let us take a look at an example, the faculty of 6: Since `n==6` and not zero, the induction step is applied: `6 * fact 5`. Because `n==5` and not zero, the induction step is applied: `6 * (5 * fact 4)`, and so on, until `n==0`, when the zero step is applied and the recursion terminates.

```
fact 6   ==> 6 * fact 5
         ==> 6 * (5 * fact 4)
         ==> 6 * (5 * (4 * fact 3))
         ==> 6 * (5 * (4 * (3 * fact 2)))
         ==> 6 * (5 * (4 * (3 * (2 * fact 1))))
         ==> 6 * (5 * (4 * (3 * (2 * (1 * fact 0)))))
         ==> 6 * (5 * (4 * (3 * (2 * (1 * 1)))))
         ==> 6 * (5 * (4 * (3 * (2 * 1))))
         ==> 6 * (5 * (4 * (3 * 2)))
         ==> 6 * (5 * (4 * 6))
         ==> 6 * (5 * 24)
         ==> 6 * 120
         ==> 720
```

This reasoning is based on a single important fact: That the complete set of natural numbers including zero (N_0) is covered by two cases:

1. A natural number is either zero (zero ease).
2. Or a successor of another natural number (induction step).

The purpose of this short trip to the world of mathematics is to make easier to grasp the similar case of recursion over lists in Haskell. The two cases for lists are:

1. An empty list
2. A non-empty lists consisting of an element and the rest (possibly an empty list)

For complete definitions we must know an important function that "glues" together elements to the list - (:) pronounced as "cons" for construct. It has the following type:

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

and works like this:

```
1 : 2 : 3 : [] = [1,2,3]
```

Now, the two cases, that allow the recursive definition of lists, are:

[]	an empty list
(x: xs)	a list with at least one element

```
List = [] | (a : List)
```

This means that a list is either empty or contains an element followed by a list (which again may be empty or ...). The type variable `a` tells us that we may

have any type for the list elements[4]. The structure of a list does not depend on the type of data stored in the list. The only restriction is that all elements of the list must be of the same type. We may have lists of integer numbers, lists of Boolean values, or lists of floating point numbers but we may not create a list holding five Integers and two Booleans. With this definition it's possible to write functions on lists, for example a function which determines the sum of all elements of a list:

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

We separate the first list element recursively from the rest of the list until we have an empty list. We know the result for the empty list (it is zero). We then add the elements of the list in reverse order of the separation process. This means the last element in the is the first we add to zero, and so on. After processing all elements of the list we have the sum of the list elements as the result of the list. We can use the list constructor also to rewrite the function for reading the n-th element of a list:

```
nthListEl' (l:ls) 1 = l
nthListEl' (l:ls) n = nthListEl' ls (n-1)
```

3.2 Basic functions for lists

`head`	returns the first element of a list
`tail`	removes the first element of the list
`length`	returns the length of the list
`reverse`	reverses the list
++ (`concat`)	concatenates 2 lists: [1,2]++[3,4]=[1,2,3,4]
`map`	applies a function to each element in the list
`filter`	returns all list elements that fulfill a specified condition
`foldr`	combines the elements of a list with a specified function and a start value (e.g. adds all elements)
`zip`	takes two lists and combines the corresponding elements (the elements at the same position) by making pairs
`zipWith`	takes two lists and combines the corresponding elements (the elements at the same position) by applying the specified function

Table 3: Useful list functions

There are several important functions to work with lists. Table 3 shows some of the most important list functions and explains briefly what they do.

Some of the functions (like e.g. `map`) are especially interesting, because they have a function as an argument - we say that these functions are higher order functions. In the following sections we will use some of these functions.

[4]This concept is called polymorphism. More information can be found in section 5

3.2.1 Code your name into ASCII numbers and back (do you know what `map` does?)

Suppose we have a string and need the ASCII-code for each character of the string. A string in Haskell is a list of characters. Therefore it is obvious to use list functions to solve the problem. The function `ord` should be applied to each single character because it provides the calculation of the ASCII number. Since a string is a list of characters coding a string into a list of ASCII numbers requires applying `ord` to each character of the string. This is exactly what `map` does. The type definition of `map` is:

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

The function `map` takes two arguments, a function and a list, and returns a list. The input list has elements of an arbitrary type `a` and the resulting list has elements of an arbitrary type `b`. The function has a parameter of type `a` and returns a value of type `b`. In our example the types `a` is a character and the type `b` is an integer number. The implementation of `map` is:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

It applies the function `f` recursively to all elements of the list and creates a new list by applying the list constructor ':'.

Coding a string to a list of ASCII numbers now can be written as follows:

```
code :: String -> [Int]
code x = map ord x
```

After reloading the Definition file into Hugs we may test the new function:

```
code "Hugs"
```

The result is:

```
[72,117,103,115]
```

3.2.2 Calculation of the area of a polygon

Let us assume we need to calculate the area of a polygon. The polygon is a list of points with x and y coordinates. The Gaussian formula for the area is:

$$2F = \sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1}) \tag{4}$$

We now want to use list functions to calculate the area. We start with a definition of the polygon. In our case it is a list of points and each point is a pair of coordinates. A polygon for testing purposes could look like the following (which has the area 10000 as we can see immediately):

```
p = [(100.0,100.0),(100.0,200.0),(200.0,200.0),(200.0,100.0)]
    :: [(Float,Float)]
```

The function `p` returns a simple polygon which we can use for testing our function (the area of this polygon should be 10000).

In the first step we have to think about restructuring our list in a way that allows applying the list functions. It is clear that the last step will be calculation of the sum of partial areas and division by two. The function `foldl` combines the elements of a list with a specified function and uses a specified value as the staring point. We want to add the elements and therefore we must use '+'. Since '+' is defined to stand between the arguments (x + y) we must write it as '(+)'. The expression '(+) x y' is the same as 'x + y'. The starting point for our function is zero. Therefore the function can be written as:

```
sum_parts :: [Float] -> Float
sum_parts parts = (foldl (+) 0 parts) / 2
```

We now must find a way to calculate the partial areas $(x_i - x_{i+1})(y_i + y_{i+1})$. We would like to have four lists with the following contents:

list	contents
1	x_1, x_2, x_3, x_4
2	x_2, x_3, x_4, x_1
3	y_1, y_2, y_3, y_4
4	y_2, y_3, y_4, y_1

Table 4: Resulting lists

This would allow using `zipWith` for calculating the partial sums. The function `zipWith` can be defined as follows (I changed the name to `zipWith'` to avoid conflicts with the function already defined):

```
zipWith' :: (a -> a -> a) -> [a] -> [a] -> [a]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (a:as) (b:bs) = (f a b) : zipWith' f as bs
```

The first line defines the type of the function. The function takes two lists and a function (used for combining the elements of the lists). The next two lines stop the recursion if one of the lists is empty. The last line then shows how the function works. It takes the first elements of the lists and applies the function `f` to these elements. The result is an elements of the resulting list.

The first part of our remaining formula tells us to take the x-coordinates of the first point and subtract the x-coordinates of the second point from them $(x_i - x_{i+1})$. If we use the lists from table 4 we take the elements of the first list and subtract the elements of the second list from them. Therefore the line

```
zipWith (-) list1 list2
```

provides the subtraction for alle elements in the lists. Therefore the complete formula looks like

```
zipWith (*) (zipWith (-) list1 list2) (zipWith (+) list3 list4)
```

The only problem remaining is not the creation of the lists. The lists 1 and 3 are rather simple. They are the x- and y-coordinates to the points. We can use

`map` and apply two other functions, `fst` and `snd` to each element of the list. `fst` takes a pair and returns the first value. `snd` also takes a pair but remains the second value. If we assume that the coordinates are given in the order (x,y) we get these lists with

```
list1 poly = map fst poly
list3 poly = map snd poly
```

where `poly` contains the polygon. Since formula (4) also works if we change x and y, it is not important whether the coordinates for the polygon are given as (x,y) or as (y,x).

The other two lists are a little bit more difficult. In both cases the list is rotated. We therefore have to take the first element and move it to the back of the list. This can be done by

```
moveback poly = tail poly ++ [head poly]
```

The functions `head` and `tail` split the list into the first element (`head`) and the remainder of the list (`tail`), which is a list itself. Adding edged brackets around an expression creates a list with the expression as the element[5]. The function '++' (or `concat` for concatenate) takes two lists and merges them by putting the elements of the second list behind the last element of the first list.

A combination of all these parts provides the final function, where we also add error expressions if we have less than three points (and therefore no area):

```
area :: [(Float,Float)] -> Float
area []    = error "not a polygon"
area [x]   = error "points do not have an area"
area [x,y] = error "lines do not have an area"
area ps = abs ((foldl (+) 0.0 parts) / 2) where
  parts = zipWith (*) (zipWith (-) l1 l2) (zipWith (+) l3 l4)
  l1 = map fst ps
  l2 = tail l1 ++ [head l1]
  l3 = map snd ps
  l4 = tail l3 ++ [head l3]
```

3.3 Other useful list functions

As seen in section 3.2 functions can also be arguments to other functions. An example for this is the function `map` which we used in section 3.2.1

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

For the function definition we need not know explicitly which types we use. During runtime, however, it is important to know both types. The type of the parameter is known at runtime (we then work with existing data sets which are of a specific type). The type of the result however must not be unambiguous. In this ease we would have to specify the type of result we need by adding '::*Type*'.

[5]or elements if we have more expressions, separated by commas

The next essential list function is the filter function. We rename (to avoid a collision with the function defined in the prelude) and implement it:

```
filter2 :: (a -> Bool) -> [a] -> [a]
filter2 f []     = []
filter2 f (x:xs) = if f x then x : filter2 f xs
                          else filter2 f xs
```

The filter function takes two arguments. The first argument is a function (the filter) and the second is a list of an arbitrary type `a`. The filter takes a list element and returns a Boolean value. If the value is `True`, the element remains in the list, otherwise it is removed from the resulting list.

3.4 Calculation of roots for a list of quadratic equations

We start with a list of quadratic equations (stored as triples of real numbers like in section 2.3). The task splits into two parts. We must assure that all equations have real solutions because if there are complex solutions our function creates an error. We therefore need a function that returns `True` if the solutions are real and `False` if the solutions are complex:

```
real :: (Float, Float, Float) -> Bool
real (a,b,c) = (b*b - 4*a*c) >= 0
```

We can now use that function to filter the equations with real solutions:

```
p1 = (1.0,2.0,1.0) :: (Float, Float, Float)
p2 = (1.0,1.0,1.0) :: (Float, Float, Float)
ps = [p1,p2]
newPs = filter real ps
rootsOfPs = map roots newPs
```

The function `ps` returns a list with two quadratic equations. The first equation (`p1`) has real solutions while `p2` has complex solutions. The function `newPs` uses `real` to filter the elements of `ps` that have real solutions. Finally, `rootsOfPs` applies the function `roots` to each equation in the list. The result then is the list of solutions for the equations.

Exercise: Use the function that calculates complex roots (from the last exercise) and apply it to a list of quadratic equations.

3.5 Alternative function definition

It is always possible to write a function in different ways. The differences are usually in the abstraction of the definition or in the functions used. It is also possible to use different styles for dealing with special cases. The following example defines functions for the length of a list. Although they look rather different they all produce the same result. In all cases the abstraction is the same (all have a parameter of type `[a]` and a result of type `Int`).

```
l1 [] = 0
l1 (x:xs) = 1 + l1 xs
```

```
l2 xs = if xs == [] then 0 else 1 + l2 (tail xs)

l3 xs | xs == []  = 0
      | otherwise = 1 + l3 (tail xs)

l4 = sum . map (const 1)

l5 xs = foldl inc 0 xs
  where inc x _ = x+1

l6 = foldl' (\n _ -> n + 1) 0
```

The special case for the length of a list is the empty list. in this case the length is zero. The functions `l1` to `l3` start from this fact and use iteration to calculate the length. However, they use different methods for detecting the empty list:

- `l1` uses pattern matching
- `l2` uses an if-then-else branch
- `l3` uses guard notation (see (6.2) for more information)

The functions `l4` to `l6` use other list functions to perform the calculation. They work in the following way:

- `l4` first replaces all elements in the list with `1` and then just sums the elements of the list.
- `l5` defines a local function for increasing a local counter and uses the function `foldl` to go through the list.
- `l6` uses the same method as `l5` but defines the function in a lambda-expression[6]

3.6 Summary

We found out a few things about lists in this sections. The most important points have been:

- Lists are an excellent way for representing an arbitrary number of elements provided that all elements must be of the same type.
- There is a number of useful functions over lists in the *prelude.hs*.
- The higher-order function `map` applies a function to all elements of a list.
- Mathematical induction is the basic theoretical background for recursion.
- Recursion replaces loop constructs in imperative programs in a mathematically sound way.

[6]We will not use lambda-expressions in this tutorial. The definitions was only added because this is the definition used in the Haskell-prelude.

- Lists are isomorph to natural numbers and recursion is applied in the same manner.
- `map`, `filter` and `foldl`/`foldr` are the most important functions over lists and are useful in many applications.
- There are usually many ways to define a function in Haskell.

4 Representation of Data

Until now we only used the base types for our functions. Usually it is necessary to define data structures to store the data necessary for our functions.

4.1 Type synonyms

Let us continue with the roots example. The input for roots is a triple `(Float, Float,Float)` and the output is a pair `(x1,x2)`. The type signature for roots looked like:

```
roots :: (Float, Float, Float) -> (Float, Float)
```

If we had a number of functions in our program, such type information would not be very helpful in understanding what that function really does. It might be better to read something like:

```
roots :: Poly2 -> Roots2
```

That is why Haskell supports type synonyms. Type synonyms are user-defined names for already existing types or their combinations in tuples or lists. The lines

```
type Poly2 = (Float, Float, Float)
type Roots2 = (Float, Float)
```

mean that we can refer to any triple of `Floats` using the name `Poly2` and to a tuple of floats using the name `Roots2`. Both names say more about the type than the original tuple (the first one tells us that it contains a polynomial of the second order and the second one contains the roots of such a polynomial).

Summary: Type synonyms are just shortcuts for already existing data types which should make clearer what the program does or what input and output a function has.

4.2 User-defined data types

Combining predefined types (`Bool`, `Int`, `Float`, `Char`) in lists (`[]`) or tuples (pairs, triples, ...) is usually powerful enough to solve a number of problems we deal with. However, sometimes we need our own definitions to express:

- Tuples in more convenient form: `Poly = (Float,Float,Float)`
- Enumerated types: The days of week are Mon or Tue or Wed or Thu or Fri or Sat or Sun
- Recursive types: `Num` (N_o) is either `Zero` or a successor of `Num`

User-defined data types play an important role in the class-based style of programming and it is worth learning how they work.

```
data Polynom = Poly Float Float Float
```

Explanation:

- `data` is the keyword to start a data type definition
- `Polynom` is the name of data type (type information)
- `Poly` is the constructor function (try `:t Poly`)
- `Float` is the type of the first, second, and third argument of `Poly`

This means that we can rewrite the function `roots2` (to avoid the name clash with `roots`):

```
roots2 :: Polynom -> (Float, Float)
roots2 (Poly a b c) = ...

pOne, pTwo :: Polynom
pOne = Poly 1.0 2.0 1.0
pTwo = Poly 2.0 4.0 (-5.0)
```

The brackets around the number `-5.0` in the definition of `pTwo` are necessary because the sign for negative numbers is the same as the function name for subtraction. Without the brackets Haskell would misinterpret the '-' and would give the following error:

```
ERROR D:\Texte\Latex\code.hs:116 - Type error in explicitly typed binding
*** Term           : pTwo
*** Type           : Float -> Polynom
*** Does not match : Polynom
```

The most important thing is the distinction between the type name (`Polynom`) and the type constructor (`Poly`). The type name comes always in lines concerning the type information (containing '::'), and the type constructors in lines concerning application (containing '='). Type constructors are the only functions starting with a capital letter, and the only functions that can appear on the left-hand side of an expression.

One can define a data type with the same name for the type and for the constructor, but such practice is highly discouraged!

IMPORTANT: Finally, another portion of syntactic sugar: To avoid problems with printing results of functions, just add to any new definition of data type the following two words: `deriving Show`. This creates automatically an instance of the class[7] `Show` for the new data type

The following code gives an example for a recursive data type. Try it on the command line:

```
:t Start, :t Next, :t Next (Bus), :i Bus, testbus, testint

data Bus = Start | Next (Bus) deriving Show
```

[7] see chapter 8 for information on classes and instances.

```
myBusA, myBusB, myBusC :: Bus

myBusA = Start
myBusB = Next (Next (Next (Start)))
myBusC = Next myBusB

plus :: Bus -> Bus -> Bus
plus a Start    = a
plus a (Next b) = Next (plus a b)

testBus :: Bus
testBus = plus myBusC myBusB

howFar :: Bus -> Int
howFar Start    = 0
howFar (Next r) = 1 + howFar r

testInt :: Int
testInt = (+) (howFar myBusC) (howFar myBusB)
```

4.3 Parametrized Data Types

Data types can be parametrized with a type parameter, which must be instantiated with a type when used. A simple example for such a data type is a list:

```
data List a = L a (List a) | Empty
```

This data type defines a list as a recursion. The starting point is the empty list. Any other list has an element as the head of the list and a list as it's tail. The elements of the list are arbitrary elements but all elements within the list must have the same type. Some examples for this data type are:

```
l1,l2,l3 :: List Integer
l1 = Empty
l2 = L 1 l1
l3 = L 5 l2
li1 = L 1.5 Empty :: List Double
```

5 Polymorphism

Polymorphism means that a single function can be applied to a variety of argument types. Typical examples in other programming languages are mathematical operations like plus or minus. Usually the following expressions are valid for Integer numbers and floating point numbers. In C++-Notion we would say that the functions are "overloaded".

```
a + b
a - b
```

5.1 Ad hoc Polymorphism

If the two operations have just the same name, we call it ad hoc polymorphism. It is somewhat confusing, as two completely different operations, with different properties can be using the same name (e.g. '+' to concatenate strings is not commutative $a + b \neq b + a$). This is not desirable and programmers should watch out not to introduce such sources of error (the readers expectation for '+' is different from what is programmed).

5.2 Subset Polymorphism

A common type of polymorphism is based on a subset relation of the data types. We postulate a most general type (say number) from which subtypes (Integer, Float etc.) is constructed. We then say that the operation plus applies to all objects of type number and therefore also to the subtypes.

Subtype relations in Haskell cannot be defined directly. However, it is possible to demand, that there must be instances of specific classes for the used data type. In our roots example we would need real numbers for our parameters and the result. Since an implementation of real numbers is not possible in computer systems we have to work with an approximation (e.g. floating point numbers). However, we can demand that specific functions must be defined for the used data type - addition, multiplication, square roots, ... These functions are defined in the Haskell classes `Num` (numbers), `Fractional`, and `Floating`. The last one requires the instances of the other two classes. Therefore it is sufficient to demand the implementation of `Floating`:

```
roots :: (Floating a) => (a, a, a) -> (a, a)
```

Adding '`(Floating a) =>`' does not create the necessary instance! However, it can check the existence of the instance during run-time (when it is clear which data types are used) and give an error message `Unresolved Overloading` if a necessary instance is missing.

```
? roots (1,2,1)
ERROR - Unresolved overloading
*** Type       : Floating Integer => (Integer,Integer)
*** Expression : roots (1,2,1)
?
```

5.3 Parameterized Polymorphism

The solution used in Haskell is 'parameterized polymorphism'. An operation is applicable to all situations

```
plus :: a -> a -> a,
```

where the type parameter `a` is replaced consistently with the same type - thus for

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

or

```
plus:: Rat -> Rat -> Rat.
```

Correspondingly, data types can also have a type parameter (e.g. `data List a`).

This gives enormous power, as operations can often be expressed in a general form, not depending on the particulars of the parameter selected. For example, one can count the length of a list without knowing the type of data stored in the list.

```
listlen :: List a -> Int
listlen Empty      = 0
listlen (L _ list) = 1 + listlen list
```

6 Advanced Programming Methods

6.1 Composition

We can compose functions. In the last example we applied two functions to a list. The mathematical expression for this is $f(g(x))$. In mathematics we could also write $(g \circ f)(x)$. The key condition is that the result type of the function 'g' coincides with the argument type of the function 'f'. Haskell has a similar mechanism. The composition operator '.' takes two functions and combines them (`((f . g) x = f (g x))`)

```
rootsOfPs2 = (map roots.filter real) ps
```

The dot replaces the circle. The function `filter real` is `g`, the function `map roots` is `f`, and the variable `ps` is `x`.

6.2 Guards

Guards are another method to write `if-then-else`-expressions. Let us assume that we have the following example:

```
tst1 (a,b,c) = if d > 0
                 then d
                 else if d==0 then 0
                              else error "negative sqrt"
     where d = b*b - 4*a*c
```

This means the same as the 'guard'-notation:

```
tst2 (a,b,c) | d >  0    = d
             | d == 0    = 0
             | otherwise = error "negative d"
     where d b*b - 4*a*c
```

If the first test ($d > 0$) returns `True` the result is `d`. Otherwise the second line is checked. The result here is `0`. The expression `otherwise` in the third line is predefined and returns `True` in any case. If, therefore, the first two checks return `False` the result of the function will be an error. Thus, the result of the function `tst2` is the same as the result of the function `tst1`.

Another example where guards are useful are discontinuous functions. Let us assume that we have a function that is `-1` for negative values, `0` for zero, and

1 for positive values (this is the `signum`-function). The mathematical definition looks like the following:

$$signum\ x = \begin{cases} -1 & : & x < 0 \\ 0 & : & x = 0 \\ 1 & : & x > 0 \end{cases} \tag{5}$$

In guard notation we write (after changing the name to `sign` to avoid clashes with the existing function `signum`):

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

We see that there are only minor differences between the mathematical notation and the Haskell syntax. The vertical bars replace the winged bracket and condition and result are in reverse order in Haskell.

6.3 Sections

Standard numeric functions take two arguments and give a single result:

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

Their application looks like:

```
? 3 + 4
7
```

but may be written:

```
? (+) 3 4
7
```

If we specify the first argument together with the operator, the type signature is:

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

In other words, the first argument of '(+)' is "eaten" by the integer 3. This property is useful in map constructs:

```
? map (3+) [4,7,12]
```

Equality (==) and ordering operators (<, <=, >, >=) represent another set of functions that are commonly used in the section form. Since they all give a Boolean value as a result, their sections are useful filter constructs:

```
? filter (==2) [1,2,3,4) [2]
? filter (<=2) [1,2,3,4] [1,2]
```

Note the profund difference in the position of the operator and the argument in the section. It does not make any difference if using symmetric operation like but it makes a big one for ordering operations - try different things like

```
(<2) (2<) (>2) (2>)
```

for `p` in the form `filter p [1,2,3,4]` and see what happens.

Summary: Sections are the way to convert a function to a new function with less arguments than the original one. It is often used to convert a function with more than one argument to a function which is acceptable for mapping, filtering, and functional composition.

6.4 List comprehensions

List comprehensions are yet another device for defining lists and operations on them (usually found in textbooks about functional programming). The form should be completely natural if you think about lists in terms of set theory.

In set theory, a set of odd integers between 0 and 100 will be represented by the set: $\{x|x \in 0..100, \mathrm{odd}\, x\}$. In functional language, the same set is written as:

```
[x | x <- [0..100], odd x]
```

Thus, braces are replaced by square brackets, the '$\in$' is replaced by '`<-`' and you can define almost anything expressible as a set.

For example, to multiply all elements of one list with all elements of another list, we write a function like this:

```
? f [2,3,4] [5,6] where f xs ys = [x*y | x <- xs, y <- ys]
[10,12,15,18,20,24] :: [Integer]
```

Actually, map and filter can be expressed in terms of list comprehension:

```
map    f xs = [f x | x <- xs]
filter p xs = [x   | x <- xs, p x]
```

7 Functional vs. Imperative Programming

Example: Calculation of inner product for vectors:

The vectors a and b are one-dimensional vectors with elements $a_1, a_2, ..., a_n$ and $b_l, b_2, ..., b_n$. The inner product of a and b then is a sum of products of matching components: $a \cdot b = a_l \cdot b_l + a_2 \cdot b_2 a_n + ... + b_n$.

An imperative program will probably look like this:

```
c := 0
for i:= 1 to n do
  c := c + a[i] * b[i]
```

We can see some parts where errors may occur. We need a variable to store provisional results. This variable will be active within the whole function and, therefore, may be used on a different place, too. This makes the code harder to read and may cause errors if the programmer is not careful. We also must determine the number of elements within a vector prior to the for-do-loop because this is the second parameter of the loop. Finally, we loose contact to the mathematical notation ($\sum_{i=1}^{n} a_1 \cdot b_i$)

For a functional program there are a few versions. Here is one of the shortest and most elegant:

```
inn2 :: Num a => ([a] , [a]) -> a
inn2 = foldr (+) 0 . map (uncurry (*)) . uncurry zip
```

There are several things that look quite strange in these two lines. The first line should be clear when ignoring the `Num a =>`. This part tells the interpreter, that the arbitrary type `a` must be a number. `Num` is a class containing functions for adding, subtracting, multiplying, etc. It is not useful to implement these functions for data types which are no numbers. Therefore, the data type is a number if there is an implementation of `Num` for that data type. This is exactly what the interpreter does if lie finds something like `Num a =>`.

In the second line there is no parameter for the function. The complete line would look like this:

```
inn2 x = (foldr (+) 0 . map (uncurry uncurry zip) x
```

If we compare the the parts on the left and right side of the '=' we find that the parameter `x` is the last element in both cases. Therefore we may let the parameter away without consequences. This does not only reduce the work for writing, but it also let us concentrate on the most important part - the algorithm.

The second line consists of three functions that are combined by the composition operator:

- `uncurry zip`: The function `zip` takes two lists as parameters and combines the lists in the following way: It takes the first elements of both lists and makes a pair. This is the first element of the resulting list. Then it takes the second elements of both lists and makes a pair (which is the second element of the resulting list), and so on. The function `uncurry` modifies the function `zip` so, that it has not two separate parameters but a single pair. An example for the use of `uncurry` can be found below (function `ut`).

- `map (uncurry (*))`: This part takes the list of tuples created by the first step and multiplies the elements of the tuples. For a single tuple `uncurry (*)` executes the multiplication. The use of `map` applies that function to each tuple in the list.

- `foldr (+) 0`: This function finally sums the results of the multiplications.

For better understanding try the following examples and look what they do:

```
ft = foldr (*) 7 [2,3]
ut = uncurry (+) (5,6)
zt = zip "Haskell" [1,2,3,4,5,6,7]
testvec = inn2 ([1,2,3], [4,5,6])
```

The next step is to expand the function to a list of vectors. The parameter therefore is a list of vectors (which are a list of integer numbers). The result is the inner product of all vectors:

```
innXa, innXb :: [[Integer]] -> Integer
innXa = foldr (+) 0 . map (foldr (*) 1) . transpose
innXb = foldr1 (+) . map (foldr1 (*)) . transpose
```

The functions `innXa` and `innXb` do the same. Both consist of three parts. The difference is that `innXb` uses the function `foldr1` instead of `foldr`. `foldr1` uses

the zero element defined for the parameter function. In case of the multiplication the zero element is `1`, for the addition it is `0`.

The function `transpose` takes a list of lists as a parameter and creates a new list of lists. The first list contains the first elements of all input lists, the second lists contains the second elements of all input lists, and so on.

```
transpose :: [[a]] -> [[a]]
transpose []              = []
transpose ([]:xss)        = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:t) <- xss]) :
                          transpose (xs : [t | (h:t) <- xss])
```

The following example can be used to test the function.

```
x = [[1,2,3],[4,5,6],[1,1,1]]
testa = innXa x
testb = innXb x
```

We can now list some advantages of functional program over imperative programming:

- It operates only on its arguments (no side-effects).
- It is hierarchical, being built from simpler functions.
- It employs the functions that are generally useful in other programs.
- It is static and non-repetitive.
- It is completely general.
- It does not name its arguments.

8 Building a simple database

A database is a collection of objects with identifiers and attributes. We will use Integers as identifiers. They must be unique within the database. Attributes are functions from value sets ('name', 'age', 'color') to values ("John", "1211", "red").

The operations needed are:

- insert (adding a new object with some attributes to a database)
- select (given an ID, retrieves the object from a database)
- selectby (retrieves the objects satisfying the given condition)

The goal is to look at the specification development process: We define functions with signatures containing arbitrary data types and group them. The resulting groups are called classes. Later we define these functions for specific data types (instances of classes).

We give default definitions for general operations that do NOT depend on implementation details. We have already seen a function that does not depend on implementation - `len` in section 3.1.

If we work like that, we can hand our classes (without the instances) to different programmers and the results of their work will be compatible (although the implementations will differ).

For simplification, we introduce two type synonyms for identifiers and attributes. The identifier is an Integer number. An attribute consists of two Strings. The first defines the type of the attribute and the second defines the value of the attribute.

```
type ID = Int
type Attrib = (String, String)
```

The class `Objects` defines the behavior of objects. We define, that an object consists of an identifier of type `ID` and a list of attributes of type `Attrib`. The operations `object` (creates a new object), `getID` (returns the ID of the object) , `getAttrib` (returns the attributes of the object) are implementation dependent and, therefore, require a definition in an instance before use. However, the operation `getName` (returns the value of the Attribute "name") is independent of the implementation and we can give the default definition (an axiom).

```
class Objects o where
  object  :: ID -> [Attrib] -> o
  getID   :: o -> ID
  getAtts :: o -> [Attrib]
  getName :: o -> String
  getName = snd . head . filter (("name"==) . fst) . getAtts
```

The class `Databases` specifies the behavior of a collection of objects. It is not important what the type of the objects is, provided that there are operations for that type (defined in the class `Objects`). Again, the first five operations are implementation dependent.

```
class (Objects o) => Databases d o where
 empty :: d 0
 getLastID :: d o -> ID
 getObjects :: d o -> [o]
 setLastID  :: ID -> d o -> d o
 setObjects :: [o] -> d o -> d o

 insert     :: [Attrib] -> d o -> d o
 insert as db = setLastID i' db' where
   db' = setObjects os' db
   os' = o : os
   os  = getObjects db
   o   = object i' as
   i'  = 1 + getLastID db

 select :: ID -> d o -> o
 select i = head . filter ((i==).getID) . getObjects

 selectBy :: (o -> Bool) -> d o -> [o]
 selectBy f = filter f . getObjects
```

Now we have a complete specification of a simple database. The next step is to write an implementation to test the specification. The implementation here is only one of many possibilities:

First of all we need a particular representation of objects. We define a data type `Object` holding the identifier and the Attributes and connect the data type to the class `Objects` by implementing the class for the data type. We only have to implement the functions that have no axiom in the class definition.

```
data Object = Obj ID [Attrib] deriving Show

instance Objects Object where
  object i as = Obj i as
  getID (Obj i as) = i
  getAtts (Obj i as) = as
```

We see that all dependent functions have a direct connection to the data type. All functions either read data from the data set and return that data as the result or write data to the data set and return the data set as a result. Therefore, the functions are easy to implement. The complex tasks (like reading the value of a specific attribute) are part of the class definition.

Now we have to do the same for the database. The data set for the data base consists of an identifier and a list of objects. The identifier stored in the data set is the last identifier used for an object. It is therefore easy to calculate the identifier for a new object.

```
data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
  empty = DB 0 []
  getLastID (DB i os) = i
  setLastID i (DB j os) = DB i os
  getObjects (DB i os) = os
  setObjects os (DB i ps) = DB i os
```

We can now start testing the database. For testing purposes we use examples like the ones below. The tests are specific for the selected implementation.

```
d0, d1, d2 :: DBS Object
d0 = empty
d1 = insert [("name", "john"),("age", "30")] d0
d2 = insert [("name", "mary"),("age", "20")] d1

test1 :: Object
test1 = select 1 d1
test2 :: [Object]
test2 = selectBy (("john" ==).getName) d2
```

9 Modules

The programs we dealt with so far were short and compact, fitting on a single sheet of paper. If we want to do some serious programming, it is very likely that

our program will increase in size. An overview of a lengthy file is a difficult task. Therefore, long projects are divided in parts, containing pieces of a program that belong together. These parts are called modules (not only in Haskell).

How to write a module? It is simple, just add a line containing the keyword `module`, the actual name of the module (capitalized!), followed by the keyword `where`, and that's it. Note that this line should be the first line in your Haskell script that is not a comment or empty. An example:

```
-- filename: Test.hs
-- project : Haskell Script
-- author  : Damir Medak
-- date    : 10 June 1999

module Test where
data Person = Pers String Int
-- etc.
-- end of file
```

Some conventions are useful:

- Put exactly one module in a single file.
- Give the same names to files and modules.

Thus the example above should be saved as *Test.hs*.

Barely collecting Haskell definitions into named modules would not have improved the power of expression we want to achieve. The most important thing to learn is how to import definitions from other modules into a module. This is important both for importing already existing functions in numerous libraries delivered with Haskell and for modularization of our own definitions. An example for each usage will do.

9.1 Importing already existing definitions

A good exercise in module handling is to use an already defined function provided by a module different from the standard prelude (*prelude.hs*). As an example, consider the function `sort`, defined in the script '*\hugs\Lib\List.hs*'. We must specify that the module `List` (surprisingly named exactly as the script - '*List.hs*') is imported by our module.

```
module Test01 where
import List
xs = [2,1,4,3,5]
ys = sort xs
--end of file
```

Save a module as '*Test01.hs*' and load it to Hugs. Observe which files are loaded. If there is a nasty error message "`module List not previously loaded`", type "`:s i+`" followed by 'ENTER' in the interpreter and the message will not come up again.

9.2 Combining files into projects

There is a simple rule to follow when writing the import declarations: Each module must import all modules containing functions used within its definitions.

If our modules are fully independent, we may have a single project file that imports all other files like this:

```
module Project where
import Part01
import Part02
import Part03
--end of import
```

where declarations of parts are:

```
module   Part01 where
--etc.
module Part02 where
--etc.
module Part03 where
--etc.
```

Usually, some functions will be shared among all modules. In this case, each module must import the module where the function is defined. An example:

```
module Part01 where
class Persons p where
  getName :: p -> String
  getAge  :: p -> Int
data Person = Pers String Int
instance Persons Person where
  getName (Pers n a) = n
  getAge  (Pers n a) = a
--end of Part01

module Part02 where
import Part01
class Persons s => Students s where
  getID :: s -> Int
data Student Stud String Int Int
instance Persons Student where
  getName (Stud n a i) = n
  getAge (Stud n a i)  = a
instance Students Student where
  getID (Stud n a i) = i
--end of Part02

module Project where
import Part01
import Part02
--end of Project
```

You can easily check what happens if you forget to write one of the import-lines. Just comment out the second line in the file '*Part02.hs*' and reload the file '*Project.hs*'. You will get an error message.

Summary: Modularization is beneficial in functional programming for several reasons. Program units get smaller and easier to manage, control over connections among various modules is transparent for future use and error detecting, and a number of files can be loaded in a convenient way of loading the project file only.