Informatik 2: Functional Programming

Tobias Nipkow

Fakultät für Informatik
TU München

http://fp.in.tum.de

Wintersemester 2012/13

January 29, 2013
<table>
<thead>
<tr>
<th>Sprungtabelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>16. Oktober</td>
</tr>
<tr>
<td>23. Oktober</td>
</tr>
<tr>
<td>30. Oktober</td>
</tr>
<tr>
<td>6. November</td>
</tr>
<tr>
<td>13. November</td>
</tr>
<tr>
<td>20. November</td>
</tr>
<tr>
<td>27. November</td>
</tr>
<tr>
<td>4. Dezember</td>
</tr>
<tr>
<td>11. Dezember</td>
</tr>
<tr>
<td>18. Dezember</td>
</tr>
<tr>
<td>8. Januar</td>
</tr>
<tr>
<td>15. Januar</td>
</tr>
<tr>
<td>22. Januar</td>
</tr>
<tr>
<td>29. Januar</td>
</tr>
</tbody>
</table>
Organisatorisches

Functional Programming: The Idea

Basic Haskell

Lists

Proofs

Higher-Order Functions

Type Classes

Algebraic data Types

Modules and Abstract Data Types

Case Study: Huffman Coding

Case Study: Parsing

Lazy evaluation

I/O and Monads

Complexity and Optimization
1. Organisatorisches
Siehe http://fp.in.tum.de
Wochenplan

**Dienstag**  Vorlesung: Gehirn mitbringen

*Harter* Abgabetermin für Übungsblatt

Neues Übungsblatt

**Mi–Fr**  Übungen: Gehirn *und* Laptop mitbringen
Vorlesung orientiert sich stark an
Thompson: *Haskell, the Craft of Functional Programming*

Für Freunde der kompakten Darstellung:
Hutton: *Programming in Haskell*

Für Naturtalente: Es gibt sehr viel Literatur online. Qualität wechselhaft, nicht mit Vorlesung abgestimmt.
Klausur und Hausaufgaben

- Klausur am Ende der Vorlesung
- Wer mindestens 40% der Hausaufgabenpunkte erreicht und die Klausur besteht, bekommt einen Notenbonus von 0.3 (bei bestandener Klausur).
- Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.
2. Functional Programming: The Idea
Functions are pure/mathematical functions:
Always same output for same input

Computation = Application of functions to arguments
Example 1

In Haskell:

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

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
    total = total + i;
```
Example 2

In Haskell:

```haskell
wellknown [] = []
wellknown (x:xs) = wellknown ys ++ [x] ++ wellknown zs
  where ys = [y | y <- xs, y <= x]
       zs = [z | z <- xs, x < z]
```
In Java:

```java
void sort(int[] values) {
    if (values == null || values.length == 0) { return; }
    this.numbers = values;
    number = values.length;
    quicksort(0, number - 1);
}

void quicksort(int low, int high) {
    int i = low, j = high;
    int pivot = numbers[low + (high-low)/2];
    while (i <= j) {
        while (numbers[i] < pivot) { i++; }
        while (numbers[j] > pivot) { j--; }
        if (i <= j) { exchange(i, j); i++; j--; }
    }
    if (low < j) quicksort(low, j);
    if (i < high) quicksort(i, high);
}

void exchange(int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```
There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies.
The other way is to make it so complicated that there are no obvious deficiencies.

From the Turing Award lecture by Tony Hoare (1985)
Characteristics of functional programs

elegant
expressive
concise
readable
predictable  pure functions, no side effects
provable    it’s just (very basic) mathematics!
Aims of functional programming

• Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures

• Minimize time to read and write programs:
  ⇒ reduced development and maintenance time and costs

• Increased confidence in correctness of programs:
  clean and simple syntax and semantics
  ⇒ programs are easier to
    • understand
    • test (Quickcheck!)
    • prove correct
Historic Milestones

1930s

Alonzo Church develops the lambda calculus, the core of all functional programming languages.
Historic Milestones

1950s

John McCarthy (Turing Award 1971) develops Lisp, the first functional programming language.
Robin Milner (FRS, Turing Award 1991) & Co. develop ML, the first modern functional programming language with polymorphic types and type inference.
An international committee of researchers initiates the development of Haskell, a standard lazy functional language.
Popular languages based on FP

F# (Microsoft) = ML for the masses

Erlang (Ericsson) = distributed functional programming

Scala (EPFL) = Java + FP
FP concepts in other languages

Garbage collection: Java, C#, Python, Perl, Ruby, Javascript

Higher-order functions: Java, C#, Python, Perl, Ruby, Javascript

Generics: Java, C#

List comprehensions: C#, Python, Perl 6, Javascript

Type classes: C++ “concepts”
Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# . . .
- It gives you the edge over Millions of Java/C/C++ programmers out there
- FP concepts make you a better programmer, no matter which language you use
- To show you that programming need not be a black art with magic incantations like `public static void` but can be a science
3. Basic Haskell

- Notational conventions
- Type Bool
- Type Integer
- Guarded equations
- Recursion
- Syntax matters
- Types Char and String
- Tuple types
- Do’s and Don’ts
3.1 Notational conventions

\( e :: T \) means that expression \( e \) has type \( T \)

Function types:

\[
\begin{align*}
  f & : : A \times B \rightarrow C \\
  f & : : A \rightarrow B \rightarrow C
\end{align*}
\]

Mathematics

Haskell

Function application:

\[
\begin{align*}
  f(a) & \\
  f(a, b) & \\
  f(g(b)) & \\
  f(a, g(b)) & \\
\end{align*}
\]

\[
\begin{align*}
  f & \ a \\
  f & \ a \ b \\
  f & \ (g \ b) \\
  f & \ a \ (g \ b)
\end{align*}
\]

Prefix binds stronger than infix:

\[
\begin{align*}
  f \ a + b & \quad \text{means} \quad (f \ a) + b \\
  \text{not} \quad f & \ (a + b)
\end{align*}
\]
3.2 Type Bool

Predefined: True False not && || ==

Defining new functions:

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

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

This is an example of pattern matching.
The equations are tried in order. More later.

Is xor x y == xor2 x y true?
Testing with QuickCheck

Import test framework:

```haskell
import Test.QuickCheck
```

Define property to be tested:

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

Note naming convention `prop_`.

Check property with GHCi:

```haskell
> quickCheck prop_xor2
```

GHCi answers

```haskell
+++ OK, passed 100 tests.
```
BoolDemo.hs

For GHCi commands (:l etc) see home page
3.3 Type Integer

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

There is also the type Int of 32-bit integers.
Warning: Integer: 2 ^ 32 = 4294967296
Int: 2 ^ 32 = 0

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

\[
\text{sq} \, :: \, \text{Integer} \rightarrow \text{Integer} \\
\text{sq} \, n \, = \, n \times n
\]

Evaluation:

\[
\text{sq} \, (\text{sq} \, 3) = \text{sq} \, 3 \times \text{sq} \, 3 \\
= (3 \times 3) \times (3 \times 3) \\
= 81
\]

Evaluation of Haskell expressions means

Using the defining equations from left to right.
3.4 Guarded equations

Example: maximum of 2 integers.

max :: Integer -> Integer -> Integer
max x y
    | x >= y     = x
    | otherwise  = y

Haskell also has if-then-else:

max x y = if x >= y then x else y

True?

prop_max_assoc x y z =
    max x (max y z) == max (max x y) z
3.5 Recursion

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

-- pow x n returns x to the power of n

\[
\text{pow :: Integer -> Integer -> Integer}
\]

\[
\text{pow x n = ???}
\]

Cannot write \( x \times \cdots \times x \)  
\( n \) times

Two cases:

\[
\text{pow x n}
\]

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

More compactly:

\[
\text{pow x 0 = 1}
\]
\[
\text{pow x n | n > 0 = x * pow x (n-1)}
\]
Evaluating pow

\[
\begin{align*}
\text{pow } x \ 0 &= 1 \\
\text{pow } x \ n \ | \ n > 0 &= x \times \text{pow } x \ (n-1)
\end{align*}
\]

\[
\text{pow } 2 \ 3 = 2 \times \underbrace{\text{pow } 2 \ 2} \\
&= 2 \times (2 \times \underbrace{\text{pow } 2 \ 1}) \\
&= 2 \times (2 \times (2 \times \underbrace{\text{pow } 2 \ 0})) \\
&= 2 \times (2 \times (2 \times 1)) \\
&= 8
\]

\[
> \text{pow } 2 \ (-1)
\]

GHCi answers

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

\[ \text{pow} \ x \ n \ | \ n > 0 \ = \ x * \text{pow} \ x \ (n-1) \]

versus

\[ \text{pow} \ x \ n \ = \ x * \text{pow} \ x \ (n-1) \]

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

In either case:

State your preconditions clearly!

As a guard, a comment or using QuickCheck:

\[ P \ x \ =\rightarrow \ \text{isDefined}(f \ x) \]

where isDefined \( y \ = \ y \ == \ y \).
Example sumTo

The sum from 0 to $n = n + (n-1) + (n-2) + \ldots + 0$

```haskell
sumTo :: Integer -> Integer
sumTo 0 = 0
sumTo n | n > 0 = n + sumTo (n-1)
```

```haskell
prop_sumTo n =
  n >= 0 ==> sumTo n == n*(n+1) 'div' 2
```

Properties can be \textit{conditional}
Typical recursion patterns for integers

\[
f :: \text{Integer} \rightarrow \ldots
\]
\[
f 0 = e \quad -- \text{base case}
\]
\[
f n \mid n > 0 = \ldots f(n - 1) \ldots \quad -- \text{recursive call(s)}
\]

Always make the base case as simple as possible, typically 0, not 1

Many variations:

- more parameters
- other base cases, e.g. \( f 1 \)
- other recursive calls, e.g. \( f(n - 2) \)
- also for negative numbers
Recursion in general

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

*General problem solving strategy in *any* programming language*
3.6 Syntax matters

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

\[
\begin{align*}
f & \quad x_1 \  \ldots \  \ x_n \\
\mid & \quad test_1 \  = \  e_1 \\
\mid & \quad \ldots \\
\mid & \quad test_n \  = \  e_n
\end{align*}
\]

Each right-hand side \( e_i \) is an expression.
Note: \texttt{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 parenthesised expression $(e)$

Additional syntactic sugar:

- *if* then *else*
- *infix*
- *where*
- ...
Local definitions: where

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

\[ \text{pow4} \ x = x_2 \times x_2 \quad \text{where} \quad x_2 = x \times x \]

\[ \text{pow4} \ x = \text{sq} \ (\text{sq} \ x) \quad \text{where} \quad \text{sq} \ x = x \times x \]

\[ \text{pow8} \ x = \text{sq} \ (\text{sq} \ x_2) \]
\[ \quad \text{where} \quad x_2 = x \times x \]
\[ \quad \text{sq} \ x = x \times x \]

\[ \text{myAbs} \ x \]
\[ \quad | x > 0 \quad = \quad y \]
\[ \quad | \quad \text{otherwise} \quad = \quad -y \]
\[ \quad \text{where} \quad y = x \]
Local definitions: let

\[
\text{let } x = e_1 \text{ in } e_2
\]
defines \(x\) locally in \(e_2\)

Example:

\[
\text{let } x = 2+3 \text{ in } x^2 + 2*x
\]
= 35

Like \(e_2\) where \(x = e_1\)
But can occur anywhere in an expression
where: only after function definitions
In a sequence of definitions, each definition must begin in the same column.

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
--}
3.7 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.
3.8 Tuple types

(\text{True}, \ 'a', \ "abc") :: (\text{Bool}, \ \text{Char}, \ \text{String})

In general:

If \( e_1 :: T_1 \ldots e_n :: T_n \)
then \((e_1, \ldots, e_n) :: (T_1, \ldots, T_n)\)

In mathematics: \( T_1 \times \ldots \times T_n \)
3.9 Do’s and Don’ts
True and False

Never write

\[ b == \text{True} \]

Simply write

\[ b \]

Never write

\[ b == \text{False} \]

Simply write

\[ \text{not}(b) \]
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):
\( f (x, y) = \ldots \)

Usually better:
\[ f \ x \ y = \ldots \]

Just fine:
\[ f \ x \ y = (x + y, x - y) \]
4. Lists

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

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

[[True],[]] ::
Typing rule

If \( e_1 :: T \ldots e_n :: T \)
then \([e_1, \ldots, e_n] :: [T]\)

Graphical notation:

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array{e_1 :: T \ldots e_n :: T} \Rightarrow [e_1, \ldots, e_n] :: [T] \end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
Test

(\text{True, 'c')} ::

[(\text{True, 'c'), (\text{False, 'd')}]] ::

([\text{True, False}], ['c', 'd']) ::
List ranges

\[ [1 \ldots 3] = [1, 2, 3] \]
\[ [3 \ldots 1] = [] \]
\[ ['a' \ldots 'c'] = ['a', 'b', 'c'] \]
Concatenates two lists of the same type:

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

\[ [1, 2] + ['a'] \]
4.1 List comprehension

Set comprehensions:

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

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

List comprehension:

\[
[ x^2 \mid x \leftarrow [1 \ldots 5]]
\]

*The list of all* \(x^2\) *such that* \(x\) *is an element of* \([1 \ldots 5]\)*
List comprehension — Generators

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

\[
[ \text{toLower } c \mid c <- "Hello, World!" ]
= "hello, world!"
\]

\[
[ (x, \text{even } x) \mid x <- [1 .. 3]]
= [(1, False), (2, True), (3, False)]
\]

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

\text{pattern} <- \text{list expression}

is called a \text{generator}

Precise definition of \text{pattern} later.
List comprehension — Tests

\[
[ x \times x \mid x \leftarrow [1 .. 5], \text{odd } x]
= [1, 9, 25]
\]

\[
[ x \times x \mid x \leftarrow [1 .. 5], \text{odd } x, x > 3]
= [25]
\]

\[
[ \text{toLowerCase } c \mid c \leftarrow "Hello, World!", \text{isAlpha } c]
= "helloworld"
\]

Boolean expressions are called tests.
Defining functions by list comprehension

Example

```haskell
defining_funs.hs

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

⇒ factors 15 = [1, 3, 5, 15]

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

⇒ prime 15 = False

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

⇒ primes 100 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...
```

61
List comprehension — General form

\[ [ \text{expr} \mid E_1, \ldots, E_n ] \]

where \text{expr} is an expression and each \( E_i \) is a generator or a test
Multiple generators

\[(i,j) \mid i \leftarrow [1..2], j \leftarrow [7..9]\]

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

Analogy: each generator is a for loop:

for all \( i \leftarrow [1..2] \)
  for all \( j \leftarrow [7..9] \)
    ...

Key difference:

Loops \textit{do} something
Expressions \textit{produce} something
Dependent generators

\[(i, j) \mid i \leftarrow [1..3], j \leftarrow [i..3]\]

= \[(1, j) \mid j \leftarrow [1..3]\] ++
  \[(2, j) \mid j \leftarrow [2..3]\] ++
  \[(3, j) \mid j \leftarrow [3..3]\]

= [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
The meaning of list comprehensions

\[ [e \mid x \leftarrow [a_1, \ldots, a_n]] \]
= (let \( x = a_1 \) in \([e] \)) ++ \( \cdots \) ++ (let \( x = a_n \) in \([e] \))

\[ [e \mid b] \]
= if \( b \) then \([e] \) else \( [] \)

\[ [e \mid x \leftarrow [a_1, \ldots, a_n], E] \]
= (let \( x = a_1 \) in \([e \mid E] \)) ++ \( \cdots \) ++
  (let \( x = a_n \) in \([e \mid E] \))

\[ [e \mid b, E] \]
= if \( b \) then \([e \mid E] \) else \( [] \)
Example: concat

\[
\text{concat } xss = [x \mid xs <- xss, x <- xs]
\]

\[
\text{concat } [[1,2], [4,5,6]]
\]

\[
= [x \mid xs <- [[1,2], [4,5,6]], x <- xs]
\]

\[
= [x \mid x <- [1,2]] ++ [x \mid x <- [4,5,6]]
\]

\[
= [1,2] ++ [4,5,6]
\]

\[
= [1,2,4,5,6]
\]

What is the type of concat?

\[
[[a]] \to [a]
\]
4.2 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

⇒ length :: [T] -> Int for all types T
Type variable syntax

Type variables must start with a lower-case letter
Typically: a, b, c, ...
Two kinds of polymorphism

Subtype polymorphism as in Java:

\[ f :: T \rightarrow U \quad T' \leq T \]
\[ f :: T' \rightarrow U \]

(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:

Types may contain type variables ("parameters")

\[ f :: T \]
\[ f :: T[U/a] \]

where \( T[U/a] = \"T with a replaced by U\"

Example: \((a \rightarrow a)[Bool/a] = Bool \rightarrow Bool\)

(Often called \(ML\)-style polymorphism)
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, last :: [a] -> a
head "list" = 'l',  last "list" = 't'

tail, init :: [a] -> [a]
tail "list" = "ist",  init "list" = "lis"

take, drop :: Int -> [a] -> [a]
take 3 "list" = "lis",  drop 3 "list" = "t"

-- 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')]

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

-- A property
prop_zip xs ys = length xs == length ys ==> unzip(zip xs ys) == (xs, ys)
Haskell libraries

- Prelude and much more
- Hoogle — searching the Haskell libraries
- Hackage — a collection of Haskell packages

See Haskell pages and Thompson’s book for more information.
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

What exactly is the type of sum, prod, +, *, ==, ...???
Polymorphism versus Overloading

Polymorphism: one definition, many types
Overloading: different definition 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 \( a \to a \to a \) for any type of class Num

- Class \( \text{Num} \) is the class of \textit{numeric types}.
- Predefined numeric types: \text{Int}, \text{Integer}, \text{Float}
- Types of class \( \text{Num} \) 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 possess:
  
  
  ```haskell
  (==) :: 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 possess:

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

More on type classes later. Don’t confuse with OO classes.
null xs = xs == []

Why?

== on [a] must (potentially) call == on a

Better:

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

In Prelude!
Warning: QuickCheck and polymorphism

QuickCheck does not work well on polymorphic properties

Example
QuickCheck does not find a counterexample to

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

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

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

Now QuickCheck works
Conditional properties have result type \( \text{Property} \)

**Example**

\[
\text{prop\_rev10 :: [Int] \rightarrow Property}
\]

\[
\text{prop\_rev10 \_xs =}
\]

\[
\text{length \_xs \_\leq \_10 \rightarrow reverse(reverse \_xs) == \_xs}
\]
4.3 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]
PictureDemo.hs
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
Exercise

Ensure that the lower left square of chessboard $n$ is always black.
4.4 Pattern matching

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

<table>
<thead>
<tr>
<th>syntactic sugar</th>
<th>in reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>3 : []</td>
</tr>
<tr>
<td>[2, 3]</td>
<td>2 : 3 : []</td>
</tr>
<tr>
<td>[1, 2, 3]</td>
<td>1 : 2 : 3 : []</td>
</tr>
<tr>
<td>([x_1, \ldots, x_n])</td>
<td>(x_1 : \ldots : x_n : [])</td>
</tr>
</tbody>
</table>

Note: \(x : y : zs = x : (y : zs)\)
\((:\)\) associates to the right
Every list is either

\[
\text{[] or of the form }
\]
x : xs  \text{ where}

\[
x \text{ is the } \textit{head} \text{ (first element, Kopf), and}
\]
xs  \text{ is the } \textit{tail} \text{ (rest list, Rumpf)}

[] and (:) are called \textit{constructors} because every list can be \textit{constructed uniquely} from them.

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.

⇒ Patterns allow unique decomposition = pattern matching.

A pattern can be

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

Note: True and False are constructors, too!
Function definitions by pattern matching

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

\[
\begin{align*}
    f \ pat_1 & = e_1 \\
    \vdots & \\
    f \ pat_n & = e_n
\end{align*}
\]

If \( f \) has multiple arguments:

\[
\begin{align*}
    f \ pat_{11} \ldots \ pat_{1k} & = e_1 \\
    \vdots & \\
\end{align*}
\]

Conditional equations:

\[
f \ patterns \mid condition = e
\]

When \( f \) is called, the equations are tried in the given order
Function definitions by pattern matching

Example (contrived)

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

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

asc3 :: Ord a => [a] -> Bool
asc3 (x : y : z : _) = x < y && y < z
asc3 (x : y : _) = x < y
asc3 _ = True
```
4.5 Recursion over lists

Example

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

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

sum :: Num a => [a] -> a
sum [] = 0
sum (x : xs) = x + sum xs
Primitive recursion on lists:

\[ f \; [] = base \quad -- \quad \text{base case} \]
\[ f \; (x : xs) = rec \quad -- \quad \text{recursive case} \]

- \textit{base}: no call of \( f \)
- \textit{rec}: only call(s) \( f \; xs \)

\( f \) may have additional parameters.
Finding 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)
Example

\[
\text{inSort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\
\text{inSort} \ [\] \quad = \quad [\] \\
\text{inSort} \ (x:xs) \quad = \quad \text{ins} \ x \ (\text{inSort} \ xs)
\]

\[
\text{ins} :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{ins} \ x \ [\] \quad = \quad [x] \\
\text{ins} \ x \ (y:ys) \mid x \leq y \quad = \quad x : y : ys \\
\mid \text{otherwise} \quad = \quad y : \text{ins} \ x \ ys
\]
Beyond primitive recursion: Multiple arguments

Example

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

Alternative definition:

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

zip’ is undefined for lists of different length!
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]
Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

\[
\text{ups } [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]
\]

\[
\text{ups} :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \rightarrow [[a]]
\]

\[
\begin{align*}
\text{ups2} (x:xs) (y:ys) &= \begin{cases} \\
\text{ups2} xs (x:y:ys) & \text{if } x \geq y \\
\text{reverse } (y:ys) : \text{ups2} (x:xs) [] & \text{otherwise}
\end{cases} \\
\text{ups2} (x:xs) [] &= \text{ups2} xs [x] \\
\text{ups2} [] ys &= [\text{reverse } ys]
\end{align*}
\]

\[
\text{ups} :: \text{Ord } a \Rightarrow [a] \rightarrow [[a]]
\]

\[
\text{ups } xs = \text{ups2 } xs []
\]
How can we quickCheck the result of ups?
Identifiers of list type end in ‘s’:
xs, ys, zs, ...
Mutual recursion

Example

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

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

\[ x = y + 5 \]
\[ y = x + 1 \quad \text{where} \quad x = 7 \]
\[ f \ y = y + x \]

\[ > f 3 \]

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

\[ x = y + 5 \]
\[ y = x + 1 \text{ where } x = 7 \]

\[ f \ y = y + x \]

\[ > f \ 3 \]

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

\[x = y + 5\]
\[y = x + 1 \text{ where } x = 7\]
\[f \ y = y + x\]

\[> f \ 3\]

**Binding occurrence**

**Bound occurrence**

**Scope of binding**
Scoping by example

\[ x = y + 5 \]
\[ y = x + 1 \quad \text{where} \quad x = 7 \]
\[ f \ y = y + x \]

> \( f \ 3 \)

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

\[ x = y + 5 \]
\[ y = x + 1 \quad \text{where} \quad x = 7 \]
\[ f \ y = y + x \]

> \textbf{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
5. Proofs

Proving properties
Definedness
Aim

Guarentee 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
5.1 Proving properties

What do we prove?

Equations $e_1 = e_2$

How do we prove them?

By using defining equations $f \ p = t$
A first, simple example

Remember:

\[
\begin{align*}
  [] ++ ys &= ys \\
  (x:xs) ++ ys &= x : (xs ++ ys)
\end{align*}
\]

Proof of \([1,2] ++ [] = [1] ++ [2]::

\[
\begin{align*}
  &1:2:[[]] ++ [] \\
  &= 1 : (2:[] ++ []) \quad \text{-- by def of ++} \\
  &= 1 : 2 : ([] ++ []) \quad \text{-- by def of ++} \\
  &= 1 : 2 : [] \quad \text{-- by def of ++} \\
  &= 1 : ([] ++ 2:[]) \quad \text{-- by def of ++} \\
  &= 1:[[]] ++ 2:[] \quad \text{-- by def of ++}
\end{align*}
\]

Observation: first used equations from left to right (ok), then from right to left (strange!)
A more natural proof of $[1,2] ++ [] = [1] ++ [2]$: 

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

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

Proofs of $e_1 = e_2$ are often better presented as two reductions to some expression $e$:

$$e_1 = \ldots = e$$
$$e_2 = \ldots = e$$
**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 *associativity* of `++`:

\[(xs ++ ys) ++ zs = xs ++ (ys ++ zs)\]
Properties of recursive functions are proved by induction

Induction on natural numbers: see Diskrete Strukturen

Induction on lists: here and now
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 $P(x:xs)$

This is called *structural induction* on $xs$.
It is a special case of induction on the length of $xs$. 
Example: associativity of ++

**Lemma** app_assoc: \((xs ++ ys) ++ zs = xs ++ (ys ++ zs)\)

**Proof** by structural induction on \(xs\)

Base case:
To show: \(([] ++ ys) ++ zs = [] ++ (ys ++ zs)\)

\[
([] ++ ys) ++ zs \\
= ys ++ zs \\
= [] ++ (ys ++ zs)
\]

-- by def of ++

Induction step:
To show: \(((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)\)

\[
((x:xs) ++ ys) ++ zs \\
= (x : (xs ++ ys)) ++ zs \\
= x : ((xs ++ ys) ++ zs) \\
= x : (xs ++ (ys ++ zs)) \\
= x : (xs ++ (ys ++ zs))
\]

-- by def of ++

-- by IH

\[
(x:xs) ++ (ys ++ zs) \\
= x : (xs ++ (ys ++ zs))
\]

-- by def of ++
Lemma $P(xs)$

Proof by structural induction on $xs$

Base case:
To show: $P([])$

Proof of $P([])$

Induction step:
To show: $P(x:xs)$

Proof of $P(x:xs)$ using IH $P(xs)$
Example: length of ++

Lemma \[\text{length}(xs ++ ys) = \text{length} \; xs + \text{length} \; ys\]

Proof by structural induction on \(xs\)

Base case:

To show: \(\text{length} ([] ++ ys) = \text{length} \; [] + \text{length} \; ys\)

\[
\begin{align*}
\text{length} ([] ++ ys) \\
= \text{length} \; ys & \quad \text{-- by def of ++} \\
\text{length} \; [] + \text{length} \; ys \\
= 0 + \text{length} \; ys & \quad \text{-- by def of length} \\
= \text{length} \; ys \\
\end{align*}
\]

Induction step:

To show: \(\text{length}((x:xs)++ys) = \text{length}(x:xs) + \text{length} \; ys\)

\[
\begin{align*}
\text{length}((x:xs) ++ ys) \\
= \text{length}(x : (xs ++ ys)) & \quad \text{-- by def of ++} \\
= 1 + \text{length}(xs ++ ys) & \quad \text{-- by def of length} \\
= 1 + \text{length} \; xs + \text{length} \; ys & \quad \text{-- by IH} \\
\text{length}(x:xs) + \text{length} \; ys \\
= 1 + \text{length} \; xs + \text{length} \; ys & \quad \text{-- by def of length}
\end{align*}
\]
Example: reverse of ++

Lemma $\text{reverse}(xs ++ ys) = \text{reverse} ys ++ \text{reverse} xs$

Proof by structural induction on $xs$

Base case:

To show: $\text{reverse} ([] ++ ys) = \text{reverse} ys ++ \text{reverse} []$

\[
\begin{align*}
\text{reverse} ([] ++ ys) &= \text{reverse} ys \\
\text{reverse} ys ++ \text{reverse} [] &= \text{reverse} ys ++ [] \\
\text{reverse} ys &= \text{reverse} ys \\
\end{align*}
\]

-- by def of ++

-- by def of reverse

-- by Lemma app_Nil2

Lemma app_Nil2: $xs ++ [] = xs$

Proof exercise
Induction step:

To show: \( \text{reverse}((x:xs)++ys) = \text{reverse} ys ++ \text{reverse}(x:xs) \)

\[
\begin{align*}
\text{reverse}((x:xs) ++ ys) &= \text{reverse}(x : (xs ++ ys)) \quad -- \text{by def of ++} \\
&= \text{reverse}(xs ++ ys) ++ [x] \quad -- \text{by def of reverse} \\
&= (\text{reverse} ys ++ \text{reverse} xs) ++ [x] \quad -- \text{by IH} \\
&= \text{reverse} ys ++ (\text{reverse} xs ++ [x]) \quad -- \text{by Lemma app_assoc} \\
\end{align*}
\]

\[
\begin{align*}
\text{reverse} ys ++ \text{reverse}(x:xs) &= \text{reverse} ys ++ (\text{reverse} xs ++ [x]) \quad -- \text{by def of reverse} \\
\end{align*}
\]
• Try QuickCheck
• Try to evaluate both sides to common term
• Try induction
  • Base case: reduce both sides to a common term using functiondefs 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 tricks

- Proof by cases
- Generalization
Example: proof by cases

\[
\begin{align*}
\text{rem } x \; \text{[]} & = \text{[]} \\
\text{rem } x \; (y:ys) & | \; x==y \quad = \text{rem } x \; ys \\
& | \; \text{otherwise} \quad = y : \text{rem } x \; ys
\end{align*}
\]

**Lemma** \( \text{rem } z \; (xs ++ ys) = \text{rem } z \; xs ++ \text{rem } z \; ys \)

**Proof** by structural induction on \( xs \)

**Base case:**
To show: \( \text{rem } z \; (\text{[]} ++ ys) = \text{rem } z \; \text{[]} ++ \text{rem } z \; ys \)
\[
\begin{align*}
\text{rem } z \; (\text{[]} ++ ys) \\
= \text{rem } z \; ys \quad \quad \quad \quad \quad \ldots \quad \text{by def of ++}
\end{align*}
\]
\[
\begin{align*}
\text{rem } z \; \text{[]} ++ \text{rem } z \; ys \\
= \text{rem } z \; ys \quad \quad \quad \quad \quad \ldots \quad \text{by def of rem and ++}
\end{align*}
\]
`rem x [] = []`
`rem x (y:ys) | x==y = rem x ys
| otherwise = y : rem x ys`

Induction step:
To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`
Proof by cases:

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

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

Inefficiency of reverse

\[
\text{reverse } [1,2,3] \\
= \text{reverse } [2,3] ++ [1] \\
= (\text{reverse } [3] ++ [2]) ++ [1] \\
= ((\text{reverse } []) ++ [3]) ++ [2]) ++ [1] \\
= ([3] ++ [2]) ++ [1] \\
= (3 : ([3] ++ [2])) ++ [1] \\
= [3,2] ++ [1] \\
= 3 : ([2] ++ [1]) \\
= 3 : (2 : ([] ++ [1])) \\
= [3,2,1]
\]
An improvement: itrev

\[
\text{itrev} :: [a] \to [a] \to [a]
\text{itrev} [] \, xs = xs
\text{itrev} (x:xs) \, ys = \text{itrev} \, xs \, (x:ys)
\]

\[
\text{itrev} \, [1,2,3] \, []
= \text{itrev} \, [2,3] \, [1]
= \text{itrev} \, [3] \, [2,1]
= \text{itrev} \, [] \, [3,2,1]
= [3,2,1]
\]
Proof attempt

**Lemma** \( \text{itrev \hspace{1em} xs \hspace{1em} [] \hspace{1em} = \hspace{1em} reverse \hspace{1em} xs} \)

**Proof** by structural induction on \( xs \)

Induction step fails:

To show: \( \text{itrev \hspace{1em} (x:x) \hspace{1em} [] \hspace{1em} = \hspace{1em} reverse \hspace{1em} xs} \)

\( \text{itrev \hspace{1em} (x:x) \hspace{1em} []} \)

\( = \text{itrev \hspace{1em} xs \hspace{1em} [x]} \quad -- \text{by def of itrev} \)

\( \text{reverse \hspace{1em} (x:xs)} \)

\( = \text{reverse \hspace{1em} xs \hspace{1em} ++ \hspace{1em} [x]} \quad -- \text{by def of reverse} \)

Problem: IH not applicable because too specialized: []
Lemma \textit{itrev \,xs \,ys = reverse \,xs \,++ \,ys}

Proof by structural induction on \textit{xs}

Induction step:
To show: \textit{itrev \,(x:xs) \,ys = reverse \,(x:xs) \,++ \,ys}

\textit{itrev \,(x:xs) \,ys}

\hspace{1cm} \textit{= itrev \,xs \,(x:ys)} \quad \textbf{-- by def of itrev}

\hspace{1cm} \textit{= reverse \,xs \,++ \,(x:ys)} \quad \textbf{-- by IH}

\textit{reverse \,(x:xs) \,++ \,ys}

\hspace{1cm} \textit{= (reverse \,xs \,++ \,[x]) \,++ \,ys} \quad \textbf{-- by def of reverse}

\hspace{1cm} \textit{= reverse \,xs \,++ \,([x] \,++ \,ys)} \quad \textbf{-- by Lemma \,app\_assoc}

\hspace{1cm} \textit{= reverse \,xs \,++ \,(x:ys)} \quad \textbf{-- by def of ++}

Note: IH is used with \textit{x:ys} instead of \textit{ys}
When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Justification: all variables are implicitly $\forall$-quantified, except for the induction variable.
Induction on the length of a list

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) (also for above).
Therefore qsort below and qsort above are sorted by IH.
By construction below contains only elements (<=x).
Therefore qsort below contains only elements (<=x) (proof!).
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?

```haskell
superquicksort _ = []
```

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

Simplifying assumption, implicit so far:

No undefined values

Two kinds of undefinedness:

- head [] raises exception
- \( f \ x = f \ x + 1 \) does not terminate

Undefinedness can be handled, too.
But it complicates life
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: \( \text{head } [] - \text{head } [] = 0 \)
- From the nonterminating definition
  \[ f \ x = f \ x + 1 \]
  we could conclude that \( 0 = 1 \).
Termination of a function means termination for all inputs.

Restriction:

The proof methods in this chapter assume that all recursive definitions under consideration terminate.

Most Haskell functions we have seen so far terminate.
How to prove termination

Example
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
terminates because ++ terminates and with each recursive call of reverse, the length of the argument becomes smaller.

A function \( f :: T_1 \rightarrow T \) terminates if there is a measure function \( m :: T_1 \rightarrow \mathbb{N} \) such that
- for every defining equation \( f \ p = t \)
- and for every recursive call \( f \ r \ \text{in} \ t: \ m \ p > m \ r \).

Note:
- All primitive recursive functions terminate.
- \( m \) can be defined in Haskell or mathematics.
- The conditions above can be refined to take special Haskell features into account, eg sequential pattern matching.
More generally: $f :: T_1 \to \ldots \to T_n \to T$ terminates if there is a measure function $m :: T_1 \to \ldots \to T_n \to \mathbb{N}$ such that

- for every defining equation $f \ p_1 \ldots \ p_n = t$
- and for every recursive call $f \ r_1 \ldots \ r_n$ in $t$:
  $m \ p_1 \ldots \ p_n > m \ r_1 \ldots \ r_n$. 
Haskell allows infinite values, in particular infinite lists.

Example: \([1, 1, 1, \ldots]\)

Infinite objects must be constructed by recursion:

\[
\text{ones} = 1 : \text{ones}
\]

Because we restrict to terminating definitions in this chapter, infinite values cannot arise.

Note:

- By termination of functions we really mean termination on \textit{finite} values.
- For example \texttt{reverse} terminates only on finite lists.

This is fine because we can only construct finite values anyway.
How can infinite values be useful?
Because of “lazy evaluation”.
More later.
If we use arithmetic equations like \( x - x = 0 \) unconditionally, we can “lose” exceptions:

\[
\text{head } xs - \text{head } xs = 0
\]

is only true if \( xs \neq [] \)

In such cases, we can prove equations \( e1 = e2 \) that are only partially correct:

If for some values for the variables in \( e1 \) and \( e2 \)
\( e1 \) and \( e2 \) do not produce a runtime exception
then they evaluate to the same value.
Summary

• In this chapter everything must terminate
• This avoids undefined and infinite values
• This simplifies proofs
6. 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
Extensionality
Curried functions
More library functions
Case study: Counting words
Recall [Pic is short for Picture]

\[
\text{alterH} :: \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic}
\]

\[
\text{alterH} \ \text{pic1} \ \text{pic2} \ 1 = \text{pic1}
\]

\[
\text{alterH} \ \text{pic1} \ \text{pic2} \ \text{n} = \text{beside pic1} \ (\text{alterH pic2 pic1 (n-1)})
\]

\[
\text{alterV} :: \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic}
\]

\[
\text{alterV} \ \text{pic1} \ \text{pic2} \ 1 = \text{pic1}
\]

\[
\text{alterV} \ \text{pic1} \ \text{pic2} \ \text{n} = \text{above pic1} \ (\text{alterV pic2 pic1 (n-1)})
\]

Very similar. Can we avoid duplication?

\[
\text{alt} :: \ (\text{Pic} \to \text{Pic} \to \text{Pic}) \to \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic}
\]

\[
\text{alt} \ \text{f} \ \text{pic1} \ \text{pic2} \ 1 = \text{pic1}
\]

\[
\text{alt} \ \text{f} \ \text{pic1} \ \text{pic2} \ \text{n} = \text{f pic1} \ (\text{alt f pic2 pic1 (n-1)})
\]

\[
\text{alterH pic1 pic2 n} = \text{alt beside pic1 pic2 n}
\]

\[
\text{alterV pic1 pic2 n} = \text{alt above pic1 pic2 n}
\]
Higher-order functions:
Functions that take functions as arguments

... \rightarrow (\ldots \rightarrow \ldots) \rightarrow \ldots

Higher-order functions capture patterns of computation
6.1 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:

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

map f [] = []
map f (x:xs) = f x : map f xs
Evaluating \texttt{map}

\begin{equation*}
\text{map } f \; [] \; = \; []
\end{equation*}
\begin{equation*}
\text{map } f \; (x:xs) \; = \; f \; x \; : \; \text{map } f \; xs
\end{equation*}

\begin{align*}
\text{map } \text{sqr} \; [1, -2] \\
= \text{map } \text{sqr} \; (1 : -2 : []) \\
= \text{sqr} \; 1 \; : \; \text{map } \text{sqr} \; (-2 : []) \\
= \text{sqr} \; 1 \; : \; \text{sqr} \; (-2) \; : \; (\text{map } \text{sqr} \; []) \\
= \text{sqr} \; 1 \; : \; \text{sqr} \; (-2) \; : \; [] \\
= \; 1 \; : \; 4 \; : \; [] \\
= \; [1, 4]
\end{align*}
Some properties of map

\[
\text{length \ (map \ f \ xs) = length \ xs}
\]

\[
\text{map \ f \ (xs ++ ys) = map \ f \ xs ++ map \ f \ ys}
\]

\[
\text{map \ f \ (\text{reverse} \ xs) = \text{reverse} \ (\text{map} \ f \ xs)}
\]

Proofs by induction
QuickCheck and function variables

QuickCheck does not work automatically for properties of function variables.

It needs to know how to generate and print functions.

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

Example

```haskell
prop_map_even :: [Int] -> [Int] -> Bool
prop_map_even xs ys =
    map even (xs ++ ys) = map even xs ++ map even ys
```
6.2 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]
Predefined in Prelude.
Two possible definitions:

\[
\text{filter } p \; \text{x} = [x \mid x \leftarrow \text{x}, \; p \; x]
\]

\[
\text{filter } p \; [] = []
\]

\[
\text{filter } p \; (x:xs) \mid p \; x = x : \text{filter } p \; xs
\]

\[
\mid \text{otherwise} = \text{filter } p \; xs
\]
Some properties of filter

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

filter \ p \ (\text{reverse} \ xs) = \text{reverse} \ (filter \ p \ xs)

Proofs by induction
6.3 Combining the elements of a list: \texttt{foldr}

**Example**

\[
\begin{align*}
\text{sum} \; [] & \; = \; 0 \\
\text{sum} \; (x:xs) & \; = \; x + \text{sum} \; xs \\
\text{sum} \; [x_1, \ldots, x_n] & \; = \; x_1 + \ldots + x_n + 0 \\
\text{concat} \; [] & \; = \; [] \\
\text{concat} \; (xs:xss) & \; = \; xs \; ++ \; \text{concat} \; xss \\
\text{concat} \; [xs_1, \ldots, xs_n] & \; = \; xs_1 \; ++ \; \ldots \; ++ \; xs_n \; ++ \; []
\end{align*}
\]
foldr

\[
\text{foldr} \ (\oplus) \ z \ [x_1, \ldots, x_n] = x_1 \oplus \ldots \oplus x_n \oplus z
\]

Defined in Prelude:

\[
foldr :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a
\]

\[
foldr \ f \ a \ [] = a \\
foldr \ f \ a \ (x:xs) = x \ 'f' \ foldr \ f \ a \ xs
\]

Applications:

\[
\text{sum} \ xs = \text{foldr} \ (+) \ 0 \ xs
\]

\[
\text{concat} \ xss = \text{foldr} \ (+++) \ [] \ xss
\]

What is the most general type of foldr?
foldr

foldr f a [] = a
foldr f a (x:xs) = x \(\cdot\) 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 + 0)
= -1
More applications of foldr

\[
\begin{align*}
\text{product } xs &= \text{foldr } (*) \ 1\ \ xs \\
\text{and } xs &= \text{foldr } (\&\&) \ True \ xs \\
\text{or } xs &= \text{foldr } (||) \ False \ xs \\
\text{inSort } xs &= \text{foldr } \text{ins} \ [\] \ xs
\end{align*}
\]
Quiz

What is

\[ \text{foldr}(:) \ ys \ xs \]

Example: \[ \text{foldr}(:) \ ys \ (1:2:3:[]) = 1:2:3:ys \]

\[ \text{foldr}(:) \ ys \ xs = ??? \]

Proof by induction on \( xs \) (Exercise!)
Defining functions via foldr

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

Example

If $f$ is associative and $a \ 'f' \ x = x$ then
$\text{foldr } f \ a \ (xs++ys) = \text{foldr } f \ a \ xs \ 'f' \ \text{foldr } f \ a \ ys$.

Proof by induction on $xs$. Induction step:
$\text{foldr } f \ a \ ((x:xs) ++ ys) = \text{foldr } f \ a \ (x : (xs++ys))$
$= x 'f' \ \text{foldr } f \ a \ (xs++ys)$
$= x 'f' \ (\text{foldr } f \ a \ xs \ 'f' \ \text{foldr } f \ a \ ys) \quad -- \text{by IH}$
$\text{foldr } f \ a \ (x:xs) 'f' \ \text{foldr } f \ a \ ys$
$= (x 'f' \ \text{foldr } f \ a \ xs) 'f' \ \text{foldr } f \ a \ ys$
$= x 'f' \ (\text{foldr } f \ a \ xs \ 'f' \ \text{foldr } f \ a \ ys) \quad -- \text{by assoc.}$

Therefore, if $g \ xs = \text{foldr } f \ a \ xs$,
then $g \ (xs ++ ys) = g \ xs \ 'f' \ g \ ys$.

Therefore $\text{sum } (xs++ys) = \text{sum } xs + \text{sum } ys$,
$\text{product } (xs++ys) = \text{product } xs \ * \ \text{product } ys$,...
6.4 Lambda expressions

Consider

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

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

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

is the anonymous function with

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

In mathematics: \( x \mapsto x \times x \)

Evaluation:

\[ (\lambda x \rightarrow x \times x) \ 3 \ = \ 3 \times 3 \ = \ 9 \]

Usage:

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

\( (\lambda x \rightarrow e_1) \ e_2 \)

\( x \): formal parameter
\( e_1 \): result
\( e_2 \): actual parameter

Why “\(\lambda\)”?  
The logician Alonzo Church invented \emph{lambda calculus} in the 1930s

Logicians write \( \lambda x. \ e \) instead of \( \\lambda x \rightarrow e \)
Typing lambda expressions

Example

\( (\lambda x \to x > 0) :\) Int \to Bool
because \( x :\) Int implies \( x > 0 :\) Bool

The general rule:

\( (\lambda x \to e) : T_1 \to T_2 \)
if \( x : T_1 \) implies \( e : T_2 \)
Sections of infix operators

(+ 1) means \(\lambda x \rightarrow x + 1\)
(2 *) means \(\lambda x \rightarrow 2 * x\)
(2 \(^\wedge\)) means \(\lambda x \rightarrow 2 ^ x\)
(^ 2) means \(\lambda x \rightarrow x ^ 2\)

etc

Example

squares xs = map (^ 2) xs
List comprehension

Just syntactic sugar for combinations of map

\[ [f x | x <- xs] = \text{map } f \text{ } xs \]

filter

\[ [x | x <- xs, p \ x] = \text{filter } p \text{ } xs \]

and concat

\[ [f \ x \ y | x <- xs, y <- ys] = \text{concat } (\text{map } \text{ (} \text{map } f \text{ } ys \text{) } xs) \]
6.5 Extensionality

Two functions are equal if for all arguments they yield the same result

\[
\begin{align*}
f, g : : \ & T_1 \to T: \\
& \forall a. f\ a = g\ a \\
& \frac{}{f = g}
\end{align*}
\]

\[
\begin{align*}
f, g : : \ & T_1 \to T_2 \to T: \\
& \forall a, b. f\ a\ b = g\ a\ b \\
& \frac{}{f = g}
\end{align*}
\]
6.6 Curried functions
A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]
\[ f \ x \ y \ = \ x+y \quad f \ x \ = \ \lambda y \rightarrow x+y \]

Both mean the same:

\[ f \ a \ b \quad (f \ a) \ b \]
\[ = a + b \quad = (\lambda y \rightarrow a + y) \ b \]
\[ = a + b \]

The trick: any function of two arguments can be viewed as a function of the first argument that returns a function of the second argument
In general

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

\[ T_1 \rightarrow T_2 \rightarrow T \]

is just syntactic sugar for

\[ T_1 \rightarrow (T_2 \rightarrow T) \]

\[ f e_1 e_2 \]

is just syntactic sugar for

\[ (f e_1) e_2 \]

\[ :: T_2 \rightarrow T \]

Analogously for more arguments


\[ T_1 \to (T_2 \to T) \neq (T_1 \to T_2) \to T \]

Example
\[
\begin{align*}
 f &: \text{Int} \to (\text{Int} \to \text{Int}) \\
 g &: (\text{Int} \to \text{Int}) \to \text{Int} \\
 f \ x \ y &= x + y \\
 g \ h &= h \ 0 + 1
\end{align*}
\]

Application is not associative:
\[
(f \ e_1) \ e_2 \neq f \ (e_1 \ e_2)
\]

Example
\[
\begin{align*}
 (f \ 3) \ 4 &= f \ (3 \ 4) \\
 g \ (\text{id} \ \text{abs}) &= (g \ \text{id}) \ \text{abs}
\end{align*}
\]
head tail xs

Correct?
Partial application

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

Example
Instead of $\text{sum} \; \text{xs} = \text{foldr} \; (+) \; 0 \; \text{xs}$
just define $\text{sum} = \text{foldr} \; (+) \; 0$

In general:
If $f :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$
and $a_1 :: T_1, \ldots, a_m :: T_m$ and $m \leq n$
then $f \; a_1 \; \ldots \; a_m :: T_{m+1} \rightarrow \ldots \rightarrow T_n \rightarrow T$
6.7 More library functions

\((\cdot)\) :: (b -> c) -> (a -> b) -> f \cdot g = \lambda x -> f (g x)

Example

head2 = head . tail

head2 [1,2,3]
= (head . tail) [1,2,3]
= (\lambda x -> head (tail x)) [1,2,3]
= head (tail [1,2,3])
= head [2,3]
= 2
const :: a -> (b -> a)
const x = \_ -> x

curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \ x y -> f(x, y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f = \(x, y) -> f x y
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"
6.8 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*

Unix pipes!
Step 1: Break input into words

"never say never again"

\[
\text{function } \begin{array}{c}
\text{words}
\end{array}
\]

["never", "say", "never", "again"]

Predefined in Prelude
Step 2: Sort words

```
["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

\[
[["again"], ["never", "never"], ["say"]]
\]

\[
\downarrow 
\text{map (} \lambda \text{ws} \rightarrow (\text{head ws, length ws}) \)
\]

\[
[(["again", 1), ("never", 2), ("say", 1)]
\]
Step 5: Format each group

\[
[\text{("again", 1), ("never", 2), ("say", 1)}] \\
\downarrow \\
\text{map } (\lambda (w, n) \to (w ++ ": " ++ show n))
\]

\[
[\text{"again: 1", "never: 2", "say: 1"}]
\]
Step 6: Combine the lines

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

```
function unlines

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

Predefined in Prelude
countWords :: String -> String
countWords =
  unlines
  . map (\(w, n) -> w ++ " : " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
Can we merge two consecutive maps?

\[ \text{map } f \ . \ \text{map } g = ??? \]
The optimized solution

countWords :: String -> String
countWords =
  unlines
    . map (\ws -> head ws ++ " : " ++ show(length ws))
    . group
    . sort
    . words
Proving \( \text{map } f \ . \ \text{map } g = \text{map } (f \cdot g) \)

First we prove (why?)

\[
\text{map } f \ (\text{map } g \ \text{xs}) = \text{map } (f \cdot g) \ \text{xs}
\]

by induction on \( \text{xs} \):

- **Base case:**
  \[
  \text{map } f \ (\text{map } g \ [] ) = [] \\
  \text{map } (f \cdot g) \ [] = []
  \]

- **Induction step:**
  \[
  \text{map } f \ (\text{map } g \ (x:xs)) \\
  = f \ (g \ x) : \text{map } f \ (\text{map } g \ \text{xs}) \\
  = f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs} \quad \text{-- by IH} \\
  \text{map } (f \cdot g) \ (x:xs) \\
  = f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs}
  \]

\[
\Rightarrow (\text{map } f \ . \ \text{map } g) \ \text{xs} = \text{map } f \ (\text{map } g \ \text{xs}) = \text{map } (f \cdot g) \ \text{xs}
\]

\[
\Rightarrow (\text{map } f \ . \ \text{map } g) = \text{map } (f \cdot g) \quad \text{by extensionality}
\]
7. Type Classes
Remember: type classes enable overloading

**Example**

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x = any (== x)
where Eq is the class of all types with ==
```
In general:

*Type classes are collections of types that implement some fixed set of functions*

Haskell type classes are analogous to Java interfaces: a set of function names with their types

**Example**

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

Note: the type of (==) outside the class context is `Eq a => a -> a -> Bool`
The general form of a class declaration:

```haskell
class C a where
  f1 :: T1
  ...
  fn :: Tn
```

where the $T_i$ may involve the type variable $a$
Instance

A type $T$ is an *instance* of a class $C$ if $T$ supports all the functions of $C$. Then we write $C T$.

Example
Type `Int` is an instance of class `Eq`, i.e., `Eq Int` Therefore `elem :: Int -> [Int] -> Bool`

Warning Terminology clash:
Type $T_1$ is an *instance* of type $T_2$ if $T_1$ is the result of replacing type variables in $T_2$. For example `(Bool,Int)` is an instance of `(a,b)`.
The `instance` statement makes a type an instance of a class.

Example

```haskell
instance Eq Bool where
  True  ==  True    =  True
  False ==  False   =  True
  _     ==     _    =  False
```
Instances can be constrained:

**Example**

```haskell
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x == y && xs == ys
    _      == _      = False
```

Possibly with multiple constraints:

**Example**

```haskell
instance (Eq a, Eq b) => Eq (a,b) where
    (x1,y1) == (x2,y2) =  x1 == x2 && y1 == y2
```
The general form of the instance statement:

```
instance (context) => C T where
  definitions
```

- \( T \) is a type
- `context` is a list of assumptions \( C_i T_i \)
- `definitions` are definitions of the functions of class \( C \)
Subclasses

Example

class Eq a => Ord a where
    (<=), (<) :: a -> a -> Bool

Class \texttt{Ord} inherits all the operations of class \texttt{Eq}

Because \texttt{Bool} is already an instance of \texttt{Eq},
we can now make it an instance of \texttt{Ord}:

\texttt{instance Ord Bool where}
    \texttt{b1 \(\leq\) b2 = not b1 \(\mid\mid\) b2}
    \texttt{b1 < b2 = b1 \(\leq\) b2 \&\& not (b1 == b2)}
From the Prelude: Eq, Ord, Show

class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y = not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y = x <= y && x /= y
  x > y = y < x
  x >= y = y <= x

class Show a where
  show :: a -> String
8. Algebraic \textbf{data} Types

\begin{itemize}
  \item data by example
  \item The general case
  \item Case study: boolean formulas
  \item Structural induction
\end{itemize}
So far: no really new types,
just compositions of existing types

Example: type String = [Char]

Now: data defines new types

Introduction by example: From enumerated types to recursive and polymorphic types
8.1 data by example
From the Prelude:

```haskell
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
```
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, Read, ...
Warning
Do not forget to make your data types instances of Show

Otherwise Haskell cannot even print values of your type

Warning
QuickCheck does not automatically work for data types

You have to write your own test data generator. Later.
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 | Rect Width Height
    deriving (Eq, Show)

Some values of type Shape:  Circle 1.0
                             Rect 0.9 1.1
                             Circle (-2.0)

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

From the Prelude:

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

Some values of type Maybe:
- Nothing :: Maybe a
- Just True :: Maybe Bool
- Just "?" :: Maybe String

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

```haskell
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
```

The result of deriving `Eq`:

```
instance Eq a => Eq [a] where
    []       == []       =  True
    (x:xs)   == (y:ys)   =  x == y && xs == ys
    _        == _        =  False
```

Defined explicitly:

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

211
import Control.Monad
import Test.QuickCheck

-- for QuickCheck: test data generator for Trees
instance Arbitrary a => Arbitrary (Tree a) where
    arbitrary = sized tree
    where
        tree 0 = return Empty
        tree n | n > 0 =
            oneof [return Empty,
                liftM3 Node arbitrary (tree (n `div` 2))
                (tree (n `div` 2))]

QuickCheck for Tree
prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
  find x (insert y t) == ???

(Int not optimal for QuickCheck)
Edit distance (see Thompson)

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

Example: from "fish" to "chips"

Applications: DNA Analysis, Unix diff command
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]
best :: [[Edit]] -> [Edit]
best [x] = x
best (x:xs)
    | cost x <= cost b = x
    | otherwise = b
where b = best xs

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
Example: What is the edit distance from "trittin" to "tarantino"?
transform "trittin" "tarantino" = ?

Complexity of transform: time $O( )$

The edit distance problem can be solved in time $O(mn)$ with *dynamic programming*
8.2 The general case

data $T \ a_1 \ ... \ a_p =$

\[ C_1 \ t_{11} \ ... \ t_{1k_1} \mid \]

\[ \vdots \]

\[ C_n \ t_{n1} \ ... \ t_{nk_n} \]

defines the constructors

\[ C_1 :: t_{11} \to \ ... \ t_{1k_1} \to T \ a_1 \ ... \ a_p \]

\[ \vdots \]

\[ C_n :: t_{n1} \to \ ... \ t_{nk_n} \to T \ a_1 \ ... \ a_p \]
Patterns revisited

Patterns are expressions that consist only of constructors and variables (which must not occur twice):

A pattern can be

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

\[
\text{Not(Not } T \text{)} \text{ and } T \text{ mean the same but are different:}
\]

\[
\text{Not(Not } T \text{)} /= T
\]

What is the meaning of a Form?

Its value!?

But what is the value of Var "p" ?
-- Wertebelegung

type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = the(lookup x v) where the(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
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"]
= [("b",False):v | v <- vals [[]]] ++
  [("b",True):v | v <- vals [[]]]
= [("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)]
Does vals construct all valuations?

\[
\text{prop\_vals1 } \text{xs} = \\
\quad \text{length(vals xss)} == 2 ^ \text{length xss}
\]

\[
\text{prop\_vals2 } \text{xs} = \\
\quad \text{distinct (vals xss)}
\]

\[
\text{distinct :: Eq a => [a] -> Bool} \\
\text{distinct []} = \text{True} \\
\text{distinct (x:xs)} = \text{not(elem x xs)} && \text{distinct xs}
\]

Demo
Restrict size of test cases:

prop_vals1' xs =
    length xs <= 10 ==> length(vals xs) == 2 ^ length xs

prop_vals2' xs =
    length xs <= 10 ==> distinct (vals xs)

Demo
Satisfiable and tautology

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

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

vars :: Form -> [Name]
vars F = []
vars T = []
vars (Var x) = [x]
vars (Not p) = vars p
vars (p :&: q) = nub (vars p ++ vars q)
vars (p :+: q) = nub (vars p ++ vars q)
p0 :: Form
p0 = (Var "a" &: Var "b") |:
      (Not (Var "a") &: Not (Var "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
Simplifying a formula: Not inside?

```haskell
isSimple :: Form -> Bool
isSimple (Not p) = not (isOp p)
    where
        isOp (Not p) = True
        isOp (p :&: q) = True
        isOp (p :|: q) = True
        isOp p = False
isSimple (p :&: q) = isSimple p && isSimple q
isSimple (p :|: q) = isSimple p && isSimple q
isSimple p = True
```
Simplifying a formula: Not inside!

simplify :: Form -> Form
simplify (Not p) = pushNot (simplify p)
where
    pushNot (Not p) = p
    pushNot (p :&: q) = pushNot p :|: pushNot q
    pushNot (p :|: q) = pushNot p :&: pushNot q
    pushNot p = Not p
simplify (p :&: q) = simplify q :&: simplify q
simplify (p :|: q) = simplify p :|: simplify q
simplify p = p
-- for QuickCheck: test data generator for Form
instance Arbitrary Form where
  arbitrary = sized prop
    where
      prop 0   =
        oneof [return F,
                return T,
                liftM Var arbitrary]
      prop n | n > 0 =
        oneof
          [return F,
           return T,
           liftM Var arbitrary,
           liftM Not (prop (n-1)),
           liftM2 (:&:) (prop(n 'div' 2)) (prop(n 'div' 2)),
           liftM2 (:&:) (prop(n 'div' 2)) (prop(n 'div' 2))]

prop_simplify p = isSimple(simplify p)
8.4 Structural induction
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 $P(t2)$.
    ($x$, $t1$ and $t2$ are new variables)
Example

```hs
flat :: Tree a -> [a]
flat Empty = []
flat (Node x t1 t2) =
    flat t1 ++ [x] ++ flat 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 \( \text{flat} \ (\text{mapTree} \ f \ t) = \text{map} \ f \ (\text{flat} \ t) \)

Proof by structural induction on \( t \)

Induction step:

IH1: \( \text{flat} \ (\text{mapTree} \ f \ t1) = \text{map} \ f \ (\text{flat} \ t1) \)

IH2: \( \text{flat} \ (\text{mapTree} \ f \ t2) = \text{map} \ f \ (\text{flat} \ t2) \)

To show: \( \text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2)) = \text{map} \ f \ (\text{flat} \ (\text{Node} \ x \ t1 \ t2)) \)

\[
\begin{align*}
\text{flat} \ (\text{mapTree} \ f \ (\text{Node} \ x \ t1 \ t2)) &= \text{flat} \ (\text{Node} \ (f \ x) \ (\text{mapTree} \ f \ t1) \ (\text{mapTree} \ f \ t2)) \\
&= \text{flat} \ (\text{mapTree} \ f \ t1) ++ [f \ x] ++ \text{flat} \ (\text{mapTree} \ f \ t2) \\
&= \text{map} \ f \ (\text{flat} \ t1) ++ [f \ x] ++ \text{map} \ f \ (\text{flat} \ t2) \\
&= \text{map} \ f \ (\text{flat} \ t1 + x) \text{flat} \ t2 \\
&= \text{map} \ f \ (\text{flat} \ t1 + x) \text{map} \ f \ (\text{flat} \ t2) \\
&= \text{map} \ f \ (\text{flat} \ t1) ++ [f \ x] ++ \text{map} \ f \ (\text{flat} \ t2) \\
&= \text{map} \ f \ (\text{flat} \ t1 + x) \text{flat} \ t2 \\
\end{align*}
\]

Note: Base case and -- by def of ... omitted
data T a = ... 

Assumption: T is a regular data type:

Each constructor $C_i$ of T must have a type $t_1 \rightarrow \ldots \rightarrow t_{n_i} \rightarrow T a$
such that each $t_j$ is either $T a$ or does not contain T

To prove property $P(t)$ for all finite $t :: T a$:
prove for each constructor $C_i$ that $P(C_i x_1 \ldots x_{n_i})$
assuming the induction hypotheses $P(x_j)$ for all $j$ s.t. $t_j = T a$

Example of non-regular type: data T = C [T]
9. Modules and Abstract Data Types

- Modules
- Abstract Data Types
- Correctness
9.1 Modules

Module = collection of type, function, class etc definitions

Purposes:
- Grouping
- Interfaces
- Division of labour
- Name space management: $M.f$ vs $f$
- Information hiding

GHC: one module per file
Recommendation: module $M$ in file $M.hs$
module M where -- M must start with capital letter

↑
All definitions must start in this column

• Exports everything defined in M (at the top level)

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

Not permitted: module M (T,C,D) where (why?)
Exporting modules

By default, modules do not export names from imported modules

```
module B where
import A
...
⇒ B does not export f
```

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
...
```
By default, everything that is exported is imported

```
module B where
import A
...

⇒ B imports f and g
```

Unless an import list is specified

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

⇒ B imports only f
```

Or specific names are hidden

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

Where does f come from??

Clearer: qualified names

... A.f ...

Can be enforced:

import qualified A

⇒ must always write A.f
Renaming modules

import 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
9.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

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

...
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: $S \rightsquigarrow \text{Set}$

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

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
What if Set is already a data type?

module SetByTree (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
Beware of ==

module SetByTree (Set, empty, insert, isin, size) where
...

type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
deriving (Eq)
...

Class instances are automatically exported and cannot be hidden

Client module:

import SetByTree
... insert 2 (insert 1 empty) ==
    insert 1 (insert 2 empty)
...

Result is probably False — representation is partly exposed!
The proper treatment of ==

Some alternatives:

- Do not make Tree an instance of Eq
- Hide representation:
  ```haskell
  -- do not export constructor Set:
  newtype Set a = Set (Tree a)
  data Tree a = Empty | Node a (Tree a) (Tree a)
  deriving (Eq)
  ``

- Define the right == on Tree:
  ```haskell
  instance Eq a => Eq(Tree a) where
  t1 == t2 = elems t1 == elems t2
  where
  elems Empty = []
  elems (Node x t1 t2) = elems t1 ++ [x] ++ elems t2
  ```
Similar for all class instances, not just Eq
9.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because `empty` simulates `{}`
and `insert _ _` simulates `{_} ∪ _`
and `isin _ _` simulates `_ ∈ _`
and `size _` simulates `|_|

Each concrete operation on the implementation type of lists simulates its abstract counterpart on sets

NB: We relate Haskell to mathematics
For uniformity we write `{a}` for the type of finite sets over type `a`
From lists to sets

Each list \([x_1, \ldots, x_n]\) represents the set \(\{x_1, \ldots, x_n\}\).

**Abstraction function** \(\alpha :: [a] \rightarrow \{a\}\)

\[\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}\]

In Haskell style:

\[\alpha [] = \{}\]
\[\alpha (x:xs) = \{x\} \cup \alpha xs\]

What does it mean that “lists simulate (implement) sets”:

\(\alpha\) (concrete operation) = abstract operation

\(\alpha\) empty = \{

\(\alpha\) (insert x xs) = \{x\} \cup \alpha xs

isin x xs = x \in \alpha xs

size xs = |\alpha xs|
For the mathematically inclined:

\( \alpha \) must be a homomorphism
Implementation I: lists with duplicates

empty = []
insert x xs = x : xs
isin x xs = elem x xs
size xs = length(nub xs)

The simulation requirements:

\[ \alpha \text{ empty} = \{\} \]
\[ \alpha (\text{insert } x \xs) = \{x\} \cup \alpha \xs \]
\[ \text{isin } x \xs = x \in \alpha \xs \]
\[ \text{size } \xs = |\alpha \xs| \]

Two proofs immediate, two need lemmas proved by induction
Implementation II: lists without duplicates

empty = []
insert x xs = if elem x xs then xs else x:xs
isin x xs = elem x xs
size xs = length xs

The simulation requirements:

\[ \alpha \text{ empty} = \{\} \]
\[ \alpha (\text{insert } x \text{ xs}) = \{x\} \cup \alpha \text{ xs} \]
\[ \text{isin } x \text{ xs} = x \in \alpha \text{ xs} \]
\[ \text{size } xs = |\alpha \text{ xs}| \]

Needs \textit{invariant} that \( xs \) contains no duplicates

\textbf{invar} :: [a] -> Bool
\textbf{invar} [] = True
\textbf{invar} (x:xs) = not(elem x xs) && \textbf{invar} xs
Implementation II: lists without duplicates

empty = []
insert x xs = if elem x xs then xs else x:xs
isin x xs = elem x xs
size xs = length xs

Revised simulation requirements:

\[
\begin{align*}
\alpha \quad \text{empty} &= \{\}\quad \Rightarrow \\
\text{invar } xs &\quad \Rightarrow \quad \alpha (\text{insert } x \; xs) = \{x\} \cup \alpha \; xs \\
\text{invar } xs &\quad \Rightarrow \quad \text{isin } x \; xs = x \in \alpha \; xs \\
\text{invar } xs &\quad \Rightarrow \quad \text{size } xs = |\alpha \; xs|
\end{align*}
\]

Proofs omitted. Anything else?
invar must be invariant!

In an imperative context:

If \texttt{invar} is true before an operation,
it must also be true after the operation

In a functional context:

If \texttt{invar} is true for the arguments of an operation,
it must also be true for the result of the operation

\texttt{invar} is \textit{preserved} by every operation

\texttt{invar \hspace{2pt} empty}

\texttt{invar \hspace{2pt} xs \implies \hspace{2pt} invar \hspace{2pt} (insert \hspace{2pt} x \hspace{2pt} xs)}

Proofs do not even need induction
Let $C$ and $A$ be two modules that have the same interface:
a type $T$ and a set of functions $F$

To prove that $C$ is a correct implementation of $A$ define
an abstraction function $\alpha :: C.T \to A.T$
and an invariant $\text{invar} :: C.T \to \text{Bool}$

and prove for each $f \in F$:

- $\text{invar}$ is invariant:
  \[
  \text{invar } x_1 \land \cdots \land \text{invar } x_n \implies \text{invar} (C.f x_1 \ldots x_n)
  \]
  (where $\text{invar}$ is True on types other than $C.T$)

- $C.f$ simulates $A.f$:
  \[
  \text{invar } x_1 \land \cdots \land \text{invar } x_n \implies \\
  \alpha (C.f x_1 \ldots x_n) = A.f (\alpha x_1) \ldots (\alpha x_n)
  \]
  (where $\alpha$ is the identity on types other than $C.T$)
10. Case Study: Huffman Coding
See Thompson, blackboard and the source files on the web page
11. Case Study: Parsing
See blackboard and the source files on the web page
12. Lazy evaluation

Applications of lazy evaluation
Infinite lists
Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

\emph{lazy evaluation} („verzögerte Auswertung“)

Advantages:

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

Therefore Haskell is called a \emph{lazy functional language}. Haskell is the only mainstream lazy functional language.
Evaluating expressions

Expressions are evaluated \((\textit{reduced})\) by successively applying definitions until no further reduction is possible.

Example:

\[
\text{sq} :: \text{Integer} \to \text{Integer}
\]

\[
\text{sq} \ n = n \times n
\]

One evaluation:

\[
\text{sq}(3+4) = \text{sq} \ 7 = 7 \times 7 = 49
\]

Another evaluation:

\[
\text{sq}(3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49
\]
Theorem
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
Let \( n \) have value 0 initially.

Two evaluations:

\[
\begin{align*}
\text{n + (n := 1)} &= 0 + (\text{n := 1}) = 0 + 1 = 1 \\
\text{n + (n := 1)} &= \text{n + 1} = 1 + 1 = 2
\end{align*}
\]
Reduction strategies

An expression may have many reducible subexpressions:

\[ \text{sq} \ (3+4) \]

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
\[ \text{sq} \ (3+4) = \text{sq} \ 7 = 7 * 7 \]

**Outermost reduction**  Always reduce an outermost redex.
Corresponds to *call by name*:
The unevaluated arguments
are substituted into the function body
\[ \text{sq} \ (3+4) = (3+4) * (3+4) \]
Comparison: termination

Definition:
loop = tail loop

Innermost reduction:
\[ \text{fst} (1, \text{loop}) = \text{fst}(1, \text{tail loop}) = \text{fst}(1, \text{tail(tail loop)}) = \ldots \]

Outermost reduction:
\[ \text{fst} (1, \text{loop}) = 1 \]

**Theorem** If expression \( e \) has a terminating reduction sequence, then outermost reduction of \( e \) also terminates.

**Outermost reduction terminates as often as possible**
Why is this useful?

Example
Can build your own control constructs:

```haskell
switch :: Int -> a -> a -> a
switch n x y
    | n > 0       = x
    | otherwise   = y

fac :: Int -> Int
fac n = switch n (n * fac(n-1)) 1
```
Comparison: Number of steps

Innermost reduction:

\[ \text{sq}(3+4) = \text{sq} 7 = 7 \times 7 = 49 \]

Outermost reduction:

\[ \text{sq}(3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49 \]

More outermost than innermost steps!

How can outermost reduction be improved?

Sharing!
\[ sq(3+4) = \bullet \times \bullet = \bullet \times \bullet = 49 \]

The expression 3+4 is only evaluated \textit{once}!

Lazy evaluation := outermost reduction + sharing

\textbf{Theorem}
Lazy evaluation never needs more steps than innermost reduction.
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

\[ f :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int} \]
\[ f \; [] \; ys \; = \; 0 \]
\[ f \; (x:xs) \; [] \; = \; 0 \]
\[ f \; (x:xs) \; (y:ys) \; = \; x+y \]

Lazy evaluation:
\[ f \; [1..3] \; [7..9] \quad -- \text{does f.1 match?} \]
\[ = \; f \; (1 : [2..3]) \; [7..9] \quad -- \text{does f.2 match?} \]
\[ = \; f \; (1 : [2..3]) \; (7 : [8..9]) \quad -- \text{does f.3 match?} \]
\[ = \; 1+7 \]
\[ = \; 8 \]
Guards

Example

\[
f \, m \, n \, p \mid m \geq n \land m \geq p = m \\
| n \geq m \land n \geq p = n \\
| \text{otherwise} = p
\]

Lazy evaluation:
\[
f \,(2+3)\,(4-1)\,(3+9)
\]
\[
? \quad 2+3 \geq 4-1 \land 2+3 \geq 3+9
\]
\[
? \quad = \quad 5 \geq 3 \land 5 \geq 3+9
\]
\[
? \quad = \quad \text{True} \land 5 \geq 3+9
\]
\[
? \quad = \quad 5 \geq 3+9
\]
\[
? \quad = \quad 5 \geq 12
\]
\[
? \quad = \quad \text{False}
\]
\[
? \quad 3 \geq 5 \land 3 \geq 12
\]
\[
? \quad = \quad \text{False} \land 3 \geq 12
\]
\[
? \quad = \quad \text{False}
\]
\[
? \quad \text{otherwise} = \quad \text{True}
\]
\[
= \quad 12
\]
Same principle: definitions in `where` clauses are only evaluated when needed and only as much as needed.
Haskell never reduces inside a lambda

Example: \( \lambda x \rightarrow \text{False} \&\& x \) cannot be reduced

Reasons:

- Functions are black boxes
- All you can do with a function is apply it

Example:

\[ (\lambda x \rightarrow \text{False} \&\& x) \text{True} = \text{False} \&\& \text{True} = \text{False} \]
Predefined functions

They behave like their Haskell definition (if they have one):

\[
(\&\&) :: \text{Bool} \to \text{Bool} \to \text{Bool}
\]

\[
\text{True} \&\& y = y
\]

\[
\text{False} \&\& y = \text{False}
\]

Or they evaluate their arguments first: basic arithmetic
Lazy evaluation evaluates an expression only when needed and only as much as needed.

("Call by need")
12.1 Applications of lazy evaluation
The minimum of a list

\[
\text{min} = \text{head} \ . \ \text{inSort}
\]

\[
\text{inSort} :: \text{Ord a} \Rightarrow [a] \rightarrow [a]
\]
\[
\text{inSort} [] = []
\]
\[
\text{inSort} (x:xs) = \text{ins} x (\text{inSort} \ xs)
\]

\[
\text{ins} :: \text{Ord a} \Rightarrow a \rightarrow [a] \rightarrow [a]
\]
\[
\text{ins} x [] = [x]
\]
\[
\text{ins} x (y:ys) | x \leq y = x : y : ys
\]
\[
| \text{otherwise} = y : \text{ins} x \ ys
\]

\[
\implies \text{inSort} [6,1,7,5]
\]
\[
= \text{ins} 6 (\text{ins} 1 (\text{ins} 7 (\text{ins} 5 [])))
\]
\[
\text{min } [6,1,7,5] = \text{head(inSort } [6,1,7,5])
\]
\[
= \text{head}(\text{ins } 6 \ (\text{ins } 1 \ (\text{ins } 7 \ (\text{ins } 5 \ []))))
\]
\[
= \text{head}(\text{ins } 6 \ (\text{ins } 1 \ (\text{ins } 7 \ (5 : []))))
\]
\[
= \text{head}(\text{ins } 6 \ (\text{ins } 1 \ (5 : \text{ins } 7 [])))
\]
\[
= \text{head}(\text{ins } 6 \ (1 : 5 : \text{ins } 7 []))
\]
\[
= \text{head}(1 : \text{ins } 6 \ (5 : \text{ins } 7 []))
\]
\[
= 1
\]

Lazy evaluation needs only linear time although \text{inSort} is quadratic because the sorted list is never constructed completely

Warning: this depends on the exact algorithm and does not work so nicely with all sorting functions!
type Parser a b = [a] -> Maybe (b, [a])

⇝

type Parser a b = [a] -> [(b, [a])]
\[ p1 \parallel p2 = \lambda as \rightarrow \text{case } p1 \text{ of} \]
\[ \text{Nothing} \rightarrow p2 as \]
\[ \text{just} \rightarrow \text{just} \]

\[ \Rightarrow \]

\[ p1 \parallel p2 = \lambda xs \rightarrow p1 xs ++ p2 xs \]
\[ p_1 \circledast p_2 = \lambda \text{xs} \rightarrow \]

\[
\text{case } \text{p1} \ \text{xs} \ \text{of}
\]

\[
\text{Nothing } \rightarrow \text{Nothing}
\]

\[
\text{Just}(b,ys) \rightarrow \text{case } \text{p2} \ \text{ys} \ \text{of}
\]

\[
\text{Nothing } \rightarrow \text{Nothing}
\]

\[
\text{Just}(c,zs) \rightarrow \text{Just}((b,c),zs)
\]

\[\Rightarrow\]

\[ p_1 \circledast p_2 = \lambda \text{xs} \rightarrow \]

\[
[ ((b,c),zs) \mid (b,ys) \leftarrow \text{p1} \ \text{xs}, (c,zs) \leftarrow \text{p2} \ \text{ys}]\]
\[
p >>> f = \backslash xs \rightarrow \\
\text{case } p \text{ xs of} \\
\text{Nothing } \rightarrow \text{Nothing} \\
\text{Just}(b, ys) \rightarrow \text{Just}(f \ b, ys)
\]

\[\Rightarrow\]

\[p >>> f = [(f \ b, ys) | (b, ys) \leftarrow p \ xs]\]
12.2 Infinite lists
Example
A recursive definition
ones :: [Int]
ones = 1 : ones
that 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,
Haskell lists can be finite or infinite
Printing an infinite list does not terminate

294

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

> head ones
1

Remember:

Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: \[ \text{head ones} = \text{head (1 : ones)} = 1 \]

Innermost reduction:

\[
\begin{align*}
\text{head ones} &= \text{head (1 : ones)} \\
&= \text{head (1 : 1 : ones)} \\
&= \ldots
\end{align*}
\]
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 : 2 : 3 : \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 :: [Int]
primes = sieve [2..]

sieve :: [Int] -> [Int]
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]
= ...
```
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,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],...
Primality test?

> 101 ‘elem‘ primes
 True

> 102 ‘elem‘ primes
 nontermination

prime n = n == head (dropWhile (<n) primes)
13. I/O and Monads

I/O

Monads
13.1 I/O

- So far, only batch programs: given the full input at the beginning, the full output is produced at the end
- Now, interactive programs: read input and write output while the program is running
The problem

- Haskell programs are pure mathematical functions:
  Haskell programs have no side effects

- Reading and writing are side effects:
  Interactive programs have side effects
Most languages allow functions to perform I/O without reflecting it in their type.

Assume that Haskell were to provide an input function

\[
\text{inputInt} :: \text{Int}
\]

Now all functions are potentially perform side effects.

Now we can no longer reason about Haskell like in mathematics:

\[
\text{inputInt} - \text{inputInt} = 0 \\
\text{inputInt} + \text{inputInt} = 2\times\text{inputInt}
\]

\[\ldots\]

are no longer true.
Haskell distinguishes expressions without side effects from expressions with side effects (*actions*) by their type:

\[ \text{IO} \ a \]

is the type of (I/O) actions that return a value of type \( a \).

**Example**

- **Char**: the type of pure expressions that return a Char
- **IO Char**: the type of actions that return an Char
- **IO ()**: the type of actions that return no result value
• Type () is the type of empty tuples (no fields).
• The only value of type () is (), the empty tuple.
• Therefore IO () is the type of actions that return the dummy value () (because every action must return some value)
Basic actions

- **getChar :: IO Char**
  
  Reads a Char from standard input, echoes it to standard output, and returns it as the result

- **putChar :: Char -> IO ()**
  
  Writes a Char to standard output, and returns no result

- **return :: a -> IO a**
  
  Performs no action, just returns the given value as a result
A sequence of actions can be combined into a single action with the keyword \texttt{do}.

Example

\begin{verbatim}
get2 :: IO (Char,Char)
get2 = do x <- getChar  -- result is named x
       getChar
       y <- getChar  -- result is ignored
       return (x,y)
\end{verbatim}
General format (observe layout!):

do  \ a_1 \\
   \vdots \\
   \ a_n \\

where each \( a_i \) can be one of

- an action
  Effect: execute action
- \( x <- \ action \)
  Effect: execute \( action \), give result the name \( x \)
- \( \text{let } x = expr \)
  Effect: give \( expr \) the name \( x \)
  Lazy: \( expr \) is only evaluated when \( x \) is needed!
Derived primitives

Write a string to standard output:

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (c:cs) = do putChar c
                  putStr cs
```

Write a line to standard output:

```haskell
putStrLn :: IO ()
putStrLn cs = putStr (cs ++ '\n')
```
Read a line from standard input:

\[
\text{getLine :: IO String} \\
\text{getLine = do } x \leftarrow \text{getChar} \\
\hspace{1em} \text{if } x == '\\n' \text{ then} \\
\hspace{2em} \text{return } [] \\
\hspace{1em} \text{else} \\
\hspace{2em} \text{do } xs \leftarrow \text{getLine} \\
\hspace{3em} \text{return } (x:xs)
\]

Actions are normal Haskell values and can be combined as usual, for example with if-then-else.
Example

Prompt for a string and display its length:

```haskell
strLen :: IO ()
strLen = do
  putStrLn "Enter a string: ":
  xs <- getLine
  putStrLn "The string has ":
  putStrLn (show (length xs))
  putStrLn " characters"

> strLen

Enter a string: abc
The string has 3 characters
How to read other types

Input string and convert

Useful class:

```haskell
class Read a where
    read :: String -> a
```

Most predefined types are in class Read.

Example:

```haskell
getInt :: IO Integer
getInt = do xs <- getLine
            return (read xs)
```
So far implicit: read from stdin :: Handle
write to stdout :: Handle

data Handle

Haskell defines operations to read and write characters from and to files, represented by values of type Handle. Each value of this type is a handle: a record used by the Haskell run-time system to manage I/O with file system objects.

Details: Haskell IO library
Case study

The game of Hangman
in file Hang.hs
Once IO, always IO

You cannot add I/O to a function without “polluting” its type

For example

\[
\begin{align*}
\text{sq} & \ :: \ \text{Int} \rightarrow \ \text{Int} \\
\text{sq} \ x &= x \times x \\
\text{cube} & \ :: \ \text{Int} \rightarrow \ \text{Int} \\
\text{cube} \ x &= x \times \text{sq} \ x
\end{align*}
\]

Let us try to make \text{sq} print out some message:

\[
\begin{align*}
\text{sq} \ x &= \text{do} \ \text{putStrLn}("I am in sq!\") \\
& \quad \text{return}(x \times x)
\end{align*}
\]

What is the type of \text{sq} now? \textbf{Int} \rightarrow \textbf{IO} \textbf{Int}

And this is what happens to \text{cube}:

\[
\begin{align*}
\text{cube} \ x &= \text{do} \ x2 \leftarrow \text{sq} \ x \\
& \quad \text{return}(x \times x2)
\end{align*}
\]
Haskell is a pure functional language
Functions that have side effects must show this in their type
I/O is a side effect
13.2 Monads
 >>= (‘bind’), or what do really means

Primitive:

\[(\gg\gg) : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b\]

How it works:

\[\text{act } \gg\gg\ f\]

execute action \(\text{act} :: \text{IO } a\)

which returns a result \(v :: a\)

then evaluate \(f \ v\)

which returns a result of type \(\text{IO } b\)

\[
\text{do } x \leftarrow \text{act}_1 \\
\quad \text{act}_2
\]

is syntax for \(\text{act}_1 \gg\gg (\backslash x \rightarrow \text{act}_2)\)

Example

\[
\text{do } x \leftarrow \text{getChar} \\
\quad \text{putChar } x \quad \rightsquigarrow \quad \text{getChar} \gg\gg (\backslash x \rightarrow \text{putChar } x)
\]
In general

\[
\begin{align*}
d & x_1 \leftarrow a_1 \\
& \vdots \\
& x_n \leftarrow a_n \\
act
\end{align*}
\]

is syntax for

\[
\begin{align*}
a_1 & \triangleright= \backslash x_1 \rightarrow \\
& \vdots \\
a_n & \triangleright= \backslash x_n \rightarrow \\
act
\end{align*}
\]
Beyond IO: Monads

class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

- m is a type constructor
- do notation is defined for every monad

Only example of monad so far: IO
Let's examine some more.
Maybe as a monad

A frequent code pattern when working with Maybe:

```haskell
    case m of
        Nothing -> Nothing
        Just x -> ...
```

This pattern can be hidden inside `>>=:`

```haskell
instance Monad Maybe where
    m >>= f = case m of
        Nothing -> Nothing
        Just x -> f x
    return v = Just v
```

Failure (= Nothing) propagation and unwrapping of Just is now built into do!
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x -> f x
  return v = Just v

Example: evaluation of Form

eval :: [(Name,Bool)] -> Form -> Maybe Bool
eval _ T = return True
eval _ F = return False
eval v (Var x) = lookup x v
eval v (f1 &: f2) = do
  b1 <- eval v f1
  b2 <- eval v f2
  return (b1 && b2)
...

Example:

\[ p_1 *** p_2 = \lambda xs \rightarrow \]
\[ \text{case } p_1 \; xs \; \text{of} \]
\[ \quad \text{Nothing } \rightarrow \text{Nothing} \]
\[ \quad \text{Just}(b,ys) \rightarrow \text{case } p_2 \; ys \; \text{of} \]
\[ \quad \quad \text{Nothing } \rightarrow \text{Nothing} \]
\[ \quad \quad \text{Just}(c,zs) \rightarrow \text{Just}((b,c),zs) \]

\[ \Rightarrow \]

\[ p_1 *** p_2 = \lambda xs \rightarrow \]
\[ \text{do } (b,ys) \leftarrow p_1 \; xs \]
\[ \quad (c,zs) \leftarrow p_2 \; ys \]
\[ \quad \text{return } ((b,c),zs) \]

The do version has a much more general type \( \text{Monad } m \rightarrow \ldots \)
Maybe models possible failure with Just/Nothing

The do of the Maybe monad hides Just/Nothing and propagates failure automatically
List as a monad

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return v = [v]
```

Now we can compose computations on list nicely (via do).

**Example**

```haskell
dfs :: (a -> [a]) -> (a -> Bool) -> a -> [a]
dfs nexts found start = find start
  where
    find x = if found x then return x
   else do x’ <- nexts x
            find x’
```

The Haskell way of backtracking
Lazy evaluation produces only as many elements as you ask for.
14. Complexity and Optimization

Time complexity analysis
Optimizing functional programs
How to analyze and improve the time (and space) complexity of functional programs

Based largely on Richard Bird’s book
*Introduction to Functional Programming using Haskell.*

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier
- Most languages follow cbv
- Number of lazy evaluation steps \( \leq \) number of cbv steps
  \[ \Rightarrow \] \( O \)-analysis under cbv also correct for Haskell
  but can be too pessimistic
14.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation, if-then-else on the right-hand side instead)

Justification:

The implementation does not copy data structures but works with pointers and sharing

Example: \( \text{length } (_\ : \ xs) = \text{length } xs + 1 \)
Reduce \( \text{length } [1,2,3] \)

Compare: \( \text{id } [] = [] \)
\( \text{id } (x:x) = x : \text{id } xs \)
Reduce \( \text{id } [e1,e2] \)
Copies list but shares elements.
$T_f(n) =$ number of steps required for the evaluation of $f$
when applied to an argument of size $n$
in the worst case

What is “size”?  

- Number of bits. Too low level.  
- Better: specific measure based on the argument type of $f$  
- Measure may differ from function to function.  
- Frequent measure for functions on lists: the length of the list  
  We use this measure unless stated otherwise  
  Sufficient if $f$ does not compute with the elements of the list  
  Not sufficient for function ...
How to calculate (not mechanically!) $T_f(n)$:

1. From the equations for $f$ derive equations for $T_f$
2. If the equations for $T_f$ are recursive, solve them
Example

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x : (xs ++ ys)
\end{align*}
\]

\[
\begin{align*}
T_{++}(0, n) &= O(1) \\
T_{++}(m + 1, n) &= T_{++}(m, n) + O(1)
\end{align*}
\]

\[\Rightarrow T_{++}(m, n) = O(m)\]

Note: \((++\) creates copy of first argument

Principle:

Every constructor of an algebraic data type takes time \(O(1)\).
A constant amount of space needs to be allocated.
Example

\[
\text{reverse} \; [] \; = \; [] \\
\text{reverse} \; (x:x:s) \; = \; \text{reverse} \; s \; ++ \; [x]
\]

\[
T_{\text{reverse}}(0) \; = \; O(1) \\
T_{\text{reverse}}(n + 1) \; = \; T_{\text{reverse}}(n) + T_{++}(n, 1)
\]

\[\implies T_{++}(n) = O(n^2)\]

Observation:

Complexity analysis may need functional properties of the algorithm
The worst case time complexity of an expression $e$:

Sum up all $T_f(n_1, ..., n_k)$
where $f \ e_1 \ e_n$ is a function call in $e$
and $n_i$ is the size of $e_i$

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$, both for cbv and lazy evaluation. (Why?)

Consider $\text{min} \ \text{xs} = \text{head}(\text{sort} \ \text{xs})$

$$T_{\text{min}}(n) = T_{\text{sort}}(n) + T_{\text{head}}(n)$$

For cbv also a lower bound, but not for lazy evaluation.

Complexity analysis is *compositional* under cbv
14.2 Optimizing functional programs

Premature optimization is the root of all evil
Don Knuth

But we are in week \( n - 1 \) now ;-

The ideal of program optimization:

1. Write (possibly) inefficient but correct code
2. Optimize your code \textit{and prove equivalence to correct version}
Tail recursion / Endrekursion

The definition of a function $f$ is tail recursive / endrekursiv if every recursive call is in “end position”,
- it is the last function call before leaving $f$,
- nothing happens afterwards
- no call of $f$ is not nested in another function call

Example

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (x:xs) &= \text{length } xs + 1 \\
\text{length2 } [] n &= n \\
\text{length2 } (x:xs) n &= \text{length2 } xs (n+1)
\end{align*}
\]
length [] = 0
length (x:xs) = length xs + 1

length2 [] n = n
length2 (x:xs) n = length2 xs (n+1)

Compare executions:

length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c] 2
= length2 [] 3
= 3
Fact  Tail recursive definitions can be compiled into loops. Not just in functional languages.

No (additional) stack space is needed to execute tail recursive functions

Example

length2 [] n = n
length2 (x:xs) n = length2 xs (n+1)
⇝
loop: if null xs then return n
    xs := tail xs
    n := n+1
    goto loop
Tail recursion / Endrekursion

The definition of a function \( f \) is tail recursive / endrekursiv if every recursive call is in “end position”,
\( = \) it is the last function call before leaving \( f \),
\( = \) nothing happens afterwards
\( = \) no call of \( f \) is not nested in another function call

Example

\[
\begin{align*}
\text{length} & \; [\;] \; = \; 0 \\
\text{length} & \; (x:xs) \; = \; \text{length} \; xs \; + \; 1 \\
\text{length2} & \; [\;] \; n \; = \; n \\
\text{length2} & \; (x:xs) \; n \; = \; \text{length2} \; xs \; (n+1)
\end{align*}
\]
Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.

Purpose:

- tail recursion
- replace (++) by (:

\[
\begin{align*}
\text{length2} & \ [n \, \ n = n \\
\text{length2} & \ (x:xs) \ n = \ \text{length2} \ xs \ (n+1) \\
\end{align*}
\]

\[
\text{length'} \ xs = \ \text{length2} \ xs \ 0
\]

Correctness:

**Lemma** \( \text{length2} \ xs \ n = \ \text{length} \ xs + n \)

\( \Rightarrow \) \( \text{length'} \ xs = \ \text{length} \ xs \)
Tupling of results

Typical application:

Avoid multiple traversals of the same data structure

average :: [Float] -> Float
average xs = (sum xs) / (length xs)

Requires two traversals of the argument list.
Avoid intermediate data structures

Typical example: \( \text{map } g \cdot \text{map } f = \text{map } (g \cdot f) \)

Another example: \( \text{sum } [n..m] \)
Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Problem: lazy evaluation may leave many expressions unevaluated until the end, which requires more space

Space is time because it requires garbage collection — not counted by number of reductions!