Week 1: Getting Started with Haskell

Jan van Eijck

CWI

September 5, 2011

A short history of Haskell

A short history of Haskell

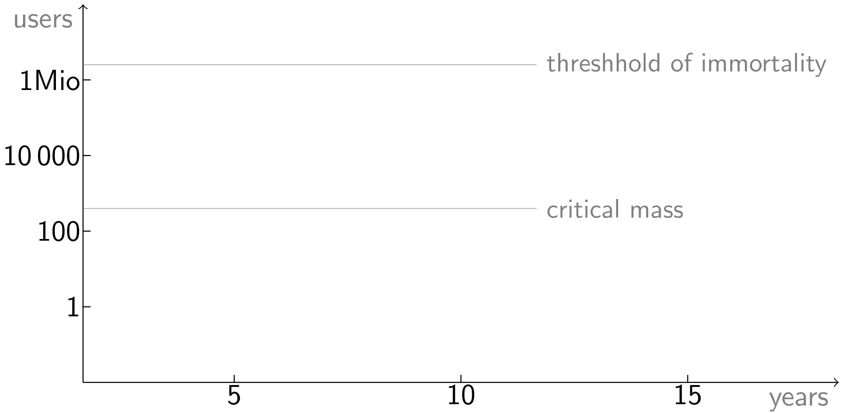
In the 80s, efforts of researchers working on functional programming were scattered across many languages (Lisp, SASL, Miranda, ML,. . .).
In 1987 a dozen functional programmers decided to meet in order to reduce unnecessary diversity in functional programming languages by **designing a common language** that is

- based on ideas that enjoy a wide consensus
- suitable for further language research as well as applications, including building large systems
- freely available

A short history of Haskell

In 1990, they published the first **Haskell** specification, named after the logician and mathematician Haskell B. Curry (1900-1982).

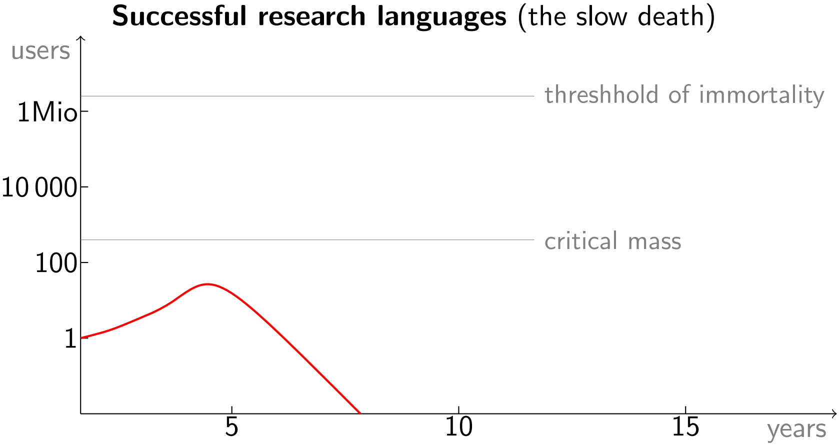
Simon Peyton-Jones: The life cycle of programming languages



Simon Peyton-Jones: The life cycle of programming languages

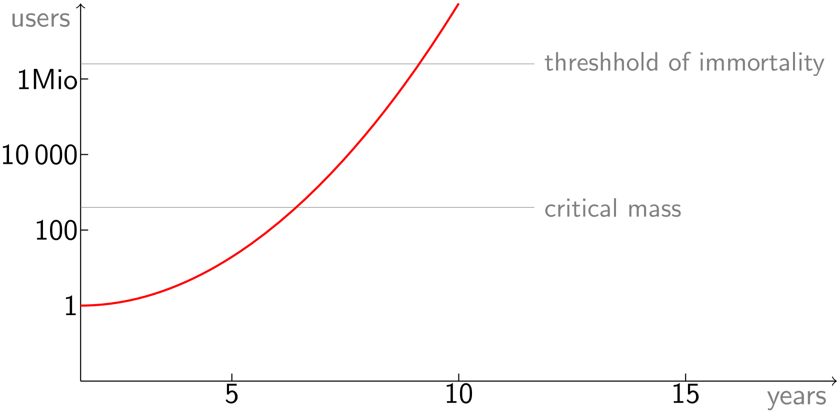
Most research languages (the quick death)

Simon Peyton-Jones: The life cycle of programming languages

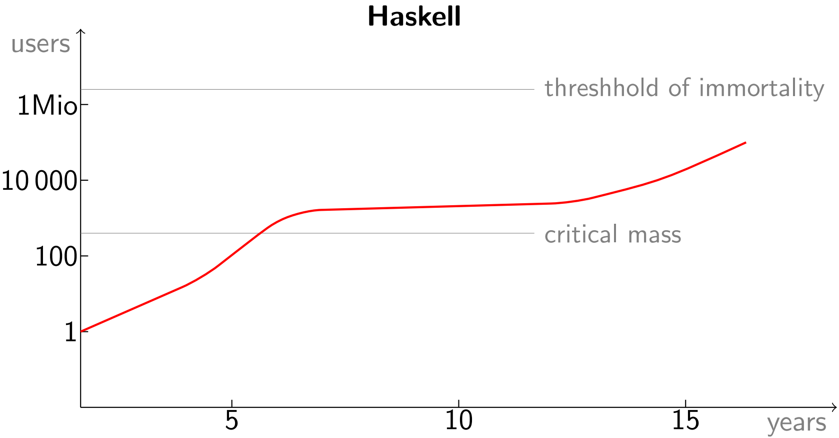


Simon Peyton-Jones: The life cycle of programming languages

C++, Java, Perl (the absence of death)



Simon Peyton-Jones: The life cycle of programming languages



Haskell is functional

A program consists entirely of functions.

- 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.

Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

A shift in thinking

Imperative thinking:

- Variables are pointers to storage locations whose value can be updated all the time.
- You give a sequence of commands telling the computer what to do step by step.

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`

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

- `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

A shift in thinking

Functional thinking:

- Variables are identifiers for an immutable, persistent value.
- You tell the computer what things are.

Examples:

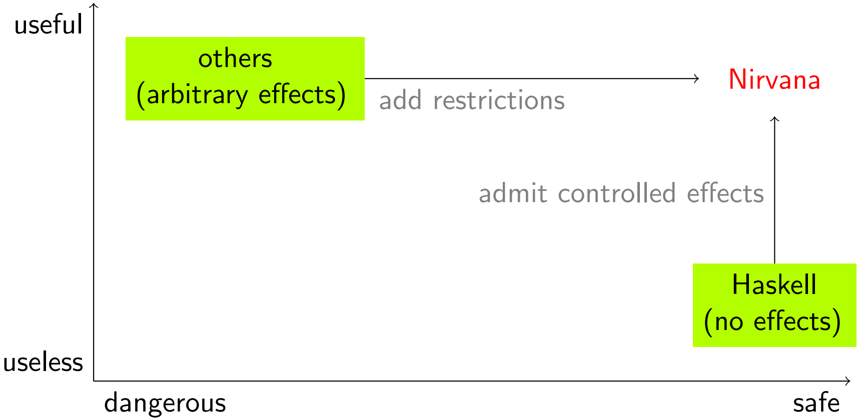
- `examplelist` is a list of integers containing the elements 1, 2, and 3
- the factorial of n is the product of all integers from 1 to n

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

A shift in thinking

Stop thinking in variable assignments, sequences and loops.
Start thinking in functions, immutable values and recursion.

Haskell is pure: Safety vs power



Why use Haskell?

Why use Haskell?

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

Why use Haskell?

- 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.

Why use Haskell?

- 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.

Why use Haskell?

- 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.

Why use Haskell?

- 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.

Why use Haskell?

- 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 study is QuickCheck.

Resources

- **For everything Haskell-related:** `haskell.org`.
- **Tutorials:**
  - Chapter 1 of “The Haskell Road”
  - Real World Haskell
    `book.realworldhaskell.org/read/`
  - Learn you a Haskell for great good
    `learnyouahaskell.com`
  - A gentle introduction to Haskell
    `haskell.org/tutorial`

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.

Haskell as a Calculator

Start the interpreter:

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Haskell as a Calculator

Start the interpreter:

```
lucht:cmpsem jve$ ghci
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

GHCi can be used to interactively evaluate expressions.

```
Prelude> 2 + 3
Prelude> 2 + 3 * 4
Prelude> 2^10
Prelude> (42 - 10) / 2
```

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
```

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.

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)`.

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.

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 Examples

```
module Week1

where

import Data.List
import Data.Char
```

Sentences can go on . . .

Sentences can go on

Sentences can go on . . .

Sentences can go on and on

Sentences can go on . . .

Sentences can go on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on and on

Sentences can go on . . .

Sentences can go on and on and on and on and on and on and on

Sentences can go on ...

Sentences can go on and on and on and on and on and on and on

```
gen :: Int -> String
gen 0 = "Sentences can go on"
gen n = gen (n-1) ++ " and on"

genS :: Int -> String
genS n = gen n ++ "."
```

A lazy list

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

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

Lambda Abstraction in Haskell

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

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

Lambda Abstraction in Haskell

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

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

- The intention is that variabele `x` stands proxy for a number of type `Int`.

Lambda Abstraction in Haskell

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

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

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.

Lambda Abstraction in Haskell

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

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

- 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`.

Lambda Abstraction in Haskell

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

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

- 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"
```

String Functions in Haskell

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

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

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]
```

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"
```

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

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.

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 `[]`.

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`.

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).

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).

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

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.

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)
```

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.

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`.

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]`.

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

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.

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.

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`.

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,

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,

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

CHOMSKY
GNIRUT

List Reversal

CHOMSKY YKSMOHC
GNIRUT

List Reversal

CHOMSKY YKSMOHC
GNIRUT TURING

List Reversal

CHOMSKY YKSMOHC
GNIRUT TURING

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

List Reversal

CHOMSKY YKSMOHC
GNIRUT TURING

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

Reversal works for any list, not just for strings.

Haskell Basic Types

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `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.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `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.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded.
- `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

Haskell Derived Types

- 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.

Haskell Derived Types

- 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 . . .

Haskell Derived Types

- 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`.

Haskell Derived Types

- 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 Hugs command `:t` to find the types of the function `map`, you get the following:

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

Mapping

If you use the Hugs command `:t` to find the types of the 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.

Mapping

If you use the Hugs command `:t` to find the types of the 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.
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

Sections

- In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its righthand side argument.

Sections

- 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.

Sections

- 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.

Sections

- 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]`

Sections

- 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.

Map

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>
```

Map

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>
```

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

Map

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>
```

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

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

Filter

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

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]
```

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]
```

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

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]
```

```
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
```

Examples

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

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

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

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

`nub` removes duplicates, as follows:

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

Function Composition

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.

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: $f \cdot g$.

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".

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)
```

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
```

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
```

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.

elem, all, and

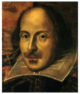
```
elem :: Eq a => a -> [a] -> Bool
elem x []      = False
elem x (y:ys) = x == y || elem x ys

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
```

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
```

Counting

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

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

Some Commands to Try Out

Some Commands to Try Out

- `putStrLn sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`

Some Commands to Try Out

- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (‘elem‘ "aeiou") sonnet73`

Some Commands to Try Out

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

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)`

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)`

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))`

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”.