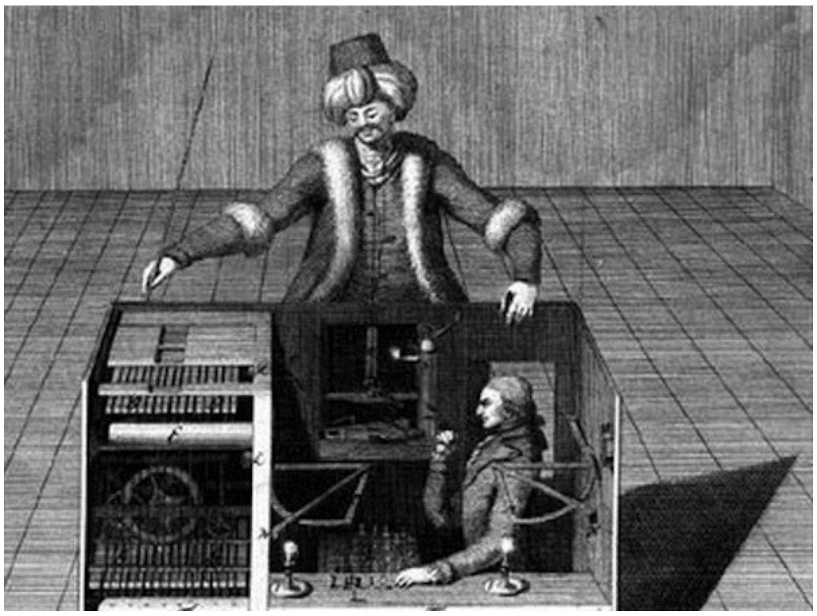
Purely Functional Algorithm Specification

– Version September 1, 2014 –

Jan van Eijck



Contents

Chapter 1

Why Learn Haskell?

Abstract

This is an introductory chapter, to be skipped or skimmed over by those who are already acquainted with Haskell. From Chapter 2 on, I will assume that you know Haskell, or are willing to figure out tricks of the language for yourself.

Key words: Functional programming, functional algorithm design.

1.1 A Short History of Haskell

In the 1980s, efforts of researchers working on functional programming were scattered across many languages (Lisp, Scheme, SASL, KRC, Hope, Id, Miranda, ML,. . .).

In 1987 a dozen functional programmers decided to meet in order to reduce unnecessary diversity in functional programming languages by designing a common language [5]. The new language should be:

- based on ideas that enjoyed wide consensus;
- suitable for further language research as well as applications, including building large systems;
- freely available (in particular: anyone should be permitted to implement the language and distribute it to whomever they please).

The new language was called *Haskell*, after the logician and mathematician Haskell Brooks Curry (1900–1982). Curry is known for his work on the lambda calculus and on combinatory logic. The lambda calculus is the foundation of Haskell. It is well-known that the pure lambda calculus is not an appropriate foundation for computation, but the so-called lazy lambda calculus is. See [1] for details (not for the fainthearted).

In 1990, the first *Haskell* specification was published [5].

Right now, Haskell has a flourishing (and very friendly) user community, and many enthousiastic supporters. If asked why learning Haskell is a good idea, they have things like this to say:

> Haskell is a wide-spectrum language, suitable for a variety of applications. It is particularly suitable for programs which need to be highly modifiable and maintainable.
>
> Much of a software product's life is spent in specification, design and maintenance, and not in programming. Functional languages are superb for writing specifications which can actually be executed (and hence tested and debugged). Such a specification then is the first prototype of the final program.
>
> Functional programs are also relatively easy to maintain, because the code is shorter, clearer, and the rigorous control of side effects eliminates a huge class of unforeseen interactions.
>
> From: `http://www.haskell.org/haskellwiki/Introduction`

For the first decade or so of its existence, Haskell was a pure research language, only known to academics. But this is rapidly changing now. Whether Haskell is with us to stay, no-one knows. Haskell may yet become immortal, but it also " …may just be a passing fancy, that in time will go." In any case, the principles on which the language are built are forever.

1.2 Literate Programming

We will use literate Haskell in what follows. The Haskell code that we mention in this chapter is collected into a so-called Haskell module. See [6] for the benefits of literate programming.

```
module WLH

where

import Data.List
```

You can find the module `WLH.hs` on the book website, at address `http://www.homepages.cwi/~jve/books/pfas/`.

1.3 How Haskell is Different

Here is a quote from an interview with John Goerzen, one of the authors of *Real World Haskell* [7]. indexGoertzen, John

> Question: One of Haskell's benefits is that you can use it as a purely functional language – but that's really different for people who've come up in the imperative or object-oriented worlds. What does it take to learn how to think in a pure fashion?
>
> Goerzen: That's probably the single biggest mind-shift that you deal with coming from a different language; that and laziness. Both of those are both pretty big.
>
> As for how to learn about it, it's a lot of relearning how to do some very basic things and then building upon that. For instance, in imperative languages, you have for loops and you have while loops. There's a lot of having a variable there and then incrementing it as you iterate over something. In Haskell, you tend to take a recursive approach rather than that. It can be a little bit scary at first because you might be thinking if you're using a language such as C, incrementing a variable is a pretty cheap operation.
>
> `http://broadcast.oreilly.com/2009/01/why-you-should-learn-haskell.html`

Haskell allows for abstract, high order programming. (Ideally, more thinking and less writing and debugging.)

Haskell is based on the lambda calculus, therefore the step from formal specification to implementation is very small.

Haskell offers you a new perspective on programming, it is powerful, and it is fun.

The type system behind Haskell is a great tool for writing specifications that catch many coding errors.

Your Haskell understanding will influence the way you look at programming: you will start to appreciate abstraction.

Haskell comes with great tools for automated test generation: a tool we will employ at some point is *QuickCheck* [2], which has served as inspiration for the development of similar tools for many other programming languages.

Haskell is functional A Haskell program consists entirely of functions. Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

The main program itself is a function with the program's input as argument and the program's output as result.

Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

This means that Haskell, like all functional languages, is extensible. If you need a certain programming construct that the language does not provide as a primitive, it is up to you to define it. We will use this feature a lot in this course.

Functions are first-class citizens, which means that they can be used as arguments to other (higher order) functions. This is extremely powerful and extremely useful.

A shift in thinking If you are an imperative thinker, you think mainly of:

- variables as pointers to storage locations whose value can be updated all the time
- sequences of commands telling the computer what to do (how to change certain memory locations) step by step.

Here are some examples:

- initialize a variable `examplelist` of type integer list,
  then add 1, then add 2, then add 3.
- in order to compute the factorial of n, initialize an integer variable `f` as 1, then for all `i` from 1 to n, set `f` to `f`$\times$`i`

If you are a functional thinker, you view bound variables as place-holders for function arguments, and you view unbound variables as identifiers for immutable persistent values, or as names for functions.

Instead of telling the computer what actions to perform in what order, you prefer telling the computer what things are.

Running through the same examples again:

- `examplelist` is a list of integers containing the elements
  1, 2, and 3

- the factorial of n is the product of all integers from 1 to n.

Here is the Haskell code for the factorial function:

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

All the same, this course will stress that functional programming is an *inclusive* paradigm, well capable of expressing the *while* loops, *repeat* loops and *for* loops that play such an important role in pseudo-code presentations of algorithms.

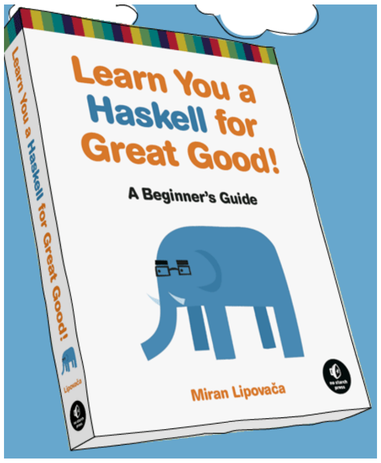
1.4 Where Should I Begin?

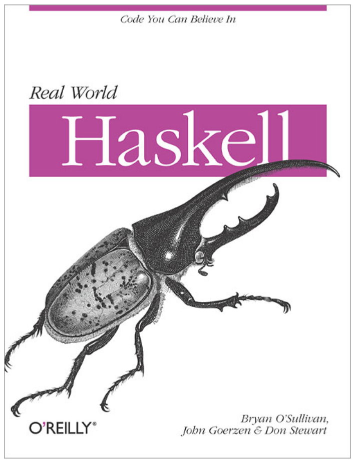
Resources For everything Haskell-related, start at `http://haskell.org`.

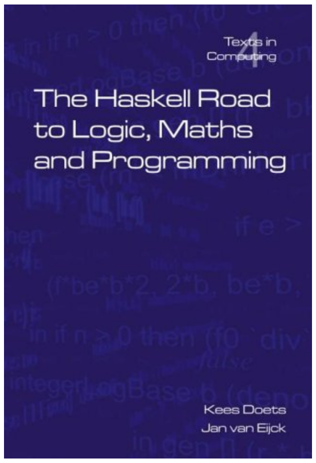
There are lots of free tutorials you may wish to consult:

- Chapter 1 of "The Haskell Road" [4], freely available from `http://homepages.cwi.nl/~jve/HR/`
- *Real World Haskell* [7], `http://book.realworldhaskell.org/read/`
- *Learn you a Haskell for great good*
  `http://learnyouahaskell.comlearnyouahaskell.com`
- *A gentle introduction to Haskell*
  `http://haskell.org/tutorial`.
- The Haskell Wikibook, `secure.wikimedia.org/wikibooks/en/wiki/Haskell`
- Hal Daume's *Yet Another Haskell Tutorial* [3] is also available as a Wikibook, from `secure.wikimedia.org/wikibooks/en/wiki/Haskell/YAHT`.
- Denis Shevchenko's tutorial, in its English translation. See `https://github.com/jhenahan/ohaskell-translations`.

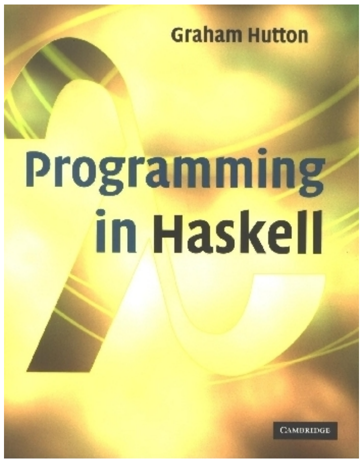
Some recommended books:

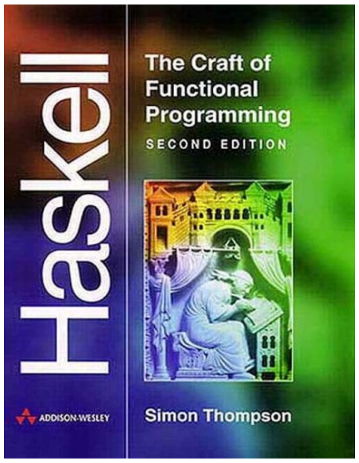


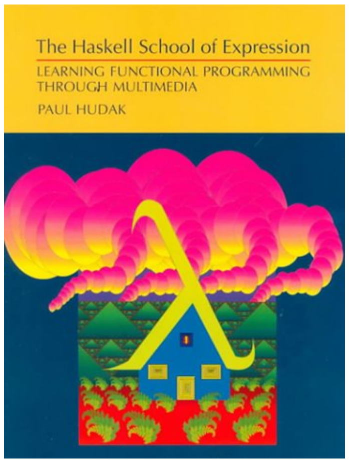




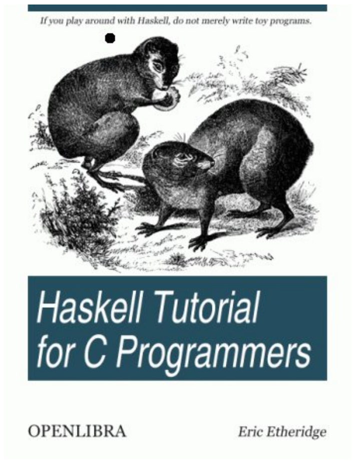
And some more:

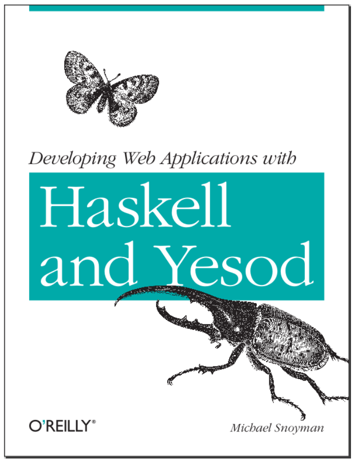


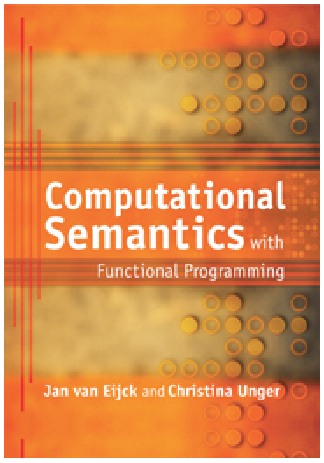



The Haskell School of Expression
LEARNING FUNCTIONAL PROGRAMMING THROUGH MULTIMEDIA
PAUL HUDAK

If you are coming to Haskell from elsewhere, or are into specific applications, one of the following might be for you:


If you play around with Haskell, do not merely write toy programs.
Haskell Tutorial for C Programmers
OPENLIBRA
Eric Etheridge




Computational Semantics with Functional Programming
Jan van Eijck and Christina Unger

1.5 Really Getting Started

Get the Haskell Platform:

- `http://hackage.haskell.org/platform/`

This includes the Glasgow Haskell Compiler (GHC) together with standard libraries and the interactive environment GHCi. Follow the instructions to install the platform.

Haskell as a Calculator Start the interpreter:

```
jve@vuur:~/courses/12/esslli12$ ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The prompt `Prelude>` you are seeing indicates that the so-called Haskell Prelude, consisting of a list of useful predefined functions, is loaded.

GHCi can be used to interactively evaluate expressions.

```
Prelude> 2 + 3

Prelude> 2 + 3 * 4

Prelude> 2^10

Prelude> (42 - 10) / 2

Prelude> (+) 2 3
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs`
   (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.

3. Now you can evaluate expressions like `double 5`, `double (2+3)`, and `double (double 5)`.
4. With `:t` you can ask GHCi about the type of an expression.
5. Leave the interactive environment with `:q`.

Some simple samples of lazy lists "Sentences can go on and on and on (and on)*"

Here is a so-called lazy list implementation:

```
sentence = "Sentences can go " ++ onAndOn

onAndOn = "on and " ++ onAndOn
```

This uses the operation ++ for list concatenation plus the double quote notation for character strings.

If you grab the module `WLH.hs` from `http://www.homepages.cwi/~jve/pfas/` and load it, you will see the following:

```
jve@vuur:~/courses/12/esslli12$ ghci WLH.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling WLH              ( WLH.hs, interpreted )
Ok, modules loaded: WLH.
*WLH>
```

If you now type `sentence` and hit the return (enter) key, you will see that sentences can indeed go on and on and .... You can quit the infinite loop by hitting `^C` (control-C). Next, check an initial segment:

```
*WLH> take 65 sentence
"Sentences can go on and on and on and on and on and on and on and"
*WLH>
```

Next, consider the following:

```
sentences = "Sentences can go on" :
              map (++ " and on") sentences
```

New ingredients here are `:` for putting a item in front of a list, and the function `map` that will be explained below.

The function `sentences` generates an infinite list of sentences. Here is the start of the list:

```
*WLH> take 10 sentences
["Sentences can go on","Sentences can go on and on","Sentences can go
 on and on and on","Sentences can go on and on and on and on","Sentences
 can go on and on and on and on and on","Sentences can go on and on and
 on and on and on and on","Sentences can go on and on and on and on and
 on and on and on","Sentences can go on and on and on and on and on and
 on and on and on","Sentences can go on and on and on and on and on and
 on and on and on and on","Sentences can go on and on and on and on and
 on and on and on and on and on"]
*WLH>
```

Lambda Abstraction in Haskell In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The standard mathematical notation for this is $\lambda x \mapsto x * x$. Haskell notation aims at remaining close to mathematical notation.

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

The types:

```
Prelude> :t (\ x -> x ++ " emeritus")
\x -> x ++ " emeritus" :: [Char] -> [Char]
Prelude> :t "professor"
"professor" :: String
Prelude> :t (\ x -> x ++ " emeritus") "professor"
(\x -> x ++ " emeritus") "professor" :: [Char]
```

Concatenation The type of the concatenation function:

```
Prelude> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

The type indicates that `(++)` not only concatenates strings. It works for lists in general.

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
              (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

The types:

```
Prelude> :t "guy"
"guy" :: [Char]
Prelude> :t (\ x -> "nice " ++ x)
(\ x -> "nice " ++ x) :: [Char] -> [Char]
Prelude> :t (\ f -> \ x -> "very " ++ (f x))
(\ f -> \ x -> "very " ++ (f x))
  :: forall t. (t -> [Char]) -> t -> [Char]
```

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.

- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Turing"` and `"Chomsky"` (note the double quotes).
- In fact, `"Chomsky"` can be seen as an abbreviation of the following character list:

  `['C','h','o','m','s','k','y'].`

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.
- The operation combines an object of type `a` with a list of objects of the same type to a new list of objects, again of the same type.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

List Reversal The reversal of the string ”CHOMSKY” is the string ”YKSMOHC”. The reversal of the string ”GNIRUT” is the string ”TURING”.

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

Reversal works for any list, not just for strings. This is indicated by the type specification `[a] -> [a]`.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded (can be of any size).
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.
- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of *type variables*. For these, `a`, `b`, ..., are used.

Haskell Derived Types Derived types can be constructed in the following way:

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component ...
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.
- By defining your own datatype from scratch, with a `data` type declaration. More about this in due course.

Mapping If you use the command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

This is an example of higher order functional programming, of a function taking another function as an argument.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sections But let us first explain the notation `(^2)`.

- In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its righthand side argument.
- `(x op)` is the operation resulting from applying `op` to its lefthand side argument.
- `(op)` is the prefix version of the operator.
- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

  ```
  [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
  ```

- Similarly, `(>3)` denotes the property of being greater than 3, and `(3>)` the property of being smaller than 3.
- `(++ " and on")` denotes the operation of appending `" and on"` to a string.

Map again If p is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

Here is a definition of `map`, including a type declaration.

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

The code is in `light grey` . This indicates that this definition will not be added to the chapter module. Adding it to the chapter module would result in a compiler error, for `map` is already defined.

Filter A function for filtering out the elements from a list that satisfy a given property:

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

The type declaration and the function definition:

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

List comprehension List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Here are some examples:

```
someEvens     = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens      = [ x | x <- [1..], even x ]
```

Equivalently:

```
someEvens     = filter even [1..1000]

evensUntil n = filter even [1..n]

allEvens      = filter even [1..]
```

Nub The function `nub` removes duplicates, as follows:

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Note the indication `Eq a =>` in the type declaration. This is to indicate that the type `a` has to satisfy some special properties, to ensure that `(/=)`, the operation for non-equality, is defined on it.

Function Composition The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.

Standard notation for this: $f \cdot g$. This is pronounced as "f after g".

Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Note the types!

Elem, all, and

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of `.` for function composition.

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Shakespeare's Sonnet 73

```
sonnet73 =
 "That time of year thou mayst in me behold\n"
 ++ "When yellow leaves, or none, or few, do hang\n"
 ++ "Upon those boughs which shake against the cold,\n"
 ++ "Bare ruin’d choirs, where late the sweet birds sang.\n"
 ++ "In me thou seest the twilight of such day\n"
 ++ "As after sunset fadeth in the west,\n"
 ++ "Which by and by black night doth take away,\n"
 ++ "Death’s second self, that seals up all in rest.\n"
 ++ "In me thou see’st the glowing of such fire\n"
 ++ "That on the ashes of his youth doth lie,\n"
 ++ "As the death-bed whereon it must expire\n"
 ++ "Consumed with that which it was nourish’d by.\n"
 ++ "This thou perceivest, which makes thy love more strong,\n"
 ++ "To love that well which thou must leave ere long."
```

Counting

```
count :: Eq a => a -> [a] -> Int
count x []                 = 0
count x (y:ys) | x == y    = succ (count x ys)
               | otherwise = count x ys
```

Alternative, in terms of `filter` and `length` (a built-in function that computes the length of a list):

```
cnt :: Eq a => a -> [a] -> Int
cnt x = length . (filter (==x))
```

The average of a list of integers, expressed as a rational (fractional) number:

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

Exercise 1.1 *Give your own definition of* `length`.

Some commands to try out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (`elem` "aeiou") sonnet73`
- `count 't' sonnet73`
- `count 't' (map toLower sonnet73)`
- `count "thou" (words sonnet73)`
- `count "thou" (words (map toLower sonnet73))`

Next, attempt the programming exercises from Chapter 1 and 2 of "The Haskell Road" [4].

Expressing the Essence of Recursion over Lists with Fold The pattern of recursive definitions over lists consists of matching the empty list `[]` for the base case, and matching the non-empty list `(x:xs)` for the recursive case. Witness:

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

This occurs so often that Haskell provides a standard higher-order function that captures the essence of what goes on in this kind of definition:

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

Here is what happens if you call `foldr` with a function f, and identity element z, and a list $[x_1, x_2, x_3, \ldots, x_n]$:

$$\text{foldr } f\ z\ [x_1, x_2, ..., x_n] = (f\ x_1\ (f\ x_2\ (f\ x_3\ \ldots\ (f\ x_n\ z)\ldots).$$

And the same thing using infix notation:

$$\text{foldr}\; f\; z\; [x_1, x_2, ..., x_n] = (x_1 \text{ `}f\text{` } (x_2 \text{ `}f\text{` } (x_3 \text{ `}f\text{` } (\ldots (x_n \text{ `}f\text{` } z) \ldots).$$

For instance, the `and` function can be defined using `foldr` as follows:

```
and = foldr (&&) True
```

Exercise 1.2

1. *Define* `length` *in terms of* `foldr`.
2. *Define* `elem x` *in terms of* `foldr`.
3. *Find out what* `or` *does, and next define your own version of* `or` *in terms of* `foldr`.
4. *Define* `map f` *in terms of* `foldr`.
5. *Define* `filter p` *in terms of* `foldr`.
6. *Define* `(++)` *in terms of* `foldr`.
7. *Define* `reversal` *in terms of* `foldr`.

While `foldr` folds to the right, the following built-in function folds to the left. Here is the official definition:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z0 xs0 = lgo z0 xs0
            where
               lgo z []     =  z
               lgo z (x:xs) = lgo (f z x) xs
```

Here is a home-made version:

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

If you apply `foldl` to a function $f :: \alpha \to \beta \to \alpha$, a left identity element $z :: \alpha$ for the function, and a list of arguments of type β, then we get:

$$\text{foldl } f\ z\ [x_1, x_2, ..., x_n] = (f \ldots (f(f(f\ z\ x_1)\ x_2)\ x_3) \ldots x_n)$$

Or, if you write f as an infix operator:

$$\text{foldl } f\ z\ [x_1, x_2, ..., x_n] = (\ldots (((z \text{ `}f\text{` } x_1) \text{ `}f\text{` } x_2) \text{ `}f\text{` } x_3)\ \ldots\ \text{`}f\text{` } x_n)$$

Exercise 1.3 *Give an alternative definition of reverse, in terms of* `foldl`.

The `foldl` function has a strict cousin `foldl'`, with the following definition:

```
foldl' f a []     = a
foldl' f a (x:xs) = let a' = f a x in a' `seq` foldl' f a' xs
```

The `let` construction defines a local binding, so that you can think of `let x = f in e` as another way of expressing the application `(\ x -> e)  f.`.

To understand the definition of `foldl'`, one has to know that `seq` is a function that first evaluates its first argument to weak head normal form, and next returns the second argument.

An expression is in *weak head normal form* if it is (i) a constructor (such as True, or Just) with or without arguments, (ii) a built-in function with at least one of its arguments lacking, or (iii) a lambda abstraction.

Example of (i): `Just ((\ x -> x^2) 15).`

Example of (ii): `(+) 2.`

Example of (iii): `\ x -> x^2.`

So in the definition of `foldl'`, first `f a x` is evaluated, the result is called `a'`, and next the function calls itself recursively, with `foldl' f a' xs`.

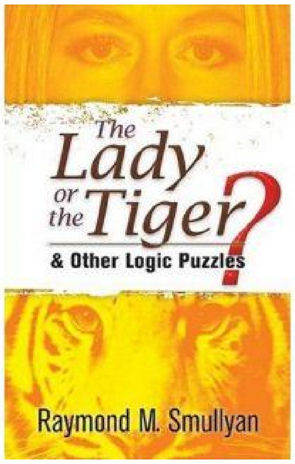
To understand `seq`, compare the following:

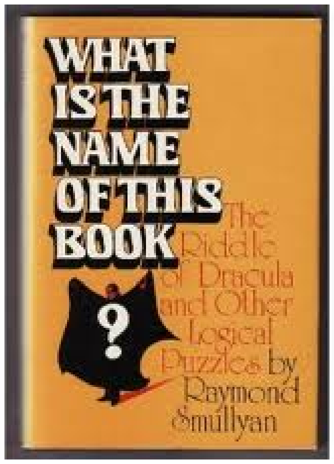
```
Prelude> :t seq
seq :: a -> b -> b
Prelude> seq undefined True
*** Exception: Prelude.undefined
Prelude> seq (\ x -> undefined) True
True
```

Explanation: the weak head normal form of `(\ x -> undefined)` is `(\ x -> undefined)`, so `seq` does not evaluate this further, but passes on its second argument.

1.6 Solving Logic Puzzles with Haskell

We can use Haskell to solve logical puzzles such as the famous Lady or Tiger puzzles by Raymond Smullyan [8].




WHAT IS THE NAME OF THIS BOOK
The Riddle of Dracula and Other Logical Puzzles by Raymond Smullyan

Here is the first puzzle. There are two rooms, and a prisoner has to choose between them. Each room contains either a lady or a tiger. In the first test the prisoner has to choose between a door with the sign "In this room there is a lady, and in the other room there is a tiger", and a second door with the sign "In one of these rooms there is a lady and in the other room there is a tiger." A final given is that one of the two signs tells the truth and the other does not. Here is a Haskell implementation that states the puzzle:

```
data Creature = Lady | Tiger deriving (Eq,Show)

sign1, sign2 :: (Creature,Creature) -> Bool
sign1 (x,y) = x == Lady && y == Tiger
sign2 (x,y) = x /= y
```

And here is the Haskell solution:

```
solution1 :: [(Creature,Creature)]
solution1 = [ (x,y) | x <- [Lady,Tiger],
                      y <- [Lady,Tiger],
                      sign1 (x,y) /= sign2 (x,y) ]
```

Running this reveals that the first room has a tiger in it, and the second room a lady:

```
*WLH> solution1
[(Tiger,Lady)]
```

Exercise 1.4 *The second puzzle of the book runs as follows. Again there are two signs. The sign on the first door says: "At least one of these rooms contains a lady." The sign on the second door says: "A tiger is in the other room." This time either the statements are both true or both false. Give a Haskell implementation of* `solution2` *that solves the puzzle. You will also have to write functions for the new signs, of course.*

It is part of the specification of a well-designed Lady and Tiger puzzle that the constraints should be such that there is a *single solution*. Trying to meet this specification is the recipe for becoming a logic puzzle designer.

Exercise 1.5 *Without looking at the Smullyan book, design some Lady and Tiger puzzles of your own, and use Haskell to test whether your puzzles are well-designed.*

For this, it is useful to introduce a *solve* procedure:

```
solve p = [ (x,y) | x <- [Lady,Tiger],
                    y <- [Lady,Tiger],
                    p (x,y)             ]
```

Exercise 1.6 *What is the type of* solve*?*

Here is another example puzzle. It is given that either both signs assert something that is true, or both signs assert falsehoods. The first sign says: "In both of these rooms there are ladies." The second sign says: "In this room there is a lady, in the other room there is a tiger." Is this a well-designed puzzle? No, it is not:

```
sign1' (this,other) = this == Lady && other == Lady
sign2' (other,this) = other == Tiger && this == Lady

solution' = solve (\ (x,y) -> sign1' (x,y) == sign2' (x,y))
```

This gives: `solution' == [(Lady,Tiger),(Tiger,Tiger)]`, a result which indicates that we have to add a constraint. The King says to the prisoner: "If you make the right choice, it is sure that you will not get eaten." Does this give us a well-designed puzzle? Yes, it does:

```
solution'' =  solve (\ (x,y) -> sign1' (x,y) == sign2' (x,y)
                             && not ((x,y) == (Tiger,Tiger)))
```

This gives `solution'' == [(Lady,Tiger)]`. It is part of the specification of this kind of logic puzzle that the givens constrain the space of possible solutions to a single item. A logic puzzle meets its specification if the puzzle has precisely one solution.

On the island of knights and knaves made famous in another logic puzzle book by Raymond Smullyan [9], there are two kinds of people. Knights always tell the truth, and knaves always lie. Of course, if you ask inhabitants of the island whether they are knights, they will always say "yes."

Suppose John and Bill are residents of the island. They are standing next to each other, with John left and Bill right. John says: "We are both knaves." Who is what? Here is a Haskell solution:

```
data Islander = Knight | Knave deriving (Eq,Show)

john :: (Islander,Islander) -> Bool
john (x,y) = (x,y) == (Knave,Knave)

solution3 :: [(Islander,Islander)]
solution3 = [(x,y) | x <- [Knight,Knave],
                     y <- [Knight,Knave],
                     john (x,y) == (x == Knight) ]
```

This reveals that John is a knave and Bill a knight:

```
*WLH> solution3
[(Knave,Knight)]
```

Exercise 1.7 *In this puzzle, again John is on the left, Bill on the right. John says: "We are both of the same kind." Bill says: "We are both of different kinds." Who is what? Implement a Haskell solution.*

Exercise 1.8 *Use the Haskell puzzle solving tools to design a few puzzles of your own. Method: work backward from puzzle answers specifications that have a unique solution to suitable puzzle forms. You can start to build some routine by designing variations on the ladies and tigers theme.*

Crime Scene Investigation A group of five school children is caught in a crime. One of them has stolen something from some kid they all dislike. The headmistress has to find out who did it. She questions the children, and this is what they say:

Matthew Carl didn't do it, and neither did I.

Peter It was Matthew or it was Jack.

Jack Matthew and Peter are both lying.

Arnold Matthew or Peter is speaking the truth, but not both.

Carl What Arnold says is not true.

Their class teacher now comes in. She says: three of these boys always tell the truth, and two always lie. You can assume that what the class teacher says is true. Use Haskell to write a function that computes who was the thief, and a function that computes which boys made honest declarations.

Hint: represent the declarations of the boys as properties for specifying the guilty boy, as follows:

```
data Boy = Matthew | Peter | Jack | Arnold | Carl
           deriving (Eq,Show)

boys = [Matthew, Peter, Jack, Arnold, Carl]

matthew, peter, jack, arnold, carl :: Boy -> Bool
matthew = \ x -> not (x==Matthew) && not (x==Carl)
peter   = \ x -> x==Matthew || x==Jack
jack    = \ x -> not (matthew x) && not (peter x)
arnold  = \ x -> matthew x /= peter x
carl    = \ x -> not (arnold x)

declarations = [matthew,peter,jack,arnold,carl]
table = zip declarations boys
```

Exercise 1.9 *Write a function* `guilty` *that lists the boys that could have done it, and a function* `honest` *that lists the boys that have made honest declarations, for each member of the solution list.*

A Puzzling Program Use a minute or so to analyze the following program.

```
main = putStrLn (s ++ show s)
  where s = "main = putStrLn (s ++ show s) \n  where s = "
```

This has the following ingredients that may still be unfamiliar to you:

- `show` for displaying an item as a string (if the item to be displayed is already a string, then this string is quoted);
- `\n` for the newline character.

Now that this was explained to you, reflect again, and tackle the exercises below.

Exercise 1.10 *Predict what will happen when the function* `main` *is executed. Next write down your prediction, and check it by executing the function.*

Exercise 1.11 *(Only for those who know some logic.) What does this have to do with logic? Hint: think of Kurt Gödel's famous proof of the incompleteness of the first order theory of arithmetic.*

1.7 Summary

If this was your first acquaintance with Haskell, make sure to actually *play around* and *do* some exercises. You will find that you will be up and running in no time.

If you already have some Haskell experience, this first chapter should have been plain sailing for you.

In the rest of this course we will focus on a particular aspect of functional programming: the use of executable specifications for programs.

Bibliography

[1] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley Longman, Boston, MA, 1990.

[2] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. Of International Conference on Functional Programming (ICFP)*, ACM SIGPLAN, 2000.

[3] Hal Daume. Yet another Haskell tutorial. `www.cs.utah.edu/~hal/docs/daume02yaht.pdf`.

[4] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing*. College Publications, London, 2004.

[5] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, 2007.

[6] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[7] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly, 2009.

[8] Raymond M. Smullyan. *The Lady or the Tiger? and Other Logic Puzzles*. Dover, 2009. First edition: 1982.

[9] Raymond M. Smullyan. *What is the name of this book?* Dover, first edition 1990 edition, 2011.