Functional Programming

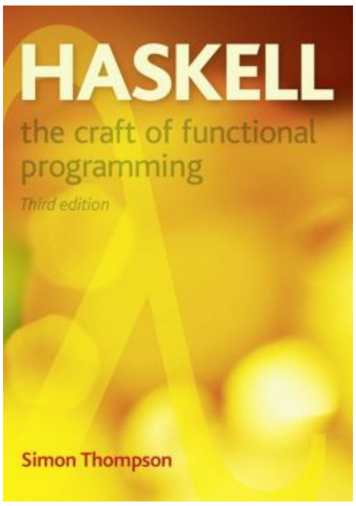
Arnold Meijster

Grading

- There are three lab assignments: $L = \frac{L_1+L_2+L_3}{3}$.
  Grading of the labs via Themis:
  `https://themis.housing.rug.nl`
- There is a final written exam: E
- Final grade: $F = \frac{4\times E+3\times L}{7}$ provided that $E \geq 5$ and $L \geq 5$
  Otherwise $F = \min(L, E)$
- In case of a resit exam, the resit grade replaces the grade E.
  Warning: there is no resit for the labs!

Literature

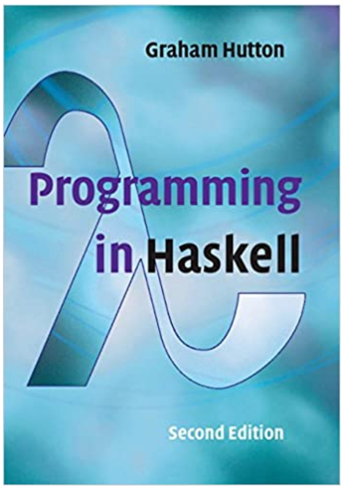
According to Ocasys, the literature for this course is the book
Haskell, the Craft of Functional Programming, 3rd ed.



If you have the 2^{nd} edition, that is fine.
Tutorial exercises are from the book. You are not advised to work through the book from page 1 to the last page. A much more gentle introduction to Haskell is to follow the slides.

Recommended Literature

The following book is highly recommended:



1. Functional Programming: The Idea

Functions are pure/mathematical mappings:
Always same output for same input

Computation = Application of functions to arguments

Example 1

In Haskell:

```
sum [1..10]
```

In C/C++/Java:

```
total = 0;
for (i = 1; i <= 10; ++i) {
  total = total + i;
}
```

Example 2

In Haskell:

```
wellknown [] = []
wellknown (x:xs) = wellknown leq ++ [x] ++ wellknown gt
   where leq = [y | y <- xs, y <= x]
         gt  = [z | z <- xs, z > x]
```


Quicksort in C

```
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

void quicksort(int low, int high, int *numbers) {
   int i = low, j = high, pivot = numbers[low];
   while (i <= j) {
      while (numbers[i] < pivot) i++;
      while (numbers[j] > pivot) j--;
      if (i <= j) {
        swap(&numbers[i], &numbers[j]);
        i++; j--;
      }
   }
   if (low < j) quicksort(low, j, numbers);
   if (i < high) quicksort(i, high), numbers);
}

void sort(int len, int *numbers) {
   quicksort(0, len - 1, numbers);
}
```

Characteristics of functional programs

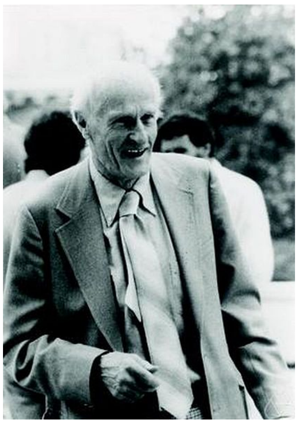
elegant	
expressive	
concise	
readable	
predictable	pure functions, no side effects
provable	it's (very basic) discrete mathematics!

Aims of functional programming

- Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures
- Minimize time to write programs:
  $\Rightarrow$ reduced development and maintenance time and costs
- Increased confidence in correctness of programs:
  clean and simple syntax and semantics
  $\Rightarrow$ programs are easier to
  - understand
  - test (Quickcheck!)
  - prove correct

Historic Milestones

1930s

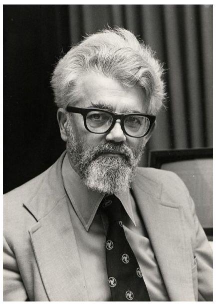
 

Alonzo Church and Stephen Kleene developed the lambda calculus, the core of all functional programming languages.

```
<Exp> ::=  <ident>
        |  <constant>
        |  ( <Exp> )
        |  lambda <ident> . <Exp>       -- function abstraction
        |  <Exp> <Exp>                  -- function application
```


Historic Milestones

1950s

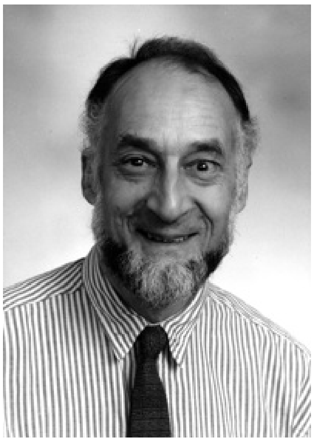


John McCarthy (Turing Award 1971) develops Lisp, the first functional programming language.

```
(defun factorial (n)
  (if (< n 2)
      1
      (* n (factorial (- n 1)))))
```

Historic Milestones

1970s



Robin Milner (FRS, Turing Award 1991) & Co. develop ML, the first modern (but impure) functional programming language with *polymorphic types* and *type inference*.

```
fun fac (0 : int) : int = 1
  | fac (n : int) : int = n * fac (n - 1)
```


Historic Milestones

1987

Haskell
A Purely Functional Language

Initial design by Philip Wadler, implementation by Simon Peyton Jones. Later, an international committee of researchers initiated the development of Haskell, a pure lazy functional language.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

fact :: Int -> Int
fact n = if n <= 0 then 1 else n * fact (n-1)

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

Why we teach FP

- FP is a fundamental programming style (like structured procedural programming and OO)
- FP is everywhere: Javascript, Scala, Erlang, (O)Caml, F# ...
- Pure functional Programs have no side-effects: they are easily mapped to symmetric multiprocessing architectures (SMPs)
- FP concepts make you a better programmer, no matter which language you use
- To show you that programming is a science, and need not be a black art with magic spells like `public static void`

2. Basic Haskell

Notational conventions
Type `Bool`
Type `Integer`
Guarded equations
Recursion
Some syntax matters
Types `Char` and `String`
Tuple types
Do's and Don'ts

GHC: Glasgow Haskell Compiler system. Documentation via link
http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html

2.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:	Mathematics	Haskell
	$f : A \times B \to C$	`f :: A -> B -> C`

Function application:	Mathematics	Haskell
	$f(a)$	`f a`
	$f(a,b)$	`f a b`
	$f(g(b))$	`f (g b)`
	$f(a,g(b))$	`f a (g b)`

Prefix operations bind stronger than infix operations:

`f a + b` means `(f a) + b`
not `f (a + b)`

2.2 Type Bool

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
xor x y  =  (x || y) && not(x && y)

xor1 :: Bool -> Bool -> Bool
xor1 x y  =  x /= y

xor2 :: Bool -> Bool -> Bool
xor2 True  True   =  False
xor2 True  False  =  True
xor2 False True   =  True
xor2 False False  =  False
```

The function xor2 is an example of the use of pattern matching.
The equations are tried in order. More later.

Is `xor x y == xor2 x y` true?

Testing with QuickCheck

QuickCheck: library for software testing.
Warning: on many linux distributions, you need to install it separately!

Import test framework:

```
import Test.QuickCheck
```

Define property (assertion) to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Check property:

```
> quickCheck prop_xor2
+++ OK, passed 100 tests.
```


`XorDemo.hs`

For GHCi commands (like `:l`, `:q`, `:t`, etc) see

http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html

2.3 Type `Integer`

Unlimited precision integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

```
Prelude> 2^200
1606938044258990275541962092341162602522202993782792835301376
```

==, <=, etc are overloaded and work on many types!

There is also the type `Int` with a platform dependent precision, which is guaranteed for the interval $[-2^{29} .. 2^{29})$.

On my 64 bit Linux installation I found:

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

Warning: be aware of precision!

```
Prelude> (2::Int)^62
4611686018427387904
Prelude> (2::Int)^62 + ((2::Int)^62 - 1)
9223372036854775807
Prelude> (2::Int)^63
-9223372036854775808
Prelude> (2::Int)^64
0
Prelude> 2^64
18446744073709551616
Prelude> 2^100
1267650600228229401496703205376
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
          = (3 * 3) * (3 * 3)
          = 81
```

Evaluation of Haskell expressions
means
applying the defining equations from left to right.

2.4 Guarded equations

Example: maximum of 2 integers.

```
maxval :: Integer -> Integer -> Integer
maxval x y
  | x >= y      = x
  | otherwise   = y
```

Haskell also has an `if-then-else`:

```
maxval x y  =  if x >= y then x else y
```

Let us (quick) check that `maxval` is associative:

```
prop_max_assoc x y z =
  maxval x (maxval y z) == maxval (maxval x y) z
> quickCheck prop_max_assoc

+++ OK, passed 100 tests.
```

Local definitions: where

A defining equation can be followed by one or more local definitions.

```
pow4 x  =  x2 * x2 where x2 = x * x

pow4 x  =  sq (sq x) where sq x = x * x

pow8 x  =  sq (sq x2)
  where x2 = x * x
        sq x = x * x
```


Local definitions: let

`let` x `=` e_1 `in` e_2

defines x locally in expression e_2

Example:

```
let x = 2+3 in x^2 + 2*x
= 35
```

Like e_2 `where` x `=` e_1
But can occur anywhere in an expression
`where`: only after function definitions

2.5 Recursion

Example: x^n (using only $*$, not ^)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0   = 1                -- the base case
  | n > 0    = x * pow x (n-1)  -- the recursive case
```

More compactly (using pattern matching):

```
pow x 0  =  1
pow x n | n > 0  =  x * pow x (n-1)
```


Evaluating pow

```
pow x 0  =  1
pow x n | n > 0  =  x * pow x (n-1)
```

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

> pow 2 (-1)

GHCi answers

```
*** Exception: PowDemo.hs:(1,1)-(2,33):
    Non-exhaustive patterns in function pow
```

Partially defined functions

```
pow x n | n > 0  =  x * pow x (n-1)
```

versus

```
pow2 x n  =  x * pow2 x (n-1)
```

- `pow`: call outside intended domain raises exception
- `pow2`: call outside intended domain leads to arbitrary behaviour, including nontermination

You are strongly advised to write functions in the style of the `pow` function.

Example `sumTo`

The sum from 0 to `n` = `0 + 1 + ...  + (n-1) + n`

```
sumTo :: Integer -> Integer
sumTo 0  =  0
sumTo n | n > 0  =
```

```
prop_sumTo n =
  sumTo n == n*(n+1) `div` 2
```

```
> quickCheck prop_sumTo

*** Exception: SumDemo.hs:(4,1)-(5,35): Non-exhaustive
patterns in function sumTo
```

Recursion in general

- Reduce a problem to a *smaller* problem, e.g. `pow x n` to `pow x (n-1)`
- Must eventually reach a *base case*
- Build up solutions from smaller solutions

General problem solving strategy
in *any* programing language

The only way of 'looping' in Haskell!

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                              -- base case
f n | n > 0  =  ... f(n - 1) ...       -- recursive call(s)
```

Always make the base case as simple as possible (typically 0)

Many variations:

- more parameters
- other base cases, e.g. `f 1`
- other recursive calls, e.g. `f(n - 2)`
- more than one recursive call (e.g. fibonacci)

Fibonacci: $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Clearly, due to the exponential time complexity of this implementation, it is unfeasible to compute `fib 100`.

However, it is easy to make a linear time implementation:

```
fib :: Integer -> Integer
fib n = f n 0 1
   where
    f 0 a b = a
    f n a b = f (n-1) b (a+b)
```

Evaluation of `fib 5 = f 5 0 1`:

`f 5 0 1 = f 4 1 1 = f 3 1 2 = f 2 2 3 = f 1 3 5 = f 0 5 8 = 5`.

2.6 Some syntax matters

Functions are defined by one or more equations.
In the simplest case, each function is defined
by an (possibly conditional) equation:

$$\begin{array}{l} f\ x_1\ \ldots\ x_n \\ \quad \mid\ test_1\ \ =\ \ e_1 \\ \quad \vdots \\ \quad \mid\ test_n\ \ =\ \ e_n \end{array}$$

Each right-hand side e_i is an expression.

Note: `otherwise = True`

Function and parameter names must begin with a lower-case letter
(Type names begin with an upper-case letter)

An *expression* can be

- a *literal* like `0` or `"xyz"`,
- or an *identifier* like `True` or `x`,
- or a *function application* $f\ e_1\ \ldots\ e_n$ where f is a function and $e_1\ \ldots\ e_n$ are expressions,
- or a parenthesized expression (e)

Note the similarity with λ-calculus.

Additional syntactic sugar:

- `if then else`
- infix
- `where`
- ...

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

> f 3

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

> f 3

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
```

Binding occurrence
Bound occurrence
Scope of binding

Scoping by example

Summary:

- Order of definitions is irrelevant
- Parameters and `where`-defs are local to each equation

Layout: the offside rule

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
 a = 10
b = 20
 c = 30
```

In a sequence of definitions,
each definition must begin in the same column.

```
a = 10 +
  20
```

```
a = 10 +
20
```

```
 a = 10 +
20
```

A definition ends with the first piece of text
in or to the left of the start column.

Prefix and infix

Function application: `f a b`

Functions can be turned into infix operators
by enclosing them in back quotes.

Example

```
5 `mod` 3  =  mod 5 3
```

Infix operators: `a + b`

Infix operators can be turned into functions
by enclosing them in parentheses.

Example

```
(+) 1 2  =  1 + 2
```


Comments

Until the end of the line: `--`

```
id x = x    -- the identity function
```

A comment block: `{- ... -}`

```
{- Comments
   are
   important
-}
```

Function composition

Consider the following two (mathematical) functions:

$$f(x) = x^2 \qquad g(x) = f(f(x))$$

Here, $g(x)$ is defined in terms of $f(x)$ by *function composition*.

In Haskell you can do this similarly:

```
f, g :: Int -> Int
f x = x*x
g x = f (f x)
```

However, it is Haskell-stylish to use the composition (.) operator:

```
f, g :: Int -> Int
f x = x*x
g = f.f
```


2.7 Types `Char` and `String`

Character literals as usual: `'a'`, `'$'`, `'\n'`, ...
Lots of predefined functions in module `Data.Char`.

String literals as usual: `"I am a string"`

Strings are lists of characters.
Lists can be concatenated with ++:
`"I am" ++ "a string"` = `"I ama string"`

More on lists later.

2.8 Tuple types

```
(True, 'a', "abc") :: (Bool, Char, String)
```

In general:

$$\begin{array}{ll} \text{If} & e_1 \;\texttt{::}\; T_1 \quad \ldots \quad e_n \;\texttt{::}\; T_n \\ \text{then} & \texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \;\texttt{::}\; \texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)} \end{array}$$

In mathematical notation: $T_1 \times \cdots \times T_n$

For pairs, the functions `fst` and `snd` are predefined in the Prelude:

```
fst (a,b) = a
snd (a,b) = b
```

Here is an example of another linear time Fibonacci function:

```
fib :: Integer -> Integer
fib n = fst (f n)
  where
    f 0 = (0,1)
    f n = let (a,b)=f(n-1) in (b,a+b)
```


2.9 Do's and Don'ts

True and False

Never write

$$b == \texttt{True}$$

Simply write

$$b$$

Never write

$$b == \texttt{False}$$

Simply write

$$not\ b$$


```
isBig :: Integer -> Bool
```

~~isBig n~~
~~| n > 9999 = True~~
~~| otherwise = False~~

```
isBig n  =  n > 9999
```

~~if b then True else False~~ b

~~if b then False else True~~ not b

~~if b then True else b'~~ b || b'

...

Tuple

Try to avoid (mostly):

```
f (x,y)  =  ...
```

Usually better:

```
f x y  =  ...
```

Just fine:

```
f x y  =  (x + y, x - y)
```


3. Lists

List comprehension
Generic functions: Polymorphism
Pattern matching on lists
Pattern matching
Recursion over lists
Case study: Pictures

Lists are the most important data type in functional programming


```
[1, 2, 3, -42] :: [Integer]

[False] :: [Bool]

['C', 'h', 'a', 'r'] :: [Char]
```
=
```
"Char" :: String
```
because
```
type String = [Char]

[not, not] ::

[] :: [a]        -- empty list for any type a

[[True],[]] ::
```

Typing rule

If e_1 `::` T ... e_n `::` T
then `[`e_1`,...,`e_n`]` `::` `[`T`]`

Graphical notation:

$$\frac{e_1 :: T \quad \dots \quad e_n :: T}{[e_1, \dots, e_n] :: [T]}$$

`[True, 'c']` is not type-correct!!!

All elements in a list must have the same type

Test

```
(True, 'c') ::

[(True, 'c'), (False, 'd')] ::

([True, False], ['c', 'd']) ::
```

List ranges

```
[1 .. 3] = [1, 2, 3]
[1,3 .. 10] = [1, 3, 5, 7, 9]
[3 .. 1] = []
['a' .. 'c'] = ['a', 'b', 'c']
```


Concatenation: ++

Concatenates two lists of the same type:

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

~~[1, 2] ++ ['a']~~

3.1 List comprehension

Set comprehensions (mathematics):

$$\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\}$$

The set of all x^2 *such that* x *is an element of* $\{1, 2, 3, 4, 5\}$

List comprehension:

```
[ x^2 | x <- [1 .. 5]]
```

The list of all `x^2` *such that* `x` *is an element of* `[1 .. 5]`

List comprehension — Generators

```
[ x^2 | x <- [1 .. 5]]
= [1, 4, 9, 16, 25]

[ toLower c | c <- "Hello, World!"]
= "hello, world!"

[ (x, even x) | x <- [1 .. 3]]
= [(1, False), (2, True), (3, False)]

[ x+y | (x,y) <- [(1,2), (3,4), (5,6)]]
= [3, 7, 11]
```

pattern <- *list expression*
is called a *generator*

List comprehension — Tests

```
[ x*x | x <- [1 .. 5], odd x]
= [1, 9, 25]

[ x*x | x <- [1 .. 5], odd x, x > 3]
= [25]

[ toLower c | c <- "Hello, World!", isAlpha c]
= "helloworld"
```

Boolean expressions are called *tests*

Defining functions by list comprehension

```
factors :: Int -> [Int]
factors n  =   [d | d <- [1 .. n], n `mod` d == 0]
```

$\Longrightarrow$ `factors 15 = [1, 3, 5, 15]`

```
prime :: Int -> Bool
prime n  =   [1,n] == factors n
```

$\Longrightarrow$ `prime 15 = False`

```
primes :: Int -> [Int]
primes n  =   [p | p <- [1 .. n], prime p]
```

$\Longrightarrow$ `primes 100 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]`

factors: a bit more efficient

Instead of

```
factors :: Int -> [Int]
factors d  =  [d | d <- [1 .. n], n `mod` d == 0]
```

it is more efficient to write

```
factors :: Int -> [Int]
factors n =  [d | d <- [1 .. n `div` 2], n `mod` d == 0] ++ [n]
```


List comprehension — General form

$$[\ \mathit{expr}\ \mid\ E_1,\ \ldots,\ E_n\ ]$$

where *expr* is an expression and each E_i is a generator or a test

Multiple generators

```
[(i,j) | i <- [1 .. 2], j <- [7 .. 9]]

= [(1,7), (1,8), (1,9), (2,7), (2,8), (2,9)]
```

Analogy: each generator is a `for` loop:

```
for all i <- [1 .. 2]
  for all j <- [7 .. 9]
    ...
```

Key difference:

Loops *do* something
Expressions *produce* something

Dependent generators

```
[(i,j) | i <- [1 .. 3], j <- [i .. 3]]
= [(1,j) | j <- [1..3]] ++
  [(2,j) | j <- [2..3]] ++
  [(3,j) | j <- [3..3]]
= [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

The meaning of list comprehensions

$[e \mid x \texttt{ <- } [a_1,\ldots,a_n]]$
$= (\texttt{let } x = a_1 \texttt{ in } [e]) \texttt{ ++ } \cdots \texttt{ ++ } (\texttt{let } x = a_n \texttt{ in } [e])$

$[e \mid b]$
$= \texttt{if } b \texttt{ then } [e] \texttt{ else } []$

$[e \mid x \texttt{ <- } [a_1,\ldots,a_n],\ \overline{E}]$
$= (\texttt{let } x = a_1 \texttt{ in } [e \mid \overline{E}]) \texttt{ ++ } \cdots \texttt{ ++ }$
$\quad (\texttt{let } x = a_n \texttt{ in } [e \mid \overline{E}])$

$[e \mid b,\ \overline{E}]$
$= \texttt{if } b \texttt{ then } [e \mid \overline{E}] \texttt{ else } []$

Convention

Identifiers of list type end in ‘s’:

`xs, ys, zs, ...`

Example: concat

```
concat xss  = [x | xs <- xss, x <- xs]

concat [[1,2], [4,5,6]]
= [x | xs <- [[1,2], [4,5,6]], x <- xs]
= [x | x <- [1,2]] ++ [x | x <- [4,5,6]]
= [1,2] ++ [4,5,6]
= [1,2,4,5,6]
```

What is the type of concat?

`[[a]] -> [a]`

3.2 Generic functions: Polymorphism

Polymorphism = one function can have many types

Example

```
length :: [Bool] -> Int
length :: [Char] -> Int
length :: [[Int]] -> Int
⋮
```

The most general type:

`length :: [a] -> Int`

where `a` is a *type variable*

$\Longrightarrow$ `length :: [a] -> Int` *for all types* a

Type variable syntax

Type variables must start with a lower-case letter

Typically: a, b, c, ...

Defining polymorphic functions

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

fst :: (a,b) -> a
fst (x,y)  =  x

swap :: (a,b) -> (b,a)
swap (x,y)  =  (y,x)

silly :: Bool -> a -> Char
silly x y  =  if x then 'c' else 'd'

silly2 :: Bool -> Bool -> Bool
silly2 x y  =  if x then x else y
```

Polymorphic list functions from the Prelude

```
length :: [a] -> Int
length [5, 1, 9] = 3
```

```
(++) ::  [a] -> [a] -> [a]
[1, 2] ++ [3, 4] = [1, 2, 3, 4]
```

```
reverse :: [a] -> [a]
reverse [1, 2, 3] = [3, 2, 1]
```

```
replicate :: Int -> a -> [a]
replicate 3 'c' = "ccc"
```


Polymorphic list functions from the Prelude

```
head "list" = 'l'
last "list" = 't'
tail "list" = "ist"
init "list" = "lis"
```

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

```
last :: [a] -> a
```

```
tail :: [a] -> [a]
```

```
init :: [a] -> [a]
```

Polymorphic list functions from the Prelude

```
take 3 "list" = "lis"
drop 3 "list" = "t"

take :: Int -> [a] -> [a]

drop :: Int -> [a] -> [a]


-- A property:
prop_take_drop xs  =
  take n xs ++ drop n xs == xs
```


Polymorphic list functions from the Prelude

```
concat :: [[a]] -> [a]
concat [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0]

zip :: [a] -> [b] -> [(a,b)]
zip [1,2] "ab" = [(1, 'a'), (2, 'b')]

BEWARE: zip [1..3] [1..4] = [(1,1), (2,2), (3,3)]

unzip :: [(a,b)] -> ([a],[b])
unzip [(1, 'a'), (2, 'b')] = ([1,2], "ab")
```

Warning: in general we have `unzip(zip xs ys) /= (xs, ys)`

Further list functions from the Prelude

```
and :: [Bool] -> Bool
and [True, False, True] = False

or :: [Bool] -> Bool
or [True, False, True] = True

-- For numeric types a:
sum, product :: [a] -> a
sum [1, 2, 2] = 5
product [1, 2, 2] = 4
```


Polymorphism versus Overloading

Polymorphism: one definition, many types

Overloading: different definitions for different types

Example

Function `(+)` is overloaded:

- on type `Int`: built into the hardware
- on type `Integer`: realized in software

So what is the type of `(+)` ?

Numeric types

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

Function `(+)` has type `a -> a -> a` for any type of class `Num`

- Class `Num` is the class of *numeric types*.
- Don't confuse with OO classes!
- Predefined numeric types: `Int`, `Integer`, `Float`, `Double`
- Types of class `Num` offer the basic arithmetic operations:

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
⋮
sum, product :: Num a => [a] -> a
```


Other important type classes

- The class `Eq` of *equality types*, i.e. types that have

```
(==) ::  Eq a => a -> a -> Bool
(/=) ::  Eq a => a -> a -> Bool
```

  Most types are of class `Eq`.
  Exception:

- The class `Ord` of *ordered types*, i.e. types that have

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

More on type classes later.

Warning: QuickCheck and polymorphism

QuickCheck does not work on polymorphic properties!

Example

QuickCheck does not find a counterexample to

```
prop_reverse :: [a] -> Bool
prop_reverse xs  =  reverse xs == xs
```

The solution: specialize the polymorphic property, e.g.

```
prop_reverse :: [Int] -> Bool
prop_reverse xs  =  reverse xs == xs
```

Now QuickCheck works

3.3 Pattern matching on lists

3.4 Pattern matching

Every list can be constructed from `[]`
by repeatedly adding an element at the front
with the "cons" operator `(:) :: a -> [a] -> [a]`

syntactic sugar	in reality
`[3]`	`3 : []`
`[2, 3]`	`2 : 3 : []`
`[1, 2, 3]`	`1 : 2 : 3 : []`
$[x_1, \ldots, x_n]$	$x_1 : \ldots : x_n : []$

Note: $x : y : zs = x : (y : zs)$
`(:)` associates to the right

$\Longrightarrow$
Every list is either

`[]`

$x : xs$ where
x is the *head* (first element), and
xs is the *tail* (rest list)

`[]` and `(:)` are called *constructors*
because every list can be constructed uniquely from them.

$\Longrightarrow$
Every non-empty list can be decomposed uniquely into head and tail.

Therefore these definitions make sense:

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

`(++)` is not a constructor:
`[1,2,3]` is not uniquely constructable with `(++)`:
`[1,2,3] = [1] ++ [2,3] = [1,2] ++ [3]`

Therefore this definition does not make sense:

```
nonsense (xs ++ ys) = length xs - length ys
```


Patterns

Patterns are expressions
consisting only of constructors and variables.
No variable must occur twice in a pattern.

$\Longrightarrow$ Patterns allow unique decomposition = *pattern matching*.

A *pattern* can be

- a variable such as `x` or a wildcard `_` (underscore)
- a literal like `1`, `'a'`, `"xyz"`, ...
- a tuple `(`p_1`, ...,` p_n`)` where each p_i is a pattern
- a constructor pattern C p_1 ... p_n
  where C is a constructor and each p_i is a pattern

Pattern matching / wildcards

Example

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

tail :: [a] -> [a]
tail (_ : xs)  =  xs

null :: [a] -> Bool
null []        =  True
null (_ : _)  =  False
```


Function definitions by pattern matching

Example

```
true12 :: [Bool] -> Bool
true12 (True : True : _)  =  True
true12 _  =  False

same12 :: Eq a => [a] -> [a] -> Bool
same12 (x : _) (_ : y : _)  =  x == y
same12 _ _ = False

asc3 :: Ord a => [a] -> Bool
asc3 (x : y : z : _)  =  x <= y && y <= z
asc3 _  =  False
```

3.5 Recursion over lists

Example

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

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

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


Primitive recursion on lists:

```
f []        =  base    -- base case
f (x : xs)  =  rec     -- recursive case
```

- *base*: (terminating) expression without a call of f
- *rec*: expression using call(s) $f\ xs$

$\Longrightarrow$ f always terminates!

f may have additional parameters.

Primitive recursive definitions

Example

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

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


Mutual recursion

Example

```
even :: Int -> Bool
even n = n == 0 || n > 0 && odd (n-1) || n < 0 && odd (n+1)

odd :: Int -> Bool
odd n = n /= 0 && (n > 0 && even (n-1) || n < 0 && even (n+1))
```

Insertion sort

Example

```
insertionSort :: Ord a => [a] -> [a]
insertionSort [] = []
insertionSort (x:xs) = insert x (insertionSort xs)

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


Beyond primitive recursion: Multiple arguments

Example

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys)  =  (x,y) : zip xs ys
zip _ _   =  []
```

Maybe you are tempted to write:

```
zip' [] []  =  []
zip' (x:xs) (y:ys)  =  (x,y) : zip' xs ys
```

`zip'` is undefined for lists having different lengths!

Beyond primitive recursion: Multiple arguments

Example

```
take :: Int -> [a] -> [a]
take 0 _     = []
take _ []    = []
take i (x:xs) | i>0  =  x : take (i-1) xs
```


General recursion: Quicksort

Example

```
quickSort :: Ord a => [a] -> [a]
quickSort []  =  []
quickSort (x:xs)  =
  quickSort below ++ [x] ++ quickSort above
    where
      below = [y | y <- xs, y <= x]
      above = [y | y <- xs, x < y]
```

A better Quicksort

Example

```
quickSort :: Ord a => [a] -> [a]
quickSort []  =  []
quickSort (x:xs)  =
  quickSort below ++ [x] ++ quickSort above
    where
      (below,above) = partition x xs
      partition :: Ord a => a -> [a] -> ([a],[a])
      partition n []  = ([],[])
      partition n (x:xs) | x <= n    = (x:leq, gt)
                         | otherwise = (leq, x:gt)
                           where (leq,gt) = partition n xs
```


Accumulating parameter

Idea: Result is accumulated in a parameter and returned later

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

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


reverse' :: [a] -> [a]
reverse' xs = rev xs []
  where
    rev [] rs     = rs
    rev (x:xs) rs = rev xs (x:rs)
```

Accumulating parameter

Example: list of all maximal ascending sublists in a list

```
ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]
```

```
ups ::  Ord a => [a] -> [[a]]
ups xs =  ups2 xs []

ups2 ::  Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial descending sublist (note reversal!)
ups2 (x:xs) []  =  ups2 xs [x]
ups2 []     ys  =  [reverse ys]
ups2 (x:xs) (y:ys)
  | x >= y      =  ups2 xs (x:y:ys)
  | otherwise   =  reverse (y:ys) : ups2 (x:xs) []
```


Accumulating parameter (more elegant)

```
ups ::  Ord a => [a] -> [[a]]
ups xs =  ups2 xs []
  where
    ups2 ::  Ord a => [a] -> [a] -> [[a]]
    ups2 (x:xs) []  =  ups2 xs [x]
    ups2 []     ys  =  [reverse ys]
    ups2 (x:xs) (y:ys)
      | x >= y      =  ups2 xs (x:y:ys)
      | otherwise   =  reverse (y:ys) : ups2 (x:xs) []
```

3.6 Case study: Pictures

```
type Picture = [String]
```

```
uarr :: Picture
uarr =
 ["  #  ",
  " ### ",
  "#####",
  "  #  ",
  "  #  "]
```

```
larr :: Picture
larr =
 ["  #  ",
  " ##  ",
  "#####",
  " ##  ",
  "  #  "]
```



```
flipH :: Picture -> Picture
flipH  =  reverse

flipV :: Picture -> Picture
flipV pic  =  [ reverse line | line <- pic]

rarr :: Picture
rarr  =  flipV larr

darr :: Picture
darr  =  flipH uarr

above :: Picture -> Picture -> Picture
above  =  (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 = [ l1 ++ l2 | (l1,l2) <- zip pic1 pic2]
```

Chessboards

```
bSq  =  replicate 5 (replicate 5 '#')

wSq  =  replicate 5 (replicate 5 ' ')

alterH :: Picture -> Picture -> Int -> Picture
alterH pic1 pic2 1  =  pic1
alterH pic1 pic2 n  =  pic1 `beside` alterH pic2 pic1 (n-1)

alterV :: Picture -> Picture -> Int -> Picture
alterV pic1 pic2 1  =  pic1
alterV pic1 pic2 n  =  pic1 `above` alterV pic2 pic1 (n-1)

chessboard :: Int -> Picture
chessboard n  =  alterV bw wb n where
  bw = alterH bSq wSq n
  wb = alterH wSq bSq n
```


4. Higher-Order Functions

Applying functions to all elements of a list: `map`
Filtering a list: `filter`
Combining the elements of a list: `foldr`
Lambda expressions
Curried functions
More library functions
Case study: Counting words

Recall [`Pic` is short for `Picture`]

```
alterH :: Pic -> Pic -> Int -> Pic
alterH pic1 pic2 1 = pic1
alterH pic1 pic2 n = beside pic1 (alterH pic2 pic1 (n-1))

alterV :: Pic -> Pic -> Int -> Pic
alterV pic1 pic2 1 = pic1
alterV pic1 pic2 n = above pic1 (alterV pic2 pic1 (n-1))
```

Very similar. Can we avoid duplication?

```
alt :: (Pic -> Pic -> Pic) -> Pic -> Pic -> Int -> Pic
alt f pic1 pic2 1 = pic1
alt f pic1 pic2 n = f pic1 (alt f pic2 pic1 (n-1))

alterH pic1 pic2 n = alt beside pic1 pic2 n

alterV pic1 pic2 n = alt above pic1 pic2 n
```


Higher-order functions:
Functions that take functions as arguments

`... -> (... -> ...) -> ...`

Higher-order functions capture patterns of computation

4.1 Applying functions to all elements of a list: `map`

Example

```
map even [1, 2, 3]
= [False, True, False]

map toLower "R2-D2"
= "r2-d2"

map reverse ["abc", "123"]
= ["cba", "321"]
```

What is the type of `map`?

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


`map`: The mother of all higher-order functions

Predefined in `Prelude`. Two possible definitions:

- Using a list comprehension:

  ```
  map f xs  =  [ f x | x <- xs ]
  ```

- using recursion:

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

Evaluating map

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

```
map sqr [1, -2]
= map sqr (1 : -2 : [])
= sqr 1 : map sqr (-2 : [])
= sqr 1 : sqr (-2) : (map sqr [])
= sqr 1 : sqr (-2) : []
= 1 : 4 : []
= [1, 4]
```


Some properties of map

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f ys

map f (reverse xs) = reverse (map f xs)
```

Proofs by induction (another lecture)

QuickCheck and higher order functions

QuickCheck does not work automatically
for properties using function variables.

It needs to know how to generate functions.

Cheap alternative: replace function variable by specific function(s)

Example

```
prop_map_even :: [Int] -> [Int] -> Bool
prop_map_even xs ys =
  map even (xs ++ ys) = map even xs ++ map even ys
```


4.2 Filtering a list: `filter`

Example

```
filter even [1, 2, 3]
= [2]

filter isAlpha "R2-D2"
= "RD"

filter null [[], [1,2], []]
= [[], []]
```

What is the type of `filter`?

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

`filter`

Predefined in `Prelude`. Two possible definitions:

- Using a list comprehension:

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

- using recursion:

```
filter p []                  =  []
filter p (x:xs) | p x        =  x : filter p xs
                | otherwise  =  filter p xs
```


Some properties of `filter`

```
filter p (xs ++ ys) = filter p xs ++ filter p ys
```

```
filter p (reverse xs) = reverse (filter p xs)
```

Proofs by induction

4.3 Combining the elements of a list: `foldr`

Example

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

$$\texttt{sum [}x_1\texttt{, ..., }x_n\texttt{] = }x_1\texttt{ + ... + }x_n\texttt{ + 0}$$

```
concat []         =  []
concat (xs:xss)   =  xs ++ concat xss
```

$$\texttt{concat [}xs_1\texttt{, ..., }xs_n\texttt{] = }xs_1\texttt{ ++ ... ++ }xs_n\texttt{ ++ []}$$

foldr

$$\texttt{foldr } (\oplus)\ z\ \texttt{[}x_1\texttt{, ..., }x_n\texttt{] = } x_1 \oplus (x_2 \oplus (\ldots \ \oplus (x_n \oplus z) \ldots))$$

Applications:

```
sum xs      =   foldr (+) 0 xs
product xs  =   foldr (*) 1 xs
concat xss  =   foldr (++) [] xss
```

Defined in `Prelude`. What is the type of `foldr`?

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

foldr

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

`foldr f a` replaces `(:)` by `` `f` `` and `[]` by `a`

Evaluating foldr

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

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + (-2 + (foldr (+) 0 []))
= 1 + (-2 + 0)
= -1
```

More applications of foldr

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

```
  product xs   =  foldr  (*)   1      xs
  and xs       =  foldr  (&&)  True   xs
  or xs        =  foldr  (||)  False  xs
  inSort xs    =  foldr  ins   []     xs
```

```
ins x []   =  [x]
ins x (y:ys) = if x <= y then x : y : ys else y : ins x ys

foldr ins [] [1, -2]
= foldr ins [] (1 : -2 : [])
= 1 `ins` (foldr ins [] (-2 : []))
= 1 `ins` (-2 `ins` (foldr `ins` [] []))
= 1 `ins` (-2 `ins` [])
= 1 `ins` [-2]
= -2:(1 `ins` [])
= -2:[1]
= [-2,1]
```


Quiz

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

What is

`foldr (:) ys xs`

Example: `foldr (:) ys (1:2:3:[]) = 1:2:3:ys`

`foldr (:) ys xs = ???`

Defining functions via `foldr`

- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

For example, if `f` is associative and `` z `f` x = x `` then
`` foldr f z (xs++ys) = foldr f z xs `f` foldr f z ys ``.

Therefore `sum (xs++ys) = sum xs + sum ys`,
`product (xs++ys) = product xs * product ys`, ...

4.4 Lambda expressions

Consider

```
squares xs  =  map sqr xs  where  sqr x = x * x
```

Do we really need to define `sqr` explicitly? No!

```
\x -> x * x
```

is the anonymous function with

formal parameter `x` and result `x * x`

In mathematics: $x \mapsto x * x$

Evaluation:

```
(\x -> x * x) 3 = 3 * 3 = 9
```

Usage:

```
squares xs  =  map (\x -> x * x) xs
```

Terminology

$$\texttt{(\textbackslash}x\ \texttt{->}\ e_1\texttt{)}\ e_2$$

x: formal parameter
e_1: result
e_2: actual parameter

Why "lambda"?

The logician Alonzo Church invented *lambda calculus* in the 1930s

Logicians write $\lambda x.\ e$ instead of `\x ->` e

Typing lambda expressions

Example

```
(\x -> x > 0) :: (Num a, Ord a) => a -> Bool
```

The general rule:

$$\texttt{(\textbackslash}x\ \texttt{->}\ e\texttt{)}\ \texttt{::}\ T_1\ \texttt{->}\ T_2$$
$$\text{if } x\ \texttt{::}\ T_1 \text{ implies } e\ \texttt{::}\ T_2$$

infix operators and lambda expressions

```
(+ 1)   means   (\x -> x + 1)
(1 +)   means   (\x -> 1 + x)
(* 2)   means   (\x -> x * 2)
(2 *)   means   (\x -> 2 * x)
(2 ^)   means   (\x -> 2 ^ x)
(^ 2)   means   (\x -> x ^ 2)
```

etc

Example

```
squares xs  =  map (^ 2) xs
```


4.5 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int      f :: Int -> (Int -> Int)
f x y  =  x+y               f x  =  \y -> x+y
```

Both mean the same:

```
f a b                       (f a) b
= a + b                     = (\y -> a + y) b
                            = a + b
```

The trick: any function of two arguments
can be looked upon as a function of the first argument
that returns a function of the second argument

Consequence of Currying

Every function is a function of one argument
(which may return a function as a result)

$$T_1 \texttt{ -> } T_2 \texttt{ -> } T$$

is just syntactic sugar for

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T)$$

$$f \; e_1 \; e_2$$

is just syntactic sugar for

$$\underbrace{(f \; e_1)}_{::\, T_2 \texttt{ -> } T} \; e_2$$

Analogously for more arguments

`->` is not associative:

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T) \neq (T_1 \texttt{ -> } T_2) \texttt{ -> } T$$

Example

```
f :: Int -> (Int -> Int)        g :: (Int -> Int) -> Int
f x y  =  x + y                 g h  =  h 0 + 1
```

Application is not associative:

$$(f \; e_1) \; e_2 \neq f \; (e_1 \; e_2)$$

Example

`(f 3) 4` $\neq$ `f (3 4)`   `g (id abs)` $\neq$ `(g id) abs`

Quiz

```
head tail xs
```

Correct?

Partial function application

Every function of n parameters
can be applied to less than n arguments

Example

Instead of `sum xs = foldr (+) 0 xs`

just define `sum = foldr (+) 0`

In general:

If f `::` T_1 `->` `...` `->` T_n `->` T

and a_1 `::` T_1`,` `...,` a_m `::` T_m and $m \leq n$

then $f\ a_1\ \ldots\ a_m$ `::` T_{m+1} `->` `...` `->` T_n `->` T

4.6 More library functions

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

Example

```
head2  =  head . tail

head2 [1,2,3]
= (head . tail) [1,2,3]
= (\x -> head (tail x)) [1,2,3]
= head (tail [1,2,3])
= head [2,3]
= 2
```



```
all :: (a -> Bool) -> [a] -> Bool
all p xs  =  and [p x | x <- xs]
```

Example

```
all (>1) [0, 1, 2]
= False
```

```
any :: (a -> Bool) -> [a] -> Bool
any p  =  or [p x | x <- xs]
```

Example

```
any (>1) [0, 1, 2]
= True
```

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

Example

```
takeWhile (not . isSpace) "the end"
= "the"
```

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

Example

```
dropWhile (not . isSpace) "the end"
= " end"
```


curry / uncurry

```
curry f  =  \ x y -> f(x,y)

uncurry f  =  \(x,y) -> f x y
```

Example:

```
f (x,y) = x+y
add = curry f      -- now add 5 6 yields 11
```

What are the types of curry and uncurry?

```
curry :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```

4.7 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

Function `putStr` yields

```
again: 1
never: 2
say: 1
```

Design principle:

Solve problem in a sequence of small steps
transforming the input gradually into the output

Similar to Unix pipes!

Step 1: Break input into words

$$\begin{array}{c}\texttt{"never say never again"} \\ \text{function} \big\downarrow \texttt{words} \\ \texttt{["never", "say", "never", "again"]}\end{array}$$

Predefined in Prelude

Step 2: Sort words

$$\texttt{["never", "say", "never", "again"]} \xrightarrow{\text{function } \texttt{sort}} \texttt{["again", "never", "never", "say"]}$$

```
["never", "say", "never", "again"]
        function | sort
                 ↓
["again", "never", "never", "say"]
```

Predefined in `Data.List`

Step 3: Group equal words together

```
["again", "never", "never", "say"]
        function | group
                 ↓
[["again"], ["never", "never"], ["say"]]
```

Predefined in `Data.List`

Step 4: Count each group

$$\begin{array}{c} \texttt{[["again"], ["never", "never"], ["say"]]} \\ \Big\downarrow \; \texttt{map (\textbackslash ws -> (head ws, length ws))} \\ \texttt{[("again", 1), ("never", 2), ("say", 1)]} \end{array}$$

Step 5: Format each group

$$\begin{array}{c} \texttt{[("again", 1), ("never", 2), ("say", 1)]} \\ \Big\downarrow \; \texttt{map (\textbackslash (w,n) -> (w ++ ": " ++ show n)} \\ \texttt{["again: 1", "never: 2", "say: 1"]} \end{array}$$

Step 6: Combine the lines

```
["again: 1", "never: 2", "say: 1"]
```

$$\Big\downarrow \text{function } \texttt{unlines}$$

```
"again: 1\nnever: 2\nsay: 1\n"
```

Predefined in Prelude

The solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```

Merging maps

Can we merge two consecutive maps?

```
map f . map g = ???
```


The optimized solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\ws -> head ws ++ ": " ++ show(length ws))
  . group
  . sort
  . words
```

5. Lazy evaluation

Applications of lazy evaluation

Infinite lists

Introduction

So far, we did not pay much attention to the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation

Advantages:

- Avoids unnecessary evaluations
- Terminates as often as possible
- Supports infinite lists
- Increases modularity

Therefore Haskell is called a *lazy functional language*.

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

One evaluation:

```
sq(3+4)  =  sq 7  =  7 * 7  =  49
```

Another evaluation:

```
sq(3+4)  =  (3+4) * (3+4)  =  7 * (3+4)  =  7 * 7  =  49
```


Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example

Two evaluations (in C, C++, or Java), where `n` is initially `0`:

```
n + (n = 1)  =  0 + (n = 1)  =  0 + 1  =  1
n + (n = 1)  =  n + 1  =  1 + 1  =  2
```

Reduction strategies

An expression may have many reducible subexpressions:

`sq (3+4)`

Terminology: *redex* = reducible expression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body
`sq (3+4)  =  sq 7  =  7 * 7`

Outermost reduction Always reduce an outermost redex.
Corresponds to *call by name:*
The unevaluated arguments
are substituted into the the function body
`sq (3+4)  =  (3+4) * (3+4)`

Comparison: Number of steps

Innermost reduction:

`sq (3+4) =  sq 7  =  7 * 7  =  49`

Outermost reduction:

`sq(3+4)  =  (3+4)*(3+4)  =  7*(3+4)  =  7*7  =  49`

More outermost than innermost steps!
How can outermost reduction be improved?
Sharing!

```
sq(3+4)  =  •  *  •  =  •  *  •  =  49
             ↘   ↙        ↘   ↙
              3+4           7
```

The expression 3+4 is only evaluated *once!*

Lazy evaluation := outermost reduction + sharing

Lazy evaluation never needs more steps than innermost reduction.

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)   =  fst(1,tail loop)
               =  fst(1,tail(tail loop))
               =  ...
```

Outermost reduction:

```
fst (1,loop)   = 1
```

If there exists a terminating reduction, then outermost reduction terminates

The principles of lazy evaluation:

- Arguments of functions are evaluated only
  if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully,
  but only far enough to evaluate the function.
  (Remember `fst (1,loop)`)
- Each argument is evaluated at most once (sharing!)

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0          -- f.1
f (x:xs) []     =  0          -- f.2
f (x:xs) (y:ys) =  x+y        -- f.3
```

Lazy evaluation:

```
f [1..10^1000] [7..10^1000]
-- does f.1 match?
=  f (1 :[2..10^1000]) [7..10^1000]
-- does f.2 match?
=  f (1 :[2..10^1000]) (7:[8..10^1000])
-- does f.3 match?
=  1+7
=  8
```

Guards

Example

```
max3 m n p
  | m >= n && m >= p  =  m
  | n >= p            =  n
  | otherwise         =  p
```

Lazy evaluation:

```
max3 (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
  ?  5 >= 4-1 && 5 >= 3+9
  ?  5 >= 3 && 5 >= 3+9
  ?  True    && 5 >= 3+9
  ?  5 >= 3+9
  ?  5 >= 12
  ?  False
  ?  3 >= 12
  ?  False
  ?  otherwise = True
=  12
```


Guards

Example

```
max3 m n p
  | m >= n && m >= p  =  m
  | n >= p            =  n
  | otherwise         =  p
```

Lazy evaluation:

```
max3 (2+3) (3+9) (4-1)
  ?  2+3 >= 3+9 && 2+3 >= 4-1
  ?  5 >= 3+9 && 5 >= 4-1
  ?  5 >= 12 && 5 >= 4-1
  ?  False    && 5 >= 4-1
  ?  12 >= 4-1
  ?  12 >= 3
  ?  True
=  12
```

Slogan

Lazy evaluation evaluates an expression only when needed
and only as much as needed.

(*"Call by need"*)

5.1 Applications of lazy evaluation

The minimum of a list

```
inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)

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

min ::  Ord a =>  [a] -> a
min =  head . inSort
```

```
⟹  inSort [6,1,7,5]
    =  ins 6 (ins 1 (ins 7 (ins 5 [])))
```



```
min [6,1,7,5]   =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 []))))
=  head(ins 6 (ins 1 (ins 7 (5 : []))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 []))
=  1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic.
The sorted list is never constructed completely

Warning: this depends on the exact algorithm and does not work so nicely with all sorting functions!

5.2 Infinite lists

Example

```
ones :: [Int]
ones  =  1 : ones
```

This defines an infinite list of 1s:

```
ones  =  1 : ones  =  1 : 1 : ones  =  ...
```

What GHCi has to say about it:

```
> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

Haskell lists can be finite or infinite

Printing an infinite list does not terminate

But Haskell can cope with infinite lists, thanks to lazy evaluation:

```
> head ones
1

> take 5 ones
[1,1,1,1,1]
```

Remember:

Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: `head ones = head (1 : ones) = 1`

Innermost reduction:

```
  head ones
= head (1 : ones)
= head (1 : 1 : ones)
= ...
```


Haskell lists are never actually infinite but only potentially infinite

Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:

1 : 1 : 1 : $\boxed{\text{code pointer to compute rest}}$

In general: finite prefix followed by code pointer

Why (potentially) infinite lists?

- They come for free with lazy evaluation
- They increase modularity:
  list producer does not need to know
  how much of the list the consumer wants

Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, ...
2. Output the first value p in the list as a prime.
3. Delete all multiples of p from the list
4. Goto step 2

   2 3 4 5 6 7 8 9 10 11 12 ...
   2 3 5 7 11 ...

In Haskell:

```
primes :: [Integer]
primes  =  sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (p:xs)  =  p : sieve [x | x <- xs, x `mod` p /= 0]
```

Lazy evaluation:

```
  primes
=  sieve [2..]
=  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x `mod` 2 /= 0]
=  2 : sieve (3 : [x | x <- [4..],  x `mod` 2 /= 0])
=  2 : 3 : sieve [x | x <- [x|x <- [4..],  x `mod` 2 /= 0],
                      x `mod` 3 /= 0]
= ...
```


Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
[101,103,107,109,113,127,131,137,139,149]
```

All twin primes (p and p+2 are primes, for example 41 and 43):

```
> [(p,q) | (p,q) <- zip primes (tail primes), p+2==q]
[(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73),(1
```

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
```
nontermination

```
isPrime n  =  n == head (dropWhile (<n) primes)
```


Infinite list of natural numbers

`nats = [0..]` is the infinite list `[0,1,2,3,4,5,....]`.

There are many other ways to construct this list:

```
nats = f 0
  where
    f n = n:f (n+1)
```

```
nats = 0:map (+1) nats
```

```
nats = 0:f (map *2) nats)
  where
    f (x:xs) = x:x+1:f xs
```

Infinite list of Fibonacci numbers

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

A naive implementation would be:

```
fibos :: [Integer]
fibos = map f [0..]
  where
    f 0 = 0
    f 1 = 1
    f n = f (n-1) + f (n-2)
```

```
*Main> :set +s
*Main> take 40 fibos
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,
10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,
1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,
63245986]
(371.67 secs, 191,960,417,736 bytes)
```


Infinite list of Fibonacci numbers

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

A much better implementation is:

```
fibs :: [Integer]
fibs = f 0 1
  where
    f m n = m : (f n (m+n))
```

```
*Main> take 40 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,
10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,
1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,
63245986]
(0.01 secs, 259,896 bytes)
```

Infinite list of Fibonacci numbers

```
fibs :: [Integer]
fibs = f 0 1
  where
    f m n = m : (f n (m+n))
```

The evaluation of `fibs` goes as follows:

```
  fibs
= f 0 1
= 0 : (f 1 (0+1))
= 0 : (f 1 1)
= 0 : 1 : (f 1 (1+1))
= 0 : 1 : (f 1 2)
= 0 : 1 : 1 : (f 2 (1+2))
= 0 : 1 : 1 : (f 2 3)
= 0 : 1 : 1 : 2 : (f 3 (2+3))
= 0 : 1 : 1 : 2 : (f 3 5)
= 0 : 1 : 1 : 2 : 3 : (f 5 (3+5))
```


Infinite list of Fibonacci numbers

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

The standard infinite list implementation is:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Underlying idea: assume you already have an infinite list of Fibonacci numbers:

```
[ 0, 1, 1, 2, 3, 5,  8, 13, .... ]
```

The tail of this list is

```
[ 1, 1, 2, 3, 5, 8, 13, 21, .... ]
```

`zipWith` combines the two lists element by element using the $(+)$ operator:

```
   [ 0, 1, 1, 2, 3,  5,  8, 13, .... ]
+  [ 1, 1, 2, 3, 5,  8, 13, 21, .... ]
=  [ 1, 2, 3, 5, 8, 13, 21, 34, .... ]
```

So the final list is obtained by prepending the zipped list with 0 and 1.

Approximating the golden ratio

In mathematics, two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities.
Formally, for $a > b > 0$:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi = \frac{1+\sqrt{5}}{2}$$

Note that this corresponds with the way we compute the Fibonacci series.
Therefore, the following property holds:

$$\frac{F_{n+1}}{F_n} \to \varphi \quad \text{for} \quad n \to \infty$$

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

gratio = zipWith (/) (tail fibs) (fibs)
```

Note that here `fibs` start at 1 (to avoid division by zero). In ghci the calculation of `take 10 gratio` yields
`[1.0,2.0,1.5,1.6666666666666667,1.6,1.625,1.6153846153846154,`
`1.619047619047619,1.6176470588235294,1.6181818181818182]`

A note about evaluation order

The operator `($)` is used to do function application at a different precedence to help avoid parentheses.

```
f x = x + 1

*Main> f (f (f (f (f (f 36)))))
42
*Main> f $ f $ f $ f $ f $ f 36
42
```

A note about evaluation order

Haskell uses lazy evaluation by default.

But you can override this using the so-called *strict function application* operator `($!)`.

`f $! x` behaves the same as `f x`, except that the evaluation of expression `x` is forced before the function `f` is applied.

Basically, it means that we use *call-by-value*!

A note about evaluation order

```
sumwith :: Int -> [Int] -> [Int]
sumwith v []     = v
sumwith v (x:xs) = sumwith (v+x) xs
```

Lazy evaluation of sumwith 0 [1,2,3]:

```
  sumwith 0 [1,2,3]
=
  sumwith (0+1) [2,3]
=
  sumwith ((0+1)+2) [3]
=
  sumwith (((0+1)+2)+3) []
=
  ((0+1)+2)+3
=
  (1+2)+3
=
  3+3
=
  6
```

A note about evaluation order

```
sumwith :: Int -> [Int] -> [Int]
sumwith v []     = v
sumwith v (x:xs) = (sumwith $! (v+x)) xs
```

Evaluation of sumwith 0 [1,2,3]:

```
  sumwith 0 [1,2,3]
=
  (sumwith ($! (0+1))) [2,3]
=
  (sumwith $! 1) [2,3]
=
  (sumwith $! (1+2)) [3]
=
  (sumwith $! 3) [3]
=
  (sumwith $! (3+3)) []
=
  (sumwith $! (6)) []
=
  6
```


6. Case Study: Recursive Descent Parsing

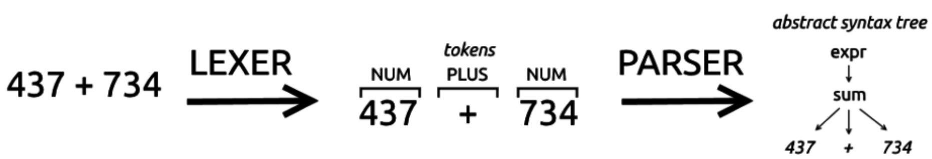
LL(1) grammar

Consider the following context free grammar:

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> <empty string>
F  -> <letters> | <digits>
```

This grammar is called an LL(1) grammar since it can be *parsed* from Left to right, use a Lookahead of 1 symbol.

Tokenizing the input




```
import Data.Char

lexer :: String -> [String]
lexer [] = []
lexer (c:cs)
 | elem c "\n\t " = lexer cs
 | elem c "*/+-"  = [c]:(lexer cs)
 | isAlpha c    = (c:takeWhile isAlpha cs): lexer(dropWhile isAlpha cs)
 | isDigit c    = (c:takeWhile isDigit cs): lexer(dropWhile isDigit cs)
 | otherwise    = error "Syntax Error: invalid character in input"
```

Main parser function

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> <empty string>
F  -> <letters> | <digits>
```

```
parser :: String -> (String,[String])
parser str = parseS "" (lexer str)
```


Parsing S

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> <empty string>
F  -> <letters> | <digits>
```

```
parseS :: String -> [String] -> (String,[String])
parseS accepted tokens = parseS' acc rest
  where (acc, rest) = parseF accepted tokens
```

Parsing S'

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> <empty string>
F  -> <letters> | <digits>
```

```
parseS' :: String -> [String] -> (String,[String])
parseS' accepted ("*":tokens) =  parseS' acc rest
  where (acc,rest) = parseF (accepted++"*") tokens
parseS' accepted ("/":tokens) = parseS' acc rest
  where (acc,rest) = parseF (accepted++"/") tokens
parseS' accepted tokens =  (accepted, tokens)
```


Parsing F

```
S  -> F S'
S' -> * F S'
S' -> / F S'
S' -> <empty string>
F  -> <letters> | <digits>
```

```
parseF :: String -> [String] -> (String, [String])
parseF accepted []       =  error "Parse error...abort"
parseF accepted (tok:tokens)
  | isAlpha (head tok)  =  (accepted++tok, tokens)
  | isDigit (head tok)  =  (accepted++tok, tokens)
  | otherwise           =  error ("Syntax Error: " ++ tok)
```

Demo ghci session

```
*Main> parser "a*b*c*d*ef"
("a*b*c*d*ef",[])
*Main> parser "a*b*c/d*ef**a"
*** Exception: Syntax Error: *
*Main> parser "a*b*c/d*ef*/a"
*** Exception: Syntax Error: /
*Main> parser "a*b*c/d*ef/a"
("a*b*c/d*ef/a",[])
*Main> parser "a*b*9/c"
("a*b*9/c",[])
*Main> parser "a*b*9/c9a*b"
("a*b*9/c",["9","a","*","b"])
```


7. Algebraic `data` Types

`data` by example

Case study: boolean formulas

So far: no real new types, just compositions of existing types

Examples:

```
type Pixel = (Int,Int)
type String = [Char]
type Picture = [String]
```

Now: `data` defines *new* types

Introduction by example: From enumerated types
to recursive and polymorphic types

7.1 `data` by example

Bool

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q  =  False
True  && q  =  q

(||) :: Bool -> Bool -> Bool
False || q  = q
True  || q  = True
```


deriving

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False

instance Show Bool where
  show True    =  "True"
  show False   =  "False"
```

Better: let Haskell write the code for you:

```
data Bool = False | True
            deriving (Eq, Show)
```

`deriving` supports many more classes: `Ord`, ...

Warning

Do not forget to make your `data` types instances of `Show`

Otherwise Haskell cannot print values of your type

Season

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Show)

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring
```

Shape

```haskell
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rectangle Width Height
             deriving Show
```

Some values of type Shape:

```haskell
Circle 1.0
Rectangle 0.9 1.1
Circle (-2.0)
```

```haskell
area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rectangle w h)  =  w * h


instance Eq Shape where
  Circle r1   == Circle r2            =  r1 == r2
  Rectangle w1 h1 == Rectangle w2 h2  =  w1 == w2 && h1 == h2
  _ == _                              =  False
```


Maybe

From the Prelude:

```haskell
data Maybe a = Nothing  | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```haskell
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =
lookup key ((x,y):xys)
  | key == x   =
  | otherwise  =
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n  =  n
add (Suc m) n  =

mul :: Nat -> Nat -> Nat
mul Zero n  =  Zero
mul (Suc m) n  =
```


Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

show is defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse ","  (map show xs)
```

Tree

```
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq, Show)
```

Some trees:

```
Empty
Node 1 Empty Empty
Node 1 (Node 2 Empty Empty) Empty
Node 1 Empty (Node 2 Empty Empty)
Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
⋮
```



```
find :: Ord a => a -> Tree a -> Bool
find _ Empty  =  False
find x (Node a l r)
  | x < a  =  find x l
  | a < x  =  find x r
  | otherwise  =  True
```

Another implementation, using short circuit evaluation would be:

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty  =  False
find x (Node a l r) = (x==a) || (x<a && find x l) || (x>a && find x r)
```

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty  = Node x Empty Empty
insert x (Node a l r)
  | x < a   =  Node a (insert x l) r
  | a < x   =  Node a l (insert x r)
  | otherwise  =  Node a l r
```

Example

```
insert 6 (Node 5 Empty (Node 7 Empty Empty))
= Node 5 Empty (insert 6 (Node 7 Empty Empty))
= Node 5 Empty (Node 7 (insert 6 Empty) Empty)
= Node 5 Empty (Node 7 (Node 6 Empty Empty) Empty)
```


Edit distance

Problem: how to get from one word to another,
with a *minimal* number of "edits".

Example: from `"fish"` to `"chips"`
`[Change 'c',Insert 'h',Copy,Change 'p',Change 's']`

Applications: DNA Analysis, Unix `diff` command

So, we want a function with the type:
`transform :: String -> String -> [Edit]`

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys    =  map Insert ys
transform xs []    =  replicate (length xs) Delete
transform (x:xs) (y:ys)
  | x == y         =  Copy : transform xs ys
  | otherwise      =  best [Change y : transform xs ys,
                            Delete   : transform xs (y:ys),
                            Insert y : transform (x:xs) ys]
```



```haskell
best :: [[Edit]] -> [Edit]
best [xs]    = xs
best (xs:xss)
  | cost xs <= cost b    = xs
  | otherwise            = b
  where b = best xss

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

Time complexity of `transform`: $O(\quad)$

The edit distance problem can be solved in time $O(mn)$
using *dynamic programming*

Patterns revisited

Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A *pattern* can be

- a literal like `1`, `'a'`, `"xyz"`, ...
- a variable
- a wildcard (i.e. `_`)
- a tuple (p_1, ..., p_n) where each p_i is a pattern
- a constructor pattern C p_1 ... p_n where
  C is a data constructor (incl. `True`, `False`, `[]` and `(:)`)
  and each p_i is a pattern

7.2 Case study: boolean formulas

```
type Name = String
data Form = F | T
          | Var Name
          | Not Form
          | And Form Form
          | Or Form Form
          deriving Eq
```

Example: `Or (Var "p") (Not(Var "p"))`
More readable: symbolic infix constructors, start with `:`

```
data Form = F | T
          | Var Name
          | Not Form
          | Form :&: Form
          | Form :|: Form
          deriving Eq
```

Now: `Var "p" :|: Not(Var "p")`

Pretty printing

```
par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
  show F = "F"
  show T = "T"
  show (Var x) = x
  show (Not p) = par("~" ++ show p)
  show (p :&: q) = par(show p ++ " & " ++ show q)
  show (p :|: q) = par(show p ++ " | " ++ show q)
```

```
> Var "p" :|: Not(Var "p")
(p | (~p))
```


Syntax versus meaning

`Form` is the *syntax* of boolean formulas, not their meaning:

`Not(Not T)` and `T` mean the same but are different:

`Not(Not T) /= T`

What is the meaning of a `Form`?

Its value!?

But what is the value of `Var "p"` ?

```
-- Evaluation
type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = sure(lookup x v)
      where sure (Just b) = b
eval v (Not p) = not(eval v p)
eval v (p :&: q) = eval v p && eval v q
eval v (p :|: q) = eval v p || eval v q
```

```
> eval [("a",False), ("b",False)]
   (Not(Var "a") :&: Not(Var "b"))
True
```


Valuations

All valuations for a given list of variable names:

```
vals :: [Name] -> [Valuation]
vals [] = [[]]
vals (x:xs) = [ (x,False):v  | v <- vals xs ] ++
              [ (x,True):v   | v <- vals xs ]

vals ["b"]
= vals "b":[]
= [("b",False):v | v <- vals []] ++
  [("b",True):v  | v <- vals []]
= [("b",False):[]] ++ [("b",True):[]]
= [[("b",False)]] ++ [[("b",True)]]
= [[("b",False)], [("b",True)]]

vals ["a","b"]
= [("a",False):v | v <- vals ["b"]] ++
  [("a",True):v  | v <- vals ["b"]]
= [[("a",False),("b",False)],[("a",False),("b",True)],
   [("a",True),("b",False)],[("a",True),("b",True)]]
```

Variables of a formula

```
vars :: Form -> [Name]
vars F = []
vars T = []
vars (Var x) = [x]
vars (Not p) = vars p
vars (p :&: q) = uniq (vars p ++ vars q)
vars (p :|: q) = uniq (vars p ++ vars q)

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


Satisfiable and tautology

```
satisfiable :: Form -> Bool
satisfiable p = or [eval v p | v <- vals(vars p)]

tautology :: Form -> Bool
tautology p = and [eval v p | v <- vals(vars p)]
```

Maybe you like better:

```
tautology :: Form -> Bool
tautology = not . satisfiable . Not
```

```
p0 :: Form
p0 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))
```

```
> p0
((a & b) | ((~a) & (~b)))

> vals (vars p0)
[[("a",False),("b",False)], [("a",False),("b",True)],
 [("a",True), ("b",False)], [("a",True), ("b",True )]]

> [ eval v p0 | v <- vals (vars p0) ]
[True, False, False, True]

> satisfiable p0
True

> tautology p0
False
```


Simplifying a formula: Not inside?

```
> isSimple (Var "a")
True
> isSimple (Not (Var "a"))
True
> isSimple (Not (Not (Var "a")))
False
> isSimple (Var "a" :|:  (Not (Var "b") :&: Not (Var "c")))
True
> isSimple (Not (Var "a" :|:  ((Var "b") :&: Not (Var "c"))))
False
```

Simplifying a formula: Not inside?

```
isSimple :: Form -> Bool
isSimple (p :&: q)  =  isSimple p && isSimple q
isSimple (p :|: q)  =  isSimple p && isSimple q
isSimple (Not p)    =  not (isOp p)
  where
  isOp (Not p)     =  True
  isOp (p :&: q)   =  True
  isOp (p :|: q)   =  True
  isOp p           =  False
isSimple _         =  True
```


Alternative: Not inside?

```
isSimple :: Form -> Bool
isSimple (p :&: q)  =  isSimple p && isSimple q
isSimple (p :|: q)  =  isSimple p && isSimple q
isSimple (Not p)    =  not (isOp p)
  where
  isOp F           = False
  isOp T           = False
  isOp (Var x)     = False
  isOp _           = True
isSimple _         =  True
```

NOT ALLOWED: Not inside?

```
isSimple :: Form -> Bool
isSimple (p _ q)  =  isSimple p && isSimple q
isSimple (Not p)  =  not (isOp p)
  where
  isOp F          = False
  isOp T          = False
  isOp (Var x)    = False
  isOp _          = True
isSimple _        =  True
```


Simplifying a formula: Not inside!

```
simplify :: Form -> Form
simplify (p :&: q)  =  simplify p :&: simplify q
simplify (p :|: q)  =  simplify p :|: simplify q
simplify (Not p)    =  pushNot (simplify p)
  where
  pushNot T          =  F
  pushNot F          =  T
  pushNot (Not p)    =  p
  pushNot (p :&: q)  =  pushNot p :|: pushNot q
  pushNot (p :|: q)  =  pushNot p :&: pushNot q
  pushNot p          =  Not p
simplify p           =  p
```

8. Modules and Abstract Data Types

Modules

Abstract Data Types

8.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Name space management: `M.f` vs `f`
- Information hiding

GHC: one module per file

Recommendation: module `M` in file `M.hs`

Module header

```
module M where    -- M must start with capital letter
↑
```

All definitions must start in this column

- Exports everything defined in `M`

Selective export:

```
module M (T, f, ...)  where
```

- Exports only `T, f, ...`

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only `T`, but not its constructors

```
module M (T(C,D,...)) where
data T = ...
```

- Exports `T` and its constructors `C`, `D`, ...

```
module M (T(..)) where
data T = ...
```

- Exports `T` and all of its constructors

Exporting modules

```
module B where
import A
...
```

```
module A where
f = ...
...
```

$\Longrightarrow$ `B` does not export `f`

By default, modules do not export names from imported modules

Unless the names are mentioned in the export list

```
module B (f) where
import A
...
```

Or the whole module is exported

```
module B (module A) where
import A
...
```


`import`

By default, everything that is exported is imported

```
module B where
import A
...
```

```
module A where
f = ...
g = ...
```

$\Longrightarrow$ `B` imports `f` and `g`

Unless an import list is specified

```
module B where
import A (f)
...
```

$\Longrightarrow$ `B` imports only `f`

Or specific names are hidden

```
module B where
import A hiding (g)
...
```

qualified

```
import A
import B
import C
... f ...
```

Where does `f` come from??

Clearer: *qualified names*

```
... A.f ...
```

Can be enforced:

```
import qualified A
```

$\Longrightarrow$ must always write `A.f`

Renaming modules

```
import qualified TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```

Painful

More readable:

```
import qualified TotallyAwesomeModule as TAM

... TAM.f ...
```

For the full description of the module system
see the Haskell report

8.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why?

Example: sets implemented as lists without duplicates

- Cannot easily change representation later
- Could distinguish what should be indistinguishable:
  `[1, 2] /= [2, 1]`
- Could create illegal value: `[1, 1]`

Example: Sets

```
module Set where
-- sets are represented as lists w/o duplicates
type Set a = [a]

empty  :: Set a
empty  =  []

insert :: a -> Set a -> Set a
insert x xs  =  ...

isin :: a -> Set a -> Set a
isin x xs  =  ...

size :: Set a -> Integer
size xs  =  ...
```

Exposes everything
Allows nonsense like `Set.size [1,1]`

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
-- Implementation
type Set a = [a]
...
```

- Explicit export list/interface
- But representation still not hidden

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
size (S xs) = length xs
```

Cannot construct values of type `Set` outside of module `Set` because `S` is not exported

```
Test.hs:3:11:  Not in scope:  data constructor 'S'
```


Uniform naming convention: S $\rightsquigarrow$ Set

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Which `Set` is exported?

Slightly more efficient: `newtype`

A newtype declaration creates a new type in much the same way as data. In fact, you can replace the newtype keyword with data. The converse is not true, because:

a newtype type has exactly one constructor with exactly one field. (A mathematician would say that newtype is an isomorphism)

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```


Conceptual insight

Data representation can be hidden
by wrapping data up in a constructor that is not exported

Implementing Set using trees

```
module Set (Set, empty, insert, isin, size) where

-- Interface
empty  :: Set a
insert :: Ord a => a -> Set a -> Set a
isin   :: Ord a => a -> Set a -> Bool
size   :: Set a -> Integer

-- Implementation
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
```

No need for `newtype`:
The representation of `Tree` is hidden
as long as its constructors are hidden

9. Proofs

Proving properties
Structural induction on data structures
Definedness

Aim

Guarantee functional (I/O) properties of software

- Testing can guarantee properties for some inputs.
- Mathematical proof can guarantee properties for all inputs.

QuickCheck is good, proof is better

Beware of bugs in the above code;
I have only proved it correct, not tried it.

Donald E. Knuth, 1977

9.1 Proving properties

What do we prove?

Equations `e1 = e2`

How do we prove them?

By using defining equations `f p = t`

A first, simple example

Remember:
```
        [] ++ ys = ys                 (1)
    (x:xs) ++ ys = x : (xs ++ ys)     (2)
```

Proof of `[1,2] ++ [] = [1] ++ [2]`

```
 [1,2] ++ []
= 1:2:[] ++ []
= 1 : (2:[] ++ [])      -- by def of ++ (2)
= 1 : 2 : ([] ++ [])    -- by def of ++ (2)
= 1 : 2 : []            -- by def of ++ (1)
= 1 : ([] ++ 2:[])      -- by def of ++ (1)
= (1:[]) ++ 2:[]        -- by def of ++ (2)
= [1] ++ [2]
```

Observation:
first used equations from left to right (feels natural),
then from right to left (feels less natural).


```
    [] ++ ys = ys                 (1)
(x:xs) ++ ys = x : (xs ++ ys)     (2)
```

A more natural proof of `[1,2] ++ [] = [1] ++ [2]`:

```
 1:2:[] ++ []
= 1 : (2:[] ++ [])      -- by def of ++ (2)
= 1 : 2 : ([] ++ [])    -- by def of ++ (2)
= 1 : 2 : []            -- by def of ++ (1)

 1:[] ++ 2:[]
= 1 : ([] ++ 2:[])      -- by def of ++ (2)
= 1 : 2 : []            -- by def of ++ (1)
```

Proofs of `e1 = e2` are often better presented
as two reductions to some expression `e`:

```
e1 = ...  = e
e2 = ...  = e
```

Fact If an equation does not contain any variables, it can be proved by evaluating both sides separately and checking that the result is identical.

But how to prove equations with variables, for example

```
length(xs ++ ys) = length xs + length ys
```

Properties of recursive functions are proved by induction.

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

`P(xs)` ↑ *induction hypothesis* (*IH*)

`P(x:xs)` ↑ new variable `x`

This is called *structural induction* on `xs`.
It is a special case of induction on the length of `xs`.

Example: `length` of `++`

```
    length [] = 0                (1)
length (x:xs) = 1 + length xs    (2)

    [] ++ ys = ys                (1)
(x:xs) ++ ys = x : (xs ++ ys)    (2)
```

Lemma `length(xs ++ ys) = length xs + length ys`

Proof by structural induction on `xs`

Base case: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys                -- by def of ++ (1)

length [] + length ys
= 0 + length ys            -- by def of length
= length ys
```


Example: `length` of `++`

```
    length [] = 0                (1)
length (x:xs) = 1 + length xs    (2)

    [] ++ ys = ys                (1)
(x:xs) ++ ys = x : (xs ++ ys)    (2)
```

Lemma `length(xs ++ ys) = length xs + length ys`

Induction step: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++ (2)
= 1 + length(xs ++ ys)        -- by def of length (2)
= 1 + length xs + length ys   -- by IH

length(x:xs) + length ys
= 1 + length xs + length ys   -- by def of length (2)
QED
```

Induction template

Lemma `P(xs)`
Proof by structural induction on `xs`

Base case: `P([])`

Proof of `P([])`

Induction step: `P(x:xs)`

Proof of `P(x:xs)` using IH `P(xs)`

Example: `reverse` of `++`

```
    reverse [] = []                (1)
reverse (x:xs) = reverse xs ++ [x] (2)

    [] ++ ys = ys                  (1)
(x:xs) ++ ys = x : (xs ++ ys)      (2)
```

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`

Base case: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys            -- by def of ++

reverse ys ++ reverse []
= reverse ys ++ []      -- by def of reverse
= reverse ys            -- by Lemma lem_Nil
```

Lemma lem_Nil: `xs ++ [] = xs`
Proof exercise

```
    reverse [] = []                 (1)
reverse (x:xs) = reverse xs ++ [x]  (2)

    [] ++ ys = ys              (1)
(x:xs) ++ ys = x : (xs ++ ys)  (2)
```

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`

Induction step:
`reverse((x:xs) ++ ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))            -- by def of ++
= reverse(xs ++ ys) ++ [x]           -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x]  -- by IH
= reverse ys ++ (reverse xs ++ [x])  -- by Lemma lem_assoc

reverse ys ++ reverse(x:xs)
= reverse ys ++ (reverse xs ++ [x])  -- by def of reverse
```


Lemma: associativity of ++

```
    [] ++ ys = ys              (1)
(x:xs) ++ ys = x : (xs ++ ys)  (2)
```

Lemma lem_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
Proof by structural induction on xs

Base case: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
([] ++ ys) ++ zs
= ys ++ zs            -- by def of ++ (1)
= [] ++ (ys ++ zs)    -- by def of ++ (1, from right to left)
```

Induction step: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
((x:xs) ++ ys) ++ zs
= (x : (xs ++ ys)) ++ zs   -- by def of ++ (2)
= x : ((xs ++ ys) ++ zs)   -- by def of ++ (2)
= x : (xs ++ (ys ++ zs))   -- by IH

(x:xs) ++ (ys ++ zs)
= x : (xs ++ (ys ++ zs))   -- by def of ++ (2)
QED
```

Proof heuristic

- Try QuickCheck
- Try induction
  - Base case: reduce both sides to a common term
    using function defs and lemmas
  - Induction step: reduce both sides to a common term
    using function defs, IH and lemmas
- If base case or induction step fails:
  conjecture, prove and use new lemmas

Two further techniques

- Proof by cases
- Generalization

Example: proof by cases

```
rem x [] = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

Proof by structural induction on xs

Base case: `rem z ([] ++ ys) = rem z [] ++ rem z ys`

```
rem z ([] ++ ys)
= rem z ys              -- by def of ++

rem z [] ++ rem z ys
= [] ++ rem z ys        -- by def of rem
= rem z ys              -- by def of ++
```



```
rem x [] = []
rem x (y:ys) | x==y      = rem x ys
             | otherwise = y : rem x ys
```

Ind. step: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases: `z == x` and `z /= x`

Case `z == x`:

```
rem z ((x:xs) ++ ys)
= rem z (x:(xs ++ ys))    -- by def of ++
= rem z (xs ++ ys)        -- by def of rem and z==x
= rem z xs ++ rem z ys    -- by IH

rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys    -- by def of rem and z==x
```

Case `z /= x`:

```
rem z ((x:xs) ++ ys)
= rem z (x:(xs ++ ys))        -- by def of ++
= x : rem z (xs ++ ys)        -- by def of rem and z/=x
= x : (rem z xs ++ rem z ys)  -- by IH

rem z (x:xs) ++ rem z ys
= (x : rem z xs) ++ rem z ys  -- by def of rem, and z/=x
= x : (rem z xs ++ rem z ys)  -- by def of ++
QED
```

Inefficiency of `reverse`

```
reverse [1,2,3]
= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= ([3] ++ [2]) ++ [1]
= (3 : ([] ++ [2])) ++ [1]
= [3,2] ++ [1]
= 3 : ([2] ++ [1])
= 3 : (2 : ([] ++ [1]))
= [3,2,1]
```


An improvement: `itrev`

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

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

Proof attempt

```
itrev [] xs     = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails: `itrev (x:xs) [] = reverse (x:xs)`

```
 itrev (x:xs) []
 = itrev xs (x:[])     -- by def of itrev
 = itrev xs [x]

 reverse (x:xs)
 = reverse xs ++ [x]   -- by def of reverse
```

Problem: IH not applicable because too specialized: `[]`

Generalization

```
itrev [] xs     = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on `xs`

Induction step: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
 itrev (x:xs) ys
 = itrev xs (x:ys)            -- by def of itrev
 = reverse xs ++ (x:ys)       -- by IH

 reverse (x:xs) ++ ys
 = (reverse xs ++ [x]) ++ ys   -- by def of reverse
 = reverse xs ++ ([x] ++ ys)   -- by Lemma lem_assoc
 = reverse xs ++ ([]++(x:ys))  -- by def of ++
 = reverse xs ++ (x:ys)        -- by def of ++
```

Note: IH is used with `x:ys` instead of `ys`

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Justification: all variables are implicitly ∀-quantified, except for the induction variable.

foldr

Defining functions via `foldr`

- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

For example, if `f` is associative and `` a `f` x = x `` then
`` foldr f a (xs++ys) = foldr f a xs `f` foldr f a ys ``.

Therefore `sum (xs++ys) = sum xs + sum ys`,
`product (xs++ys) = product xs * product ys`, etc.

Proving foldr property

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

If f is associative and a `f` x = x then
foldr f a (xs++ys) = foldr f a xs `f` foldr f a ys.

Proof by induction on xs.
Induction step:

```
foldr f a ((x:xs) ++ ys)
= foldr f a (x : (xs++ys))
= x `f` foldr f a (xs++ys)
= x `f` (foldr f a xs `f` foldr f a ys)     -- by IH
```

```
foldr f a (x:xs) `f` foldr f a ys
= (x `f` foldr f a xs) `f` foldr f a ys
= x `f` (foldr f a xs `f` foldr f a ys)     -- by assoc.  f
```

QED

So, if g xs = foldr f a xs, then g (xs ++ ys) = g xs `f` g ys.
Therefore sum (xs++ys) = sum xs + sum ys,
product (xs++ys) = product xs * product ys, ...

Proof sketch: qsort

```
qsort :: Ord a => [a] -> [a]
qsort []       =  []
qsort (x:xs)   = qsort below ++ [x] ++ qsort above
    where below = [y | y <- xs, y <= x]
          above = [z | y <- xs, x < z]
```

Lemma qsort xs is sorted

Proof by induction on the length of the argument of qsort.

Induction step: In the call qsort (x:xs) we have
length below <= length xs < length(x:xs)
length above <= length xs < length(x:xs)
Therefore qsort below and qsort above are sorted by IH.
By construction below contains only elements (<=x).
Therefore qsort below contains only elements (<=x).
Analogously for above and (x<).
Therefore qsort (x:xs) is sorted.

Is that all? Or should we prove something else about sorting?

How about this sorting function?

```
superquicksort _ = []
```

Every element should occur as often in the output as in the input!

9.2 Structural induction on data structures

Structural induction for Tree

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

To prove property `P(t)` for all finite `t :: Tree a`

Base case: Prove `P(Empty)` and

Induction step: Prove `P(Node x t1 t2)`
assuming the induction hypotheses `P(t1)` and `P(t2)`.

Example

```
inorder :: Tree a -> [a]
inorder Empty  =  []
inorder (Node x t1 t2)  = inorder t1 ++ [x] ++ inorder t2

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty  =  Empty
mapTree f (Node x t1 t2)  =
   Node (f x) (mapTree f t1) (mapTree f t2)
```

Lemma `inorder (mapTree f t) = map f (inorder t)`

Lemma `inorder (mapTree f t) = map f (inorder t)`

Proof by structural induction on t

Base case: we need to show
`inorder (mapTree f Empty) = map f (inorder Empty)`

```
inorder (mapTree f Empty)
= inorder Empty
= []

map f (inorder Empty)
= map f []
= []
QED.
```


Lemma `inorder (mapTree f t) = map f (inorder t)`

Induction step:

IH1: `inorder (mapTree f t1) = map f (inorder t1)`
IH2: `inorder (mapTree f t2) = map f (inorder t2)`

To show: `inorder (mapTree f (Node x t1 t2)) =`
`map f (inorder (Node x t1 t2))`

```
inorder (mapTree f (Node x t1 t2))
= inorder (Node (f x) (mapTree f t1) (mapTree f t2))
    -- by def of mapTree
= inorder (mapTree f t1) ++ [f x] ++ inorder (mapTree f t2)
    -- by def of inorder
= map f (inorder t1) ++ [f x] ++ map f (inorder t2)
    -- by IH1 and IH2

map f (inorder (Node x t1 t2))
= map f (inorder t1 ++ [x] ++ inorder t2)
    -- by def of inorder
= map f (inorder t1) ++ [f x] ++ map f (inorder t2)
    -- by lemma distributivity of map over ++

QED.
```

9.3 Definedness

In the proofs that we make in this course, we make the simplifying assumption that we only deal with defined values.

Two kinds of undefinedness:

`head []`	raises exception
`f x = f x + 1`	does not terminate

What is the problem?

Many familiar laws no longer hold unconditionally:

```
x - x = 0
```

is true only if `x` is a defined value.

Two examples:

- Not true: `head [] - head [] = 0`
- From the nonterminating definition
  `f x = f x + 1`
  we could conclude that $0 = 1$.

Termination

Termination of a function means termination for all inputs.

Restriction:

The proof methods used this far assume that all recursive definitions under consideration terminate.

How to prove termination

Example

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

The function `reverse` terminates because `++` terminates and with each recursive call of `reverse`, the length of the argument becomes smaller.

A Haskell function `f :: T1 -> T` terminates if there exists a *measure function* `m :: T1 ->` $\mathbb{N}$ such that for every recursive call `f p = E(f r)` we have `m p > m r`.

How to prove termination

More generally:

A Haskell function `f :: T1 -> ... -> Tn -> T` terminates if there exists a measure function `m :: T1 -> ... -> Tn ->` $\mathbb{N}$ such that for every recursive call `f p1 ... pn = E(f r1 ... rn)` we have `m p1 ... pn > m r1 ... rn`.

Summary

- In this lecture everything must terminate
- This avoids undefined and infinite values
- This simplifies proofs