|-------------|-------------|-------------|-------------|
Organisatorisches

Functional Programming: The Idea

Basic Haskell

Lists

Proofs

Higher-Order Functions

Type Classes

Algebraic data Types

I/O

Modules and Abstract Data Types

Case Study: Two Efficient Algorithms

Lazy evaluation

I/O and Monads

Complexity and Optimization

Case Study: Parsing
1. Organisatorisches
Siehe http://fp.in.tum.de
Literatur

- 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
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.
- Hausaufgabenstatistik:
  Wahrscheinlichkeit, die Klausur (oder W-Klausur) zu bestehen:
  - $\geq 40\%$ der Hausaufgabenpunkte $\Rightarrow 100\%$
  - $< 40\%$ der Hausaufgabenpunkte $\Rightarrow < 50\%$
- Aktueller persönlicher Punktestand im WWW über Statusseite
Programmierwettbewerb — Der Weg zum Ruhm

- Jede Woche eine Wettbewerbsaufgabe
- Punktetabellen im Internet:
  - Die Top 20 jeder Woche
  - Die kumulative Top 20
- Ende des Semesters: Trophäen für die Top $k$ Studenten
Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonym)
  Natürlich keine Lösungen posten!
- Fragen werden an alle Tutoren weitergeleitet
- Mehr über Piazza: Video auf http://piazza.com
- Zugang zu Piazza für Info 2 über Vorlesungsseite
- Funktioniert erst nach Anmeldung zur Übung
Haskell Installation

• Bei Problemen mit der Installation des GHC: Zwei Beratungstermine, siehe Vorlesungsseite (10.10. 10:00-12:00, 13.10. 10:00-13:00)
• Tutoren leisten in der Übung keine Hilfestellung mehr!
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:

\[
\text{sum \ [1..10]} \\
\]

In Java:

```java
total = 0;
for (i = 1; i <= 10; ++i)
    total = total + i;
```
In 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.
1950s

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

1970s

Robin Milner (FRS, Turing Award 1991) & Co. develop ML, the first modern functional programming language with polymorphic types and type inference.
1987

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:  

Mathematics  

\( f : A \times B \to C \)

Haskell  

\( f :: A \to B \to C \)

Function application:  

Mathematics  

\( f(a) \)

Haskell  

\( f \ a \)

\( f(a, b) \)

\( f \ a \ b \)

\( f(g(b)) \)

\( f \ (g \ b) \)

\( f(a, g(b)) \)

\( f \ a \ (g \ b) \)

Prefix binds stronger than infix:

\( f \ a + b \) means \( (f \ a) + b \)

not \( f \ (a + b) \)
3.2 Type Bool

Predefined: True False not && || ==

Defining new functions:

\[
\text{xor} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{xor} \ x \ y = (x || y) && \text{not}(x && y)
\]

\[
\text{xor2} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{xor2} \ True \ True = False \\
\text{xor2} \ True \ False = True \\
\text{xor2} \ False \ True = True \\
\text{xor2} \ False \ False = False
\]

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

Is \text{xor} \ x \ y == \text{xor2} \ x \ y \ true?
Testing with QuickCheck

Import test framework:

import Test.QuickCheck

Define property to be tested:

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

Note naming convention prop...

Check property with GHCi:

> quickCheck prop_xor2

GHCi answers

+++ OK, passed 100 tests.
QuickCheck

- Essential tool for Haskell programmers
- Invaluable for regression tests
- Important part of exercises & homework
- Helps you to avoid bugs
- Helps us to discover them

Every nontrivial Haskell function should come with one or more QuickCheck properties/tests

Typical test:

```haskell
prop_f x y =
    f_efficient x y == f_naive x y
```
V1.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:

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

Evaluation:

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

Evaluation of Haskell expressions
means
Using the defining equations from left to right.
3.4 Guarded equations

Example: maximum of 2 integers.

\[
\text{max} :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\text{max}\ x\ y \\
\quad |\ x \geq y = x \\
\quad |\ \text{otherwise} = y
\]

Haskell also has \texttt{if-then-else}:

\[
\text{max}\ x\ y = \text{if } x \geq y \text{ then } x \text{ else } y
\]

True?

\[
\text{prop\_max\_assoc}\ x\ y\ z = \\
\quad \text{max}\ x\ (\text{max}\ y\ z) =\ = \text{max}\ (\text{max}\ x\ y)\ z
\]
3.5 Recursion

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

\[
\text{-- pow x n returns x to the power of n}
\]

\[
\text{pow :: Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}
\]

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

Cannot write \( \underbrace{x \times \cdots \times x}_{n \text{ times}} \)

Two cases:

\[
\text{pow x n}
\]

\[
| \ n == 0 \ = \ 1 \quad \quad \quad \quad \quad \quad \quad -- \text{the base case}
\]

\[
| \ n > 0 \ = \ x \times \text{pow x (n-1)} \quad \quad \quad -- \text{the recursive case}
\]

More compactly:

\[
\text{pow x 0} = 1
\]

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

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

\[
\begin{align*}
\text{pow} \ 2 \ 3 & \ = \ 2 \ * \ \text{pow} \ 2 \ 2 \\
& \ = \ 2 \ * \ (2 \ * \ \text{pow} \ 2 \ 1) \\
& \ = \ 2 \ * \ (2 \ * \ (2 \ * \ \text{pow} \ 2 \ 0)) \\
& \ = \ 2 \ * \ (2 \ * \ (2 \ * \ 1)) \\
& \ = \ 8
\end{align*}
\]

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

GH Ci answers

*** Exception: PowDemo.hs:(1,1)-(2,33):
Non-exhaustive patterns in function \( \text{pow} \)
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:

\textbf{State your preconditions clearly!}

As a guard, a comment or using QuickCheck:

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

where \( \text{isDefined } y = y == y \).
Example sumTo

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

\[
\text{sumTo} :: \text{Integer} \rightarrow \text{Integer} \\
\text{sumTo} \ 0 \ = \ 0 \\
\text{sumTo} \ n \mid n > 0 \ = \\
\]

\[
\text{prop\_sumTo} \ n = \\
\quad n \geq 0 \implies \text{sumTo} \ n = n(n+1) \div 2 \\
\]

Properties can be 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:

\[ f \ x_1 \ \ldots \ x_n \]
\[ \ \mid \ test_1 = e_1 \]
\[ \quad \vdots \]
\[ \ \mid \ test_n = e_n \]

Each right-hand side \( e_i \) is an expression.
Note: \texttt{otherwise} = \texttt{True}

Function and parameter names must begin with a lower-case letter (Type names begin with an upper-case letter)
An *expression* can be

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

Additional syntactic sugar:

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

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

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

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

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

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

\[
\text{myAbs } x
\]

\[
\begin{align*}
| & x > 0 \quad = \quad y \\
| & \text{otherwise} \quad = \quad -y
\end{align*}
\]

\[
\text{where} \quad y = x
\]
Local definitions: \texttt{let}

\begin{equation}
\texttt{let } x = e_1 \texttt{ in } e_2
\end{equation}

defines \( x \) locally in \( e_2 \)

Example:

\begin{verbatim}
let x = 2+3 in x^2 + 2*x
= 35
\end{verbatim}

Like \( e_2 \) \texttt{where } \( x = e_1 \)

But can occur anywhere in an expression

where: only after function definitions
Layout: the offside rule

\[
a = 10 \quad a = 10 \quad a = 10
\]
\[
b = 20 \quad b = 20 \quad b = 20
\]
\[
c = 30 \quad c = 30 \quad c = 30
\]

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

\[
a = 10 + \quad a = 10 + \quad a = 10 +
\]
\[
20 \quad 20 \quad 20
\]

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

Function application: \( f \ a \ b \)

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

Example
\[ 5 \ '\text{mod}\' \ 3 \ = \ \text{mod} \ 5 \ 3 \]

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{T}rue, '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
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'

...
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:

\[
\begin{align*}
e_1 :: T & \quad \ldots \quad e_n :: T \\
\hline
[e_1, \ldots, e_n] & :: [T]
\end{align*}
\]

[True, ’c’] is not type-correct!!!

All elements in a list must have the same type
(True, 'c') ::

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

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

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

\[
[3 \ldots 1] = []
\]

\[
['a' \ldots 'c'] = ['a', 'b', 'c']
\]
Concatenation: ++

Concatenates two lists of the same type:

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

\[ [1, 2] ++ [\text{'a'}] \]
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:

\[ [ x^2 \mid x <- [1 .. 5]] \]

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

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

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

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

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

pattern <- list expression
is called a generator

Precise definition of pattern later.
List comprehension — Tests

\[
[ x^2 \mid x \leftarrow [1..5], \text{odd } x]
\]
\[= [1, 9, 25]\]

\[
[ x^2 \mid x \leftarrow [1..5], \text{odd } x, x > 3]
\]
\[= [25]\]

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

Boolean expressions are called tests
Defining functions by list comprehension

Example

\[
\text{factors} :: \text{Int} \rightarrow [\text{Int}], \\
\text{factors} \ n = [m \mid m \leftarrow [1 \ldots n], \ n \mod m == 0]
\]

\[
\Rightarrow \text{factors 15} = [1, 3, 5, 15]
\]

\[
\text{prime} :: \text{Int} \rightarrow \text{Bool}, \\
\text{prime} \ n = \text{factors} \ n == [1,n]
\]

\[
\Rightarrow \text{prime 15} = \text{False}
\]

\[
\text{primes} :: \text{Int} \rightarrow [\text{Int}], \\
\text{primes} \ n = [p \mid p \leftarrow [1 \ldots n], \ \text{prime} \ p]
\]

\[
\Rightarrow \text{primes 100} = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
\]

List comprehension — General form

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

where `expr` is an expression and each `E_i` is a generator or a test
Multiple generators

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

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

Analogy: each generator is a for loop:

\[
\text{for all } i \leftarrow [1 \ldots 2] \\
\quad \text{for all } j \leftarrow [7 \ldots 9] \\
\quad \ldots
\]

Key difference:

Loops do something
Expressions produce something
Dependent generators

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

\[ = [(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

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

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

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

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

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

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

What is the type of concat?

[[a]] -> [a]
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:

\[
\begin{align*}
&f :: T \rightarrow U \quad T' \leq T \\
\implies &f :: T' \rightarrow U
\end{align*}
\]

(remember: horizontal line = implication)

**Parametric polymorphism** as in Haskell:

Types may contain type variables ("parameters")

\[
\begin{align*}
f :: T \\
\implies f :: T[U/a]
\end{align*}
\]

where \( T[U/a] = \text{"} T \text{ with } a \text{ replaced by } U \text{"} \)

Example: \((a \rightarrow a)[Bool/a] = \text{bool} \rightarrow \text{bool}\)

(Often called **ML-style polymorphism**)

72
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

\[ \text{length} :: [a] \rightarrow \text{Int} \]
\[ \text{length} [5, 1, 9] = 3 \]

\[ (\text{++}) :: [a] -> [a] -> [a] \]
\[ [1, 2] ++ [3, 4] = [1, 2, 3, 4] \]

\[ \text{reverse} :: [a] \rightarrow [a] \]
\[ \text{reverse} [1, 2, 3] = [3, 2, 1] \]

\[ \text{replicate} :: \text{Int} \rightarrow a \rightarrow [a] \]
\[ \text{replicate} 3 \ 'c' = "ccc" \]
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 n 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 \( a \rightarrow a \rightarrow a \) for any type of class Num

- Class \textit{Num} is the class of \textit{numeric types}.
- Predefined numeric types: \textit{Int}, \textit{Integer}, \textit{Float}
- Types of class \textit{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
  \[
  (==) :: \text{Eq } a \implies a \to a \to \text{Bool}
  \]
  \[
  (/=) :: \text{Eq } a \implies a \to a \to \text{Bool}
  \]
  Most types are of class `Eq`. Exception:

- The class `Ord` of *ordered types*, i.e. types that possess
  \[
  (<) :: \text{Ord } a \implies a \to a \to \text{Bool}
  \]
  \[
  (\leq) :: \text{Ord } a \implies a \to a \to \text{Bool}
  \]

More on type classes later. Don’t confuse with OO classes.
Warning: == []

null xs = xs == []

Why?

== on [a] may 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

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

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

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

Now QuickCheck works
Conditional properties have result type Property

Example

```
prop_rev10 :: [Int] -> Property
prop_rev10 xs =
    length xs <= 10 ==> 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]
Pictures.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 \((::) : : a \rightarrow [a] \rightarrow [a]\)

<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 

\[
\text{[]} \quad \text{or of the form}
\]

\[x : \; xs\quad \text{where} \]

\[x \quad \text{is the head (first element, Kopf), and} \]

\[xs \quad \text{is the tail (rest list, Rumpf)} \]

\[
\text{[]} \quad \text{and (:) are called constructors}
\]

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:

\[
\text{head} \left( x : \; xs \right) = x \\
\text{tail} \left( x : \; xs \right) = xs
\]
(+++) 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 = \textit{pattern matching}.

A \textit{pattern} can be

• a variable such as \texttt{x} or a \texttt{wildcard} \_ \texttt{(underscore)}
• a \texttt{literal} like \texttt{1}, ’\texttt{a}’, "\texttt{xyz}"\texttt{, . . .}
• a \texttt{tuple} \( (p_1, \ldots, p_n) \) where each \( p_i \) is a pattern
• a \texttt{constructor pattern} \texttt{C p}_1 \ldots \texttt{p}_n
  where \( C \) is a constructor and each \( p_i \) is a pattern

Note: \texttt{True} and \texttt{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

\[
f \text{pat}_1 = e_1 \\
\vdots \\
f \text{pat}_n = e_n
\]

If \( f \) has multiple arguments:

\[
f \text{pat}_{11} \ldots \text{pat}_{1k} = e_1 \\
\vdots
\]

Conditional equations:

\[
f \text{patterns} \mid \text{condition} = e
\]

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

Example (contrived)

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:

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

- **base**: no call of \( f \)
- **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

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

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

Example

```haskell
ascending :: Ord a => [a] -> bool
ascending [] = True
ascending [_] = True
ascending (x : y : zs) = x <= y && ascending (y : zs)
```
Beyond primitive recursion: Multiple arguments

Example

\[
\text{zip} :: [a] \to [b] \to [(a,b)] \\
\text{zip } (x:xs) \ (y:ys) = (x,y) : \text{zip } xs \ ys \\
\text{zip } _ \ _ = []
\]

Alternative definition:

\[
\text{zip'} [] [] = [] \\
\text{zip'} (x:xs) \ (y:ys) = (x,y) : \text{zip'} xs \ ys
\]

\text{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

```haskell
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

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

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

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) [] = ups2 xs [x]
ups2 [] ys = [reverse ys]
ups2 (x:xs) (y:ys)
  | x >= y  = ups2 xs (x:y:ys)
  | otherwise = reverse (y:ys) : ups2 (x:xs) []
How can we quickCheck the result of ups?
Convention

Identifiers of list type end in ‘s’:
xs, ys, zs, ...
Mutual recursion

Example

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

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

> \( f\ 3 \)

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

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

\[
> \quad f \ 3
\]

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

> f 3

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

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

> f 3

Binding occurrence
Bound occurrence
Scope of binding
Scoping by example

Summary:

- Order of definitions is irrelevant
- Parameters and \texttt{where}-defs are local to each equation
5. Proofs

Proving properties
Definedness
Interlude: Type inference/reconstruction
Aim

Guarantee functional (I/O) properties of software

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

QuickCheck is good, proof is better

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

Donald E. Knuth, 1977
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 \([1,2] ++ [] = [1] ++ [2]\):

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

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:[] \+ \[]) \quad \text{-- by def of \(\+\)} \\
= 1 : 2 : ([] \+ \[]) \quad \text{-- by def of \(\+\)} \\
= 1 : 2 : [] \quad \text{-- by def of \(\+\)}
\]

\[
1:[] \+ 2:[] \\
= 1 : ([] \+ 2:[]) \quad \text{-- by def of \(\+\)} \\
= 1 : 2 : [] \quad \text{-- 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 $P(x:xs)$

One and the same fixed $xs$!

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

\[
(\[] ++ ys) ++ zs \\
= ys ++ zs \quad -- \text{by def of ++} \\
= [] ++ (ys ++ zs) \quad -- \text{by def of ++}
\]

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

\[
((x:xs) ++ ys) ++ zs \\
= (x : (xs ++ ys)) ++ zs \quad -- \text{by def of ++} \\
= x : ((xs ++ ys) ++ zs) \quad -- \text{by def of ++} \\
= x : (xs ++ (ys ++ zs)) \quad -- \text{by IH} \\
(x:xs) ++ (ys ++ zs) \\
= x : (xs ++ (ys ++ zs)) \quad -- \text{by def of ++}
\]
Lemma $P(xs)$

Proof by structural induction on $xs$

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

Proof of $P([])$

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

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

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

Proof by structural induction on \( xs \)

Base case:

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

\[
\text{length } ([] ++ ys) \\
= \text{length } ys \quad -- \text{by def of } ++ \\
\text{length } [] + \text{length } ys \\
= 0 + \text{length } ys \quad -- \text{by def of length} \\
= \text{length } ys
\]
Induction step:
IH: length(xs ++ ys) = length xs + length ys
To show: length((x:xs)++ys) = length(x:xs) + length ys
length((x:xs) ++ ys)
= length(x : (xs ++ ys))        -- by def of ++
= 1 + length(xs ++ ys)         -- by def of length
= 1 + length xs + length ys   -- by IH
length(x:xs) + length ys
= 1 + length xs + length ys   -- by def of length
Example: reverse of ++

\textbf{Lemma} \texttt{reverse(xs ++ ys) = reverse ys ++ reverse xs}  
\textbf{Proof} by structural induction on \texttt{x}s  
Base case:  
To show:\texttt{ reverse ([] ++ ys) = reverse ys ++ reverse []}  
\texttt{reverse ([] ++ ys)}  
\texttt{= reverse ys} \quad \text{-- by def of ++}  
\texttt{reverse ys ++ reverse []}  
\texttt{= reverse ys ++ []} \quad \text{-- by def of reverse}  
\texttt{= reverse ys} \quad \text{-- by Lemma \texttt{app_Nil2}}

\textbf{Lemma} \texttt{app_Nil2: xs ++ [] = xs}  
\textbf{Proof} exercise
Induction step:
IH: reverse(xs ++ ys) = reverse ys ++ reverse xs
To show: reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys)) -- by def of ++
= reverse(xs ++ ys) ++ [x] -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x] -- by IH
= reverse ys ++ (reverse xs ++ [x]) -- by Lemma app_assoc
reverse ys ++ reverse(x:xs)
= reverse ys ++ (reverse xs ++ [x]) -- by def of reverse
Proof heuristic

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

- Proof by cases
- Generalization
Example: proof by cases

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

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

Proof by structural induction on xs

Base case:
To show: rem z ([] ++ ys) = rem z [] ++ rem z ys
rem z ([] ++ ys)
= rem z ys               -- by def of ++
rem z [] ++ rem z ys
= rem z ys               -- by def of rem and ++
rem x [] = []
rem x (y:ys) | x==y = rem x ys
| otherwise = y : rem x ys

Induction step:
IH: rem z (xs ++ ys) = rem z xs ++ rem z ys
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
rem z (x:xs) ++ rem z ys
= rem z xs ++ rem z ys -- by def of rem
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
rem z (x:xs) ++ rem z ys
= x : (rem z xs ++ rem z ys) -- by def of rem and ++
Proof by cases

Works just as well for if-then-else, for example

\[
\text{rem } x \; [] = []
\]
\[
\text{rem } x \; (y:ys) = \text{if } x == y \; \text{then } \text{rem } x \; ys
\]
\[
\text{else } y \; : \; \text{rem } x \; ys
\]
Inefficiency of reverse

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

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

An improvement: itrev

\[\text{itrev} :: [a] \rightarrow [a] \rightarrow [a]\]
\[\text{itrev} \; [] \; xs = xs\]
\[\text{itrev} \; (x:xs) \; ys = \text{itrev} \; xs \; (x:ys)\]

Lemma $\text{itrev } xs \; [] = \text{reverse } xs$

Proof by structural induction on $xs$

Induction step fails:
IH: $\text{itrev } xs \; [] = \text{reverse } xs$

To show: $\text{itrev } (x:xs) \; [] = \text{reverse } (x:xs)$

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

Problem: IH not applicable because too specialized: $[]$
Generalization

**Lemma** \( \text{itrev} \; \text{xs} \; \text{ys} = \text{reverse} \; \text{xs} \; ++ \; \text{ys} \)

**Proof** by structural induction on \( \text{xs} \)

Induction step:

IH: \( \text{itrev} \; \text{xs} \; \text{ys} = \text{reverse} \; \text{xs} \; ++ \; \text{ys} \)

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

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

Note: IH is used with \( \text{x:ys} \) instead of \( \text{ys} \)
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

\[ qsort :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \]
\[ qsort \; [] \; = \; [] \]
\[ qsort \; (x:xs) \; = \; qsort \; \text{below} \; ++ \; [x] \; ++ \; qsort \; \text{above} \]
  \hspace{1cm}
  \text{where}\; \text{below} = [y \mid y \leftarrow xs, y \leq x]
  \hspace{1cm}
  \text{above} = [z \mid y \leftarrow 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 \( \text{length below} \leq \text{length} \; xs < \text{length} \; (x:xs) \) (also for above).

Therefore \( qsort \; \text{below} \) and \( qsort \; \text{above} \) are sorted by IH.

By construction \( \text{below} \) contains only elements \((\leq x)\).

Therefore \( qsort \; \text{below} \) contains only elements \((\leq x)\) (proof!).

Analogously for \( \text{above} \) and \((x<)\).

Therefore \( qsort \; (x:xs) \) is sorted.
Is that all? Or should we prove something else about sorting?

How about this sorting function?

```
  superquicksort _ = []
```

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

**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 \( f \ r \) 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: $f :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$ terminates if there is a measure function $m :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \mathbb{N}$ 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 $t$: $m \ p_1 \ldots \ p_n > m \ r_1 \ldots \ r_n$.

Of course, all other functions that are called by $f$ must also terminate.
Infinite values

Haskell allows infinite values, in particular infinite lists.

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

Infinite objects must be constructed by recursion:

\[
ones = 1 : \ones
\]

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.
Exceptions

If we use arithmetic equations like $x - x = 0$ unconditionally, we can “lose” exceptions:

$$\text{head } xs - \text{head } xs = 0$$
$$\text{is only true if } xs /= []$$

In such cases, we can prove equations $e_1 = e_2$ that are only partially correct:

If $e_1$ and $e_2$ do not produce a runtime exception then they evaluate to the same value.
In this chapter everything must terminate
This avoids undefined and infinite values
This simplifies proofs
5.3 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression (and all subexpressions)

Given: an expression $e$

Type inference:

1. Give all variables and functions in $e$ their most general type
2. From $e$ set up a system of equations between types
3. Simplify the equations
Example: concat (replicate x y)

Initial type table:

\begin{align*}
  x &:: a \\
  y &:: b \\
  \text{replicate} &:: \text{Int} \to c \to [c] \\
  \text{concat} &:: [[d]] \to [d]
\end{align*}

For each subexpression $f \ e_1 \ldots e_n$ generate $n$ equations:

\begin{align*}
  a &= \text{Int}, \ b = c \\
  [c] &= [[d]]
\end{align*}

Simplify equations:

\begin{align*}
  [c] &= [[d]] & \leadsto & c = [d] \\
  b &= c & \leadsto & b = [d]
\end{align*}

Solution to equation system: $a = \text{Int}, \ b = [d], \ c = [d]$

Final type table:

\begin{align*}
  x &:: \text{Int} \\
  y &:: [d] \\
  \text{replicate} &:: \text{Int} \to [d] \to [[d]] \\
  \text{concat} &:: [[d]] \to [d]
\end{align*}
Algorithm

1. Give the variables $x_1, \ldots, x_n$ in $e$ the types $a_1, \ldots, a_n$ where the $a_i$ are distinct type variables.

2. Give each occurrence of a function $f :: \tau$ in $e$ a new type $\tau'$ that is a copy of $\tau$ with fresh type variables.

3. For each subexpression $f \; e_1 \ldots e_n$ of $e$ where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where $e_i$ has type $\sigma_i$ generate the equations $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$.

4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable $a$ by $\tau$ everywhere (if $a$ does not occur in $\tau$)
   - $T \; \sigma_1 \ldots \sigma_n = T \; \tau_1 \ldots \tau_n \leadsto \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$
     (where $T$ is a type constructor, e.g. $[.]$, $\to$, etc)
   - $a = T \ldots a \ldots$ or $T \ldots a \ldots = a$: type error!
   - $T \ldots = T' \ldots$ where $T \neq T'$: type error!
• For simple expressions you should be able to infer types “durch scharfes Hinsehen”
• Use the algorithm if you are unsure or the expression is complicated
• Or use the Haskell interpreter
6. Higher-Order Functions

Applying functions to all elements of a list: \texttt{map}

Filtering a list: \texttt{filter}

Combining the elements of a list: \texttt{foldr}

Lambda expressions

Extensionality

Curried functions

More library functions

Case study: Counting words
Recall [Pic is short for Picture]

\[
\text{alterH} :: \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic} \\
\text{alterH} \text{ pic1} \text{ pic2} \text{ 1} = \text{pic1} \\
\text{alterH} \text{ pic1} \text{ pic2} \text{ n} = \text{beside pic1} \left(\text{alterH pic2 pic1 (n-1)}\right) \\
\]

\[
\text{alterV} :: \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic} \\
\text{alterV} \text{ pic1} \text{ pic2} \text{ 1} = \text{pic1} \\
\text{alterV} \text{ pic1} \text{ pic2} \text{ n} = \text{above pic1} \left(\text{alterV pic2 pic1 (n-1)}\right) \\
\]

Very similar. Can we avoid duplication?

\[
\text{alt} :: (\text{Pic} \to \text{Pic} \to \text{Pic}) \to \text{Pic} \to \text{Pic} \to \text{Int} \to \text{Pic} \\
\text{alt} \text{ f} \text{ pic1} \text{ pic2} \text{ 1} = \text{pic1} \\
\text{alt} \text{ f} \text{ pic1} \text{ pic2} \text{ n} = \text{f pic1} \left(\text{alt f pic2 pic1 (n-1)}\right) \\
\]

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

\[ \ldots \to (\ldots \to \ldots) \to \ldots \]

Higher-order functions capture patterns of computation
6.1 Applying functions to all elements of a list: \texttt{map}

\textbf{Example}

\begin{verbatim}
map even [1, 2, 3] = [False, True, False]
map toLower "R2-D2" = "r2-d2"
map reverse ["abc", "123"] = ["cba", "321"]
\end{verbatim}

What is the type of \texttt{map}?

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
\end{verbatim}
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 map

\[
\begin{align*}
\text{map } f \hspace{1em} &\hspace{1em} [\hspace{1em}] \hspace{1em} = \hspace{1em} [] \\
\text{map } f \hspace{1em} (x:xs) \hspace{1em} &\hspace{1em} = \hspace{1em} f \hspace{1em} x \hspace{1em} : \hspace{1em} \text{map } f \hspace{1em} xs \\
\text{map } \text{sqr} \hspace{1em} [1, \hspace{1em} -2] \hspace{1em} &\hspace{1em} = \hspace{1em} \text{map } \text{sqr} \hspace{1em} (1 \hspace{1em} : \hspace{1em} -2 \hspace{1em} : \hspace{1em} []) \\
&\hspace{1em} = \hspace{1em} \text{sqr} \hspace{1em} 1 \hspace{1em} : \hspace{1em} \text{map } \text{sqr} \hspace{1em} (-2 \hspace{1em} : \hspace{1em} []) \\
&\hspace{1em} = \hspace{1em} \text{sqr} \hspace{1em} 1 \hspace{1em} : \hspace{1em} \text{sqr} \hspace{1em} (-2) \hspace{1em} : \hspace{1em} (\text{map } \text{sqr} \hspace{1em} []) \\
&\hspace{1em} = \hspace{1em} \text{sqr} \hspace{1em} 1 \hspace{1em} : \hspace{1em} \text{sqr} \hspace{1em} (-2) \hspace{1em} : \hspace{1em} [] \\
&\hspace{1em} = \hspace{1em} 1 \hspace{1em} : \hspace{1em} 4 \hspace{1em} : \hspace{1em} [] \\
&\hspace{1em} = \hspace{1em} [1, \hspace{1em} 4]
\end{align*}
\]
Some properties of map

length (map f xs) = length xs

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

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

Proofs by induction
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

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:

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

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

\[
\text{filter } p \ (x\text{:xs}) \mid p \ x \ = \ x \ : \ \text{filter } p \ x\text{s} \\
\mid \ \text{otherwise} \ = \ \text{filter } p \ x\text{s}
\]
Some properties of filter

True or false?

\[ \text{filter } p \ (xs ++ ys) = \text{filter } p \ xs ++ \text{filter } p \ ys \]

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

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

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

foldr (⊕) z [x₁, ..., xₙ] = x₁ ⊕ ... ⊕ xₙ ⊕ z

Defined in Prelude:

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

Applications:

sum xs = foldr (+) 0 xs

concat xss = foldr (++) [] xss

What is the most general type of foldr?
foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs

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

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

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
= 1 + -2 + 0
= -1
More applications of foldr

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

\[ \text{squares } x_\text{s} = \text{map } \lambda x \rightarrow x \times x \text{ where } \lambda x \rightarrow x \times x \]

Do we really need to define \( \lambda x \rightarrow x \times x \) explicitly? No!

\[ \lambda x \rightarrow x \times x \]

is the anonymous function with

formal parameter \( x \) and result \( x \times x \)

In mathematics:

\[ x \mapsto x \times x \]

Evaluation:

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

Usage:

\[ \text{squares } x_\text{s} = \text{map } (\lambda x \rightarrow x \times x) \, x_\text{s} \]
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 \textit{lambda calculus} in the 1930s 

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

Example

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

The general rule:

\[(\lambda x \to e) :: T_1 \to T_2 \]
if \( x :: T_1 \) implies \( e :: T_2 \)
Evaluating lambda expressions

\[(\lambda x \rightarrow \text{body}) \ arg \ = \ \text{body} \ \text{with} \ x \ \text{replaced by} \ arg\]

Example

\[(\lambda xs \rightarrow xs \ ++ \ xs) \ [1] \ = \ [1] \ ++ \ [1]\]
Sections of infix operators

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

etc

Example

squares xs = map (^ 2) xs
List comprehension

Just syntactic sugar for combinations of map

\[
[f \ x \mid x \gets xs] = \text{map} \ f \ xs
\]

filter

\[
[x \mid x \gets xs, p \ x] = \text{filter} \ p \ xs
\]

and concat

\[
[f \ x \ y \mid x \gets xs, y \gets ys] = \text{concat} \(
\]

\[\text{)}
\]
6.5 Extensionality

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

\[ f, g :: T_1 \rightarrow T: \]
\[ \forall a. \ f \ a = g \ a \]
\[ f = g \]

\[ f, g :: T_1 \rightarrow T_2 \rightarrow T: \]
\[ \forall a, b. \ f \ a \ b = g \ a \ b \]
\[ f = g \]
6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

\[ f \ x \ y \ = \ x+y \quad f \ x \ = \ \lambda y \to x+y \]

Both mean the same:

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

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)

\[ T_1 \to T_2 \to T \]

is just syntactic sugar for

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

\[ f \; e_1 \; e_2 \]

is just syntactic sugar for

\[ (f \; e_1) \; e_2 \]

\[ :: T_2 \to T \]

Analogously for more arguments
\( \rightarrow \) is not associative:
\[
T_1 \rightarrow (T_2 \rightarrow T) \neq (T_1 \rightarrow T_2) \rightarrow T
\]

**Example**

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

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

**Example**

\[
\begin{align*}
(f \; 3) \; 4 &= f \; (3 \; 4) & g \; (\text{id} \; \text{abs}) &= (g \; \text{id}) \; \text{abs}
\end{align*}
\]
Quiz

head tail xs

Correct?
Partial application

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

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

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

\((.)::(b\rightarrow c)\rightarrow(a\rightarrow b)\rightarrow\)
\(f\ .\ g\ =\ \ \backslash x\rightarrow f\ (g\ x)\)

Example

\(head2\ =\ head\ .\ tail\)

\(head2\ [1,2,3]\)
\(=\ (head\ .\ tail)\ [1,2,3]\)
\(=\ (\backslash x\rightarrow head\ (tail\ x))\ [1,2,3]\)
\(=\ head\ (tail\ [1,2,3])\)
\(=\ head\ [2,3]\)
\(=\ 2\)
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 = []

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

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\nnnever: 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"

\[
\begin{align*}
\text{function} & \quad \text{words} \\
\downarrow & \\
["never", "say", "never", "again"]
\end{align*}
\]

Predefined in Prelude
Step 2: Sort words

> function sort

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

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

Predefined in Data.List
Step 3: Group equal words together

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

function *group* 

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

Predefined in *Data.List*
Step 4: Count each group

\[
\begin{align*}
[["again"], ["never", "never"], ["say"]] \\
\downarrow \quad \text{map (ws -> (head ws, length ws)} \\
[("again", 1), ("never", 2), ("say", 1)]
\end{align*}
\]
Step 5: Format each group

\[
[(
"again", 1),
("never", 2),
("say", 1)]
\]

\[
\text{map } \quad (w, n) \rightarrow (w ++ ": " ++ show n)
\]

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

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

function \texttt{unlines}

\texttt{"again: 1\nnever: 2\nsay: 1\n"}

Predefined in Prelude
The solution

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

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

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:**
  $$\text{map } f \ (\text{map } g \ (x:xs))$$
  $$= f \ (g \ x) : \text{map } f \ (\text{map } g \ \text{xs})$$
  $$= f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs} \quad \text{-- by IH}$$
  $$\text{map } (f \cdot g) \ (x:xs)$$
  $$= f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs}$$

$$\implies (\text{map } f \cdot \text{map } g) \ \text{xs} = \text{map } f \ (\text{map } g \ \text{xs}) = \text{map } (f \cdot g) \ \text{xs}$$

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

Example

```haskell
elem ::
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 $a$

Type classes support generic programming: Code that works not just for one type but for a whole class of types, all types that implement the functions of the class.
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 $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq Int}$
Therefore $\text{elem} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Bool}$

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 $(\text{Bool,Int})$ 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:

\[
\text{instance (context) => } C \ T \text{ where }
\]

\[
\text{definitions}
\]

\[
T \text{ is a type}
\]

\[
\text{context} \text{ is a list of assumptions } C_i \ T_i
\]

\[
\text{definitions} \text{ are definitions of the functions of class } C
\]
Subclasses

Example

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

Class Ord inherits all the operations of class Eq

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

instance Ord Bool where
  b1 <= b2 = not b1 || b2
  b1 < b2  = b1 <= 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 data Types

data by example
The general case
Case study: boolean formulas
Structural induction
So far: no really new types, just compositions of existing types

Example: type String = [Char]

Now: \texttt{data} defines \textit{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
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:

```haskell
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)
-- assumption: < is a linear ordering
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)
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 $O(\quad )$

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

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

\[
\begin{align*}
C_1 & \quad t_{11} \quad ... \quad t_{1k_1} \\
& \vdots \\
C_n & \quad t_{n1} \quad ... \quad t_{nk_n}
\end{align*}
\]

defines the *constructors*

\[
\begin{align*}
C_1 & \quad :: \quad t_{11} \quad \rightarrow \quad ... \quad t_{1k_1} \quad \rightarrow \quad T \quad a_1 \quad ... \quad a_p \\
& \vdots \\
C_n & \quad :: \quad t_{n1} \quad \rightarrow \quad ... \quad t_{nk_n} \quad \rightarrow \quad T \quad a_1 \quad ... \quad a_p
\end{align*}
\]
Constructors are functions too!

Constructors can be used just like other functions

Example

map Just [1, 2, 3] = [Just 1, Just 2, Just 3]

But constructors can also occur in patterns!
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:

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

\[ \text{Not(Not } T) \neq T \]

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) = fromJust(lookup x v)
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 \texttt{vals} construct \textit{all} valuations?

\begin{verbatim}
prop_vals1 xs =
    length(vals xs) == 2 ^ length xs

prop_vals2 xs =
    distinct (vals xs)

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

Demo
Restrict size of test cases:

\[
\text{prop_vals1'} \ xs = \\
\quad \text{length } xs \leq 10 \implies \\
\quad \text{length(vals } xs) = 2^\text{length } xs \\
\]

\[
\text{prop_vals2'} \ xs = \\
\quad \text{length } xs \leq 10 \implies \text{distinct } \text{(vals } xs) \\
\]

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 $P(\text{Empty})$ and

**Induction step:** Prove $P(\text{Node } x \ t1 \ t2)$
assuming the induction hypotheses $P(t1)$ and $P(t2)$.
($x$, $t1$ and $t2$ are new variables)
Example

\[
\text{flat} :: \text{Tree } a \rightarrow [a] \\
\text{flat} \text{ Empty} = [] \\
\text{flat} (\text{Node } x \ t1 \ t2) = \\
\quad \text{flat} \ t1 ++ [x] ++ \text{flat} \ t2
\]

\[
\text{mapTree} :: (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b \\
\text{mapTree} \ f \ \text{Empty} = \ \text{Empty} \\
\text{mapTree} \ f \ (\text{Node } x \ t1 \ t2) = \\
\quad \text{Node} \ (f \ x) \ (\text{mapTree} \ f \ t1) \ (\text{mapTree} \ f \ t2)
\]
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: \( \text{flat} (\text{mapTree} f (\text{Node} x t1 t2)) = \text{map} f (\text{flat} (\text{Node} x t1 t2)) \)

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

-- by IH1 and IH2

map f (\text{flat} (\text{Node} x t1 t2))

= \text{map} f (\text{flat} t1 ++ [x] ++ \text{flat} t2)

= \text{map} f (\text{flat} t1) ++ [f x] ++ \text{map} f (\text{flat} t2)

-- by lemma distributivity of map over ++

Note: Base case and -- by def of ... omitted
The general (regular) case

data T a = ...

Assumption: T is a regular data type:

- Each constructor \( C_i \) of T must have a type
  \[
  t_1 \to \ldots \to t_{n_i} \to T \ a
  \]
  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. I/O

File I/O
Network 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
An impure solution

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 potentially perform side effects.

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

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

... are no longer true.
The pure solution

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

\[
\begin{align*}
\text{Char:} & \text{ the type of pure expressions that return a Char} \\
\text{IO Char:} & \text{ the type of actions that return a Char} \\
\text{IO ():} & \text{ the type of actions that return no result value}
\end{align*}
\]
• 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
Sequencing: do

A sequence of actions can be combined into a single action with the keyword do

Example

```haskell
get2 :: IO (Char,Char)
get2 = do x <- getChar
         -- result is named x
         getChar
         -- result is ignored
         y <- getChar
         return (x,y)
```
General format (observe layout!):

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

where each \ a_i \ can be one of

- an action \\
  Effect: execute action

- \ x \leftarrow \text{action} \\
  Effect: execute \text{action} :: \text{IO} \ a, \ give \ result \ the \ name \ x :: a

- \text{let } x = \text{expr} \\
  Effect: give \text{expr} the name x \\
  Lazy: \text{expr} is only evaluated when x is needed!
Write a string to standard output:

\[
\text{putStr :: String} \rightarrow \text{IO ()}
\]
\[
\text{putStr []} = \text{return ()}
\]
\[
\text{putStr (c:cs)} = \text{do putChar c}
\]
\[
\text{putStr cs}
\]

Write a line to standard output:

\[
\text{putStrLn :: IO ()}
\]
\[
\text{putStrLn cs} = \text{putStr (cs ++ "\n")}
\]
Read a line from standard input:

```haskell
getLine :: IO String
getLine = do x <- getChar
           if x == '\n' then
               return []
           else
               do xs <- getLine
                  return (x:xs)
```

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

class Read a where
    read :: String -> a

Most predefined types are in class Read.

Example:

gGetInt :: IO Integer
getInt = do xs <- getLine
            return (read xs)
Case study

The game of Hangman
in file hangman.hs
main :: IO ()
main = do putStrLn "Input secret word: "
    word <- getWord ""
    clear_screen
    guess word
    main
guess :: String -> IO ()
guess word = loop "" "" gallows where

loop :: String -> String -> [String] -> IO()
loop guessed missed gals =
do let word' =
    map (\x -> if x 'elem' guessed
            then x else '–')
        word
    writeAt (1,1)
        (head gals ++ "\n" ++ "Word: " ++ word’ ++
            "\nMissed: " ++ missed ++ "\n")
    if length gals == 1
    then putStrLn ("YOU ARE DEAD: " ++ word)
    else if word’ == word then putStrLn "YOU WIN!"
    else do c <- getChar
            let ok = c 'elem' word
            loop (if ok then c:guessed else guessed)
                 (if ok then missed else missed++[c])
                 (if ok then gals else tail gals)
Once IO, always IO

You cannot add I/O to a function without giving it an IO type

For example

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

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

\[
\text{sq} \ x = \text{do} \ \text{putStr}("I \ am \ in \ \text{sq}!") \\
\text{return}(x \times x)
\]

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

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

\[
\text{cube} \ x = \text{do} \ x2 \leftarrow \text{sq} \ x \\
\text{return}(x \times x2)
\]
Haskell is a pure functional language
Functions that have side effects must show this in their type
I/O is a side effect
Separate I/O from processing to reduce I/O creep:

```haskell
main :: IO ()
main = do s <- getLine
    let r = process s
    putStrLn r
main

process :: String -> String
process s = ...
```
9.1 File I/O
The simple way

- **type FilePath = String**
- **readFile :: FilePath -> IO String**
  
  Reads file contents *lazily*, only as much as is needed

- **writeFile :: FilePath -> String -> IO ()**
  
  Writes whole file

- **appendFile :: FilePath -> String -> IO ()**
  
  Appends string to file
import System.IO
data Handle

Opaque type, implementation dependent

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.
Files and handles

- data IOMode = ReadMode | WriteMode
  | AppendMode | ReadWriteMode
- openFile :: FilePath -> IOMode -> IO Handle
  Creates handle to file and opens file
- hClose :: Handle -> IO ()
  Closes file
By convention
all IO actions that take a handle argument begin with \texttt{h}
In ReadMode

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`
- `hGetContents :: Handle -> IO String`

Reads the whole file *lazily*
In WriteMode

- \texttt{hPutChar} :: \texttt{Handle} \to \texttt{Char} \to \texttt{IO ()}
- \texttt{hPutStr} :: \texttt{Handle} \to \texttt{String} \to \texttt{IO ()}
- \texttt{hPutStrLn} :: \texttt{Handle} \to \texttt{String} \to \texttt{IO ()}
- \texttt{hPrint} :: \texttt{Show a} \Rightarrow \texttt{Handle} \to \texttt{a} \to \texttt{IO ()}
stdin and stdout

- stdin :: Handle
  stdout :: Handle

- getChar = hGetChar stdin
  putChar = hPutChar stdout
There is much more in the Standard IO Library (including exception handling for IO actions)
Example (interactive cp: icp.hs)

main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
    hClose fromH
    hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
    do putStrLn prompt
       name <- getLine
       handle <- openFile name mode
       return handle
If `xyz.hs` contains a definition of `main`:

- `runhaskell xyz`
- `ghc xyz \implies executable file xyz`
9.2 Network I/O
import Network
Types

- **data Socket**
  A socket is one endpoint of a two-way communication link between two programs running on the network.

- **data PortId = PortNumber PortNumber | ...**

- **data PortNumber**
  
  ```
  instance Num PortNumber
  ⇒ PortNumber 9000 :: PortId
  ```
Server functions

- `listenOn :: PortId -> IO Socket`
  Create server side socket for specific port

- `accept :: Socket -> IO (Handle, ..., ...)`
  \(\Rightarrow\) can read/write from/to socket via handle

- `sClose :: Socket -> IO ()`
  Close socket
Initialization for Windows

withSocketsDo :: IO a -> IO a

Standard use pattern:

main = withSocketsDo $ do ...

Does nothing under Unix
Example (pingPong.hs)

main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 9000
  (h, _, _) <- accept sock
  hSetBuffering h LineBuffering
  loop h
  sClose sock

loop :: Handle -> IO ()
loop h = do
  input <- hGetLine h
  if take 4 input == "quit"
    then do hPutStrLn h "goodbye!"
           hClose h
    else do hPutStrLn h ("got " ++ input)
           loop h
Client functions

- `type HostName = String`  
  For example "haskell.org" or "192.168.0.1"

- `connectTo :: HostName -> PortId -> IO Handle`  
  Connect to specific port of specific host
Example (wGet.hs)

main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  putStrLn "Resource?"
  res <- getLine
  putStrLn (("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
  putStrLn s
For more detail see

http://hackage.haskell.org/package/network/docs/
Network.html

http://hackage.haskell.org/package/network/docs/
Network-Socket.html
10. Modules and Abstract Data Types

- Modules
- Abstract Data Types
- Correctness
10.1 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

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

⇒ B does not export f
```

Unless the names are mentioned in the export list

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

Or the whole module is exported

```haskell
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

\[\Rightarrow\] 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
10.2 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

-- Explicit export list/interface

-- But representation still not hidden
Does not help: hiding the type name Set
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 []
inset 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
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`
10.3 Correctness

Why is module \texttt{Set} a correct implementation of (finite) sets?

Because \texttt{empty} simulates \{\}\nand \texttt{insert \_ \_} simulates \{\_\} \cup \_
and \texttt{isin \_ \_} simulates \_ \in \_
and \texttt{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 \{\textit{a}\} for the type of finite sets over type \textit{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: \(\alpha [] = \emptyset\)
\(\alpha (x:xs) = \{x\} \cup \alpha xs\)

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

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

\(\alpha\) empty = \(\emptyset\)

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

\(\text{isin} \ x \ xs\) = \(x \in \alpha \ xs\)

\(\text{size} \ xs\) = \(|\alpha \ xs|\)
For the mathematically inclined:

\[ \alpha \text{ must be a homomorphism} \]
Implementation I: lists with duplicates

empty \quad = \quad []

insert \ x \ xs \quad = \quad x : \ xs

isin \ x \ xs \quad = \quad elem \ x \ xs

size \ xs \quad = \quad length(nub \ xs)

The simulation requirements:

\[ \alpha \emptyset = \{\}\]

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

\[ \text{isin x xs} = x \in \alpha \text{ xs} \]

\[ \text{size xs} = |\alpha \text{ xs}| \]

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

\begin{align*}
\text{empty} & = [] \\
\text{insert } x \; \text{xs} & = \text{if elem } x \; \text{xs then xs else } x:xs \\
\text{isin } x \; \text{xs} & = \text{elem } x \; \text{xs} \\
\text{size } \text{xs} & = \text{length } \text{xs}
\end{align*}

The simulation requirements:

\begin{align*}
\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 } \text{xs} & = |\alpha \ \text{xs}|
\end{align*}

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

\begin{align*}
\text{invar} :: [a] \rightarrow \text{Bool} \\
\text{invar} [\text{}] & = \text{True} \\
\text{invar} (x:xs) & = \text{not}(\text{elem } x \; \text{xs}) \&\& \text{invar } \text{xs}
\end{align*}
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:

\[ \alpha \empty = \{\} \]

\[ \text{invar } xs \implies \alpha (\text{insert } x \ xs) = \{x\} \cup \alpha \ xs \]

\[ \text{invar } xs \implies \text{isin } x \ xs = x \in \alpha \ xs \]

\[ \text{invar } xs \implies \text{size } xs = |\alpha \ xs| \]

Proofs omitted. Anything else?
invar must be invariant!

In an imperative context:

If invar is true before an operation, it must also be true after the operation.

In a functional context:

If invar is true for the arguments of an operation, it must also be true for the result of the operation.

invar is preserved by every operation.

\[
\text{invar empty} \\
\text{invar } \text{xs} \implies \text{invar (insert } x \text{ xs)}
\]

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 $\alpha :: C.T \rightarrow A.T$
and an invariant $\text{invar} :: C.T \rightarrow \text{Bool}$

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 $C.T$)
11. Case Study: Two Efficient Algorithms
This lecture covers two classic efficient algorithms in functional style on the blackboard:

**Huffman Coding**
See the Haskell book by Thompson for a detailed exposition.

**Skew Heaps**
See the original paper for an imperative presentation and the derivation of the amortized complexity:


The Haskell source files are on the course web page.
Huffman Coding

- Aim: encode text with as few bits as possible. Lossless compression, not encryption.
- Method: each character is mapped to a bit list. (Length of bit list depends on frequency of character.)

Example

\[
\begin{align*}
e & \mapsto 0, \ m & \mapsto 10, \ n & \mapsto 11 \\
\Rightarrow \ enem & \mapsto 011010 \quad \text{(which is uniquely decodable)}
\end{align*}
\]

Strings are encoded character by character
Prefix-free Codes

Definition

• A code is a mapping from characters to bit lists.
• A code is uniquely decodable if every bit list is the image of at most one string.
• A code is prefix-free if for no two different characters $x$ and $y$ the code for $x$ is a prefix of the code for $y$.

Example

a $\mapsto 1$, b $\mapsto 11$

Not prefix free and not uniquely decodable: aa $\mapsto 11$ and b $\mapsto 11$.

Fact Prefix-free codes are uniquely decodable.

We are only interested in prefix-free codes.
A prefix-free code can be represented as a binary tree.

Example
\[ e \mapsto 0, \ m \mapsto 10, \ n \mapsto 11 \]
Huffman’s Algorithm

Constructs an optimal code (tree) for a given frequency table based on the string to be encoded.

Example

String: "go go gopher"
Table: [(‘g’,3), (‘o’,3), (‘ ’,2), (‘p’,1), …]

A code t is optimal for a string cs if for all codes t’:

\[
\text{length (encode t cs)} \leq \text{length (encode t’ cs)}
\]

Key algorithmic ideas:

- Construct code tree bottom up
- Work on list of trees
- Always combine the “least frequent” trees into a new tree
Skew Heap

Implementation of *priority queue* as a *heap*, i.e., a binary tree where every child is larger than the parent:
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 

*lazy evaluation* („verzögerte Auswertung”)

Advantages:

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

Therefore Haskell is called a *lazy functional language*. Haskell is the only mainstream lazy functional language.
Evaluating expressions

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

Example:

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

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:

\[
\begin{align*}
n + (n := 1) & = 0 + (n := 1) = 0 + 1 = 1 \\
n + (n := 1) & = n + 1 = 1 + 1 = 2
\end{align*}
\]
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
\[ \text{sq} (3+4) = \text{sq} 7 = 7 \times 7 \]

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

Definition:
loop = tail loop

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

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!
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

\[ f :: \textbf{[Int]} \to \textbf{[Int]} \to \textbf{Int} \]
\[ f \, [] \, ys = 0 \]
\[ f \, (x:xs) \, [] = 0 \]
\[ f \, (x:xs) \, (y:ys) = x+y \]

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

\[
\begin{align*}
f \mathrm{m} \mathrm{n} \mathrm{p} & | \quad \mathrm{m} \geq \mathrm{n} \land \mathrm{m} \geq \mathrm{p} = \mathrm{m} \\
& | \quad \mathrm{n} \geq \mathrm{m} \land \mathrm{n} \geq \mathrm{p} = \mathrm{n} \\
& | \quad \text{otherwise} \quad = \mathrm{p}
\end{align*}
\]

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

Example: $\lambda x \rightarrow \text{False} \land x$ cannot be reduced

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

Example:
$$(\lambda x \rightarrow \text{False} \land x) \text{True} = \text{False} \land \text{True} = \text{False}$$
Built-in functions

Arithmetic operators and other built-in functions evaluate their arguments first

Example
3 * 5 is a redex
0 * head(...) is not a redex
They behave like their Haskell definition:

$$(\&\&) : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$

True $\&\&$ y = y
False $\&\&$ y = False
Lazy evaluation evaluates an expression only when needed and only as much as needed.

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

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

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

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

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

Lazy evaluation needs only linear time although 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!
Maximum of a list

max = last . inSort

Complexity?
Takeuchi Function

\[ t :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t \ x \ y \ z \mid x \leq y = y \]
\[ \mid \text{otherwise} = t \ (t \ (x-1) \ y \ z) \]
\[ t \ (y-1) \ z \ x \]
\[ t \ (z-1) \ x \ y \]

In C:

```c
int t(int x, int y, int z) {
    if (x <= y)
        return y;
    else
        return t(t(x-1, y, z), t(y-1, z, x), t(z-1, x, y));
}
```

Try \( t 15 10 0 \) — Haskell beats C!
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

348


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 : 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 $2, 3, 4, \ldots$
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

$2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ \ldots$
$2 \ 3 \ 5 \ 7 \ 11 \ \ldots$
In Haskell:

```haskell
primes :: [Int]
primes = sieve [2..]

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

Lazy evaluation:

```haskell
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 <- [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)]
> 101 ‘elem‘ primes
True

> 102 ‘elem‘ primes
nontermination

prime n = n == head (dropWhile (<n) primes)
Sharing!

There is only one copy of primes

Every time part of primes needs to be evaluated

Example: when computing take 5 primes
primes is (invisibly!) updated to remember the evaluated part

Example: primes = 2 : 3 : 5 : 7 : 11 : sieve ...

The next uses of primes are faster:

Example: now primes !! 2 needs only 3 steps

Nothing special, just the automatic result of sharing
The list of Fibonacci numbers

Idea: 0 1 1 2 ...
     + 0 1 1 ...
     = 0 1 2 3 ...

From Prelude: zipWith
Example: zipWith f [a1, a2, ...] [b1, b2, ...]
         = [f a1 b1, f a2 b2, ...]

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

How about
fibs = 0 : 1 : [x+y | x <- fibs, y <- tail fibs]
Hamming numbers

Definition
\[ H = \{1\} \cup \{2 \cdot h \mid h \in H\} \cup \{3 \cdot h \mid h \in H\} \cup \{5 \cdot h \mid h \in H\} \]
(Due to Richard Hamming, Turing award winner 1968)

Problem: list \( H \) in increasing order: 1, 2, 3, 4, 5, 6, 8, 9, …

\[
\text{hams} :: \text{[Int]}
\]
\[
\text{hams} = 1 : \text{merge} \ [2\times h \mid h \leftarrow \text{hams}]
\]
\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{merge} \ [3\times h \mid h \leftarrow \text{hams}]
\]
\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad [5\times h \mid h \leftarrow \text{hams}])
\]

merge \((x:xs)\) \((y:ys)\)
\[
| \ x < y \quad = \quad x : \text{merge} \ x s \ (y:ys) \\
| \ x > y \quad = \quad y : \text{merge} \ (x:xs) \ y s \\
| \ \text{otherwise} \quad = \quad x : \text{merge} \ x s \ y s
\]
Game tree

data Tree p v = Tree p v [Tree p v]

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice

```
gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
gameTree next val = tree where
  tree p = Tree p (val p) (map tree (next p))

chessTree = gameTree ...
```
minimax :: Ord v => Int -> Bool -> Tree p v -> v
minimax d player1 (Tree p v ts) =
  if d == 0 || null ts then v
  else let vs = map (minimax (d-1) (not player1)) ts
       in if player1 then maximum vs else minimum vs

> minimax 3 True chessTree
Generates chessTree up to level 3

> minimax 4 True chessTree
Needs to search 4 levels, but only level 4 needs to be generated
13. I/O and Monads

I/O
File I/O
Network 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
An impure solution

Most languages allow functions to perform I/O without reflecting it in their type.

Assume that Haskell were to provide an input function

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

Now all functions potentially perform side effects.

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

\[
\begin{align*}
\text{inputInt} - \text{inputInt} &= 0 \\
\text{inputInt} + \text{inputInt} &= 2*\text{inputInt} \\
\vdots
\end{align*}
\]

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**

- \textbf{Char}: the type of pure expressions that return a \texttt{Char}
- \textbf{IO Char}: the type of actions that return a \texttt{Char}
- \textbf{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
Sequencing: do

A sequence of actions can be combined into a single action with the keyword do

Example

def get2 :: IO (Char,Char)
def get2 = do x <- getChar  -- result is named x
            getChar
            y <- getChar  -- result is ignored
            return (x,y)
General format (observe layout!):

\[
do \ a_1 \\
\vdots \\
\ a_n
\]

where each \( a_i \) can be one of

- an action
  
  Effect: execute action

- \( x \leftarrow \text{action} \)
  
  Effect: execute \text{action}, give result the name \( x \)

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

Write a string to standard output:

\[
\text{putStr :: String } \rightarrow \text{ IO ()}
\]
\[
\text{putStr [] } = \text{ return ()}
\]
\[
\text{putStr (c:cs) } = \text{ do putChar c}
\]
\[
\hspace{1cm} \text{putStr cs}
\]

Write a line to standard output:

\[
\text{putStrLn :: IO ()}
\]
\[
\text{putStrLn cs } = \text{ putStr (cs ++ 'n')}
\]
Read a line from standard input:

```haskell
getLine :: IO String
getLine = do x <- getChar
    if x == '\n' then
        return []
    else
        do xs <- getLine
           return (x:xs)
```

Actions are normal Haskell values and can be combined as usual, for example with if-then-else.
Prompt for a string and display its length:

```
strLen :: IO ()
strLen = do putStr "Enter a string: "
        xs <- getLine
        putStr "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)
```
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

```haskell
sq :: Int -> Int
cube :: Int -> Int
sq x = x* x
cube x = x * sq x
```

Let us try to make `sq` print out some message:

```haskell
sq x = do putStr("I am in sq!")
        return(x*x)
```

What is the type of `sq` now? `Int -> IO Int`

And this is what happens to `cube`:

```haskell
cube x = do x2 <- sq x
          return(x * x2)
```
Haskell is a pure functional language
Functions that have side effects must show this in their type
I/O is a side effect
Separate I/O from processing to reduce I/O creep:

```haskell
main :: IO ()
main = do s <- getLine
    let r = process s
    putStrLn r
    main

process :: String -> String
process s = ...
```
13.2 File I/O
The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`
  Reads file contents *lazily*, only as much as is needed
- `writeFile :: FilePath -> String -> IO ()`
  Writes whole file
- `appendFile :: FilePath -> String -> IO ()`
  Appends string to file
import System.IO
data Handle

Opaque type, implementation dependent

*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.*
Files and handles

- data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
- openFile :: FilePath -> IOMode -> IO Handle
  Creates handle to file and opens file
- hClose :: Handle -> IO ()
  Closes file
By convention
all IO actions that take a handle argument begin with `h`
In ReadMode

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`
- `hGetContents :: Handle -> IO String`

Reads the whole file *lazily*
In WriteMode

- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`
- `hPutStrLn :: Handle -> String -> IO ()`
- `hPrint :: Show a => Handle -> a -> IO ()`
stdin and stdout

- stdin :: Handle
  stdout :: Handle

- getChar = hGetChar stdin
  putChar = hPutChar stdout
There is much more in the **Standard IO Library**
(including exception handling for IO actions)
Example (interactive cp: icp.hs)

main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
contents <- hGetContents fromH
hPutStr toH contents
hClose fromH
hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
    do putStrLn prompt
       name <- getLine
       handle <- openFile name mode
       return handle
Executing \texttt{xyz.hs}

If \texttt{xyz.hs} contains a definition of \texttt{main}:

- \texttt{runhaskell xyz}
  
or

- \texttt{ghc xyz} $\mapsto$ executable file \texttt{xyz}
13.3 Network I/O
import Network
Types

- `data Socket`

- `data PortId = PortNumber PortNumber | ...`

- `data PortNumber`
  
  `instance Num PortNumber`
  
  `⇒ PortNumber 9000 :: PortId`
Server functions

- `listenOn :: PortId -> IO Socket`
  Create server side socket for specific port

- `accept :: Socket -> IO (Handle, ..., ...)`
  can read/write from/to socket via handle

- `sClose :: Socket -> IO ()`
  Close socket
Initialization for Windows

withSocketsDo :: IO a -> IO a

Standard use pattern:

main = withSocketsDo $ do ...

Does nothing under Unix
Example (pingPong.hs)

main :: IO ()
main = withSocketsDo $ do
    sock <- listenOn $ PortNumber 9000
    (h, _, _) <- accept sock
    hSetBuffering h LineBuffering
    loop h
sClose sock

loop :: Handle -> IO ()
loop h = do
    input <- hGetLine h
    if take 4 input == "quit"
    then do hPutStrLn h "goodbye!"
           hClose h
    else do hPutStrLn h ("got " ++ input)
           loop h
Client functions

- `type Hostname = String`

- `connectTo :: Hostname -> PortId -> IO Handle`
  Connect to specific port of specific host
Example (wget.hs)

main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  hPutStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
  putStrLn s
13.4 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 \( \text{IO } b \)

\[
\begin{align*}
\text{do } x & \leftarrow \text{act}_1 \\
\text{act}_2
\end{align*}
\]

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

Example

\[
\begin{align*}
\text{do } x & \leftarrow \text{getChar} \\
\text{putChar } x
\end{align*}
\]

\( \mapsto \)

\[ \text{getChar} \gg\gg (\lambda x \rightarrow \text{putChar } x) \]
In general

\[
\text{do } \quad x_1 \leftarrow a_1 \\
\vdots \\
\quad x_n \leftarrow a_n \\
\quad \text{act}
\]

is syntax for

\[
\text{a}_1 \quad \gg= \quad \backslash x_1 \rightarrow \\
\vdots \\
\text{a}_n \quad \gg= \quad \backslash x_n \rightarrow \\
\quad \text{act}
\]
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:

\[
p1 *** p2 = \lambda xs \rightarrow \\
\text{case } p1 \ xs \ \text{of} \\
\quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \text{Just}(b,ys) \rightarrow \text{case } p2 \ ys \ \text{of} \\
\quad \quad \text{Nothing} \rightarrow \text{Nothing} \\
\quad \quad \text{Just}(c,zs) \rightarrow \text{Just}((b,c),zs)
\]

\rightarrow

\[
p1 *** p2 = \lambda xs \rightarrow \\
\quad \text{do } (b,ys) <- p1 \ xs \\
\quad \quad (c,zs) <- p2 \ ys \\
\quad \quad \text{return } ((b,c),zs)
\]

The do version has a much more general type \texttt{Monad m} -> ...
Maybe models possible failure with Just/Nothing

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

```haskell
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 ≤ number of cbv steps
  \[\implies O\text{-analysis under cbv also correct for Haskell but can be too pessimistic}\]
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: \( \text{length } (_ : \text{xs}) = \text{length } \text{xs} + 1 \)
Reduce \( \text{length } [1,2,3] \)

Compare: \( \text{id } [] = [] \)
\( \text{id } (x : \text{xs}) = x : \text{id } \text{xs} \)
Reduce \( \text{id } [e1,e2] \)
Copies list but shares elements.
\( T_\varepsilon(n) \) = number of steps required for the evaluation of \( \varepsilon \) 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 \( \varepsilon \)
- 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 \( \varepsilon \) does not compute with the elements of the list.
  Not sufficient for function \( \ldots \).
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

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

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

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

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

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

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

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

Observation:

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

Sum up all $T_f(n_1, \ldots, n_k)$
where $f \ e_1 \ldots e_n$ 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 $\text{min } xs = \text{head(sort } xs)$

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

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*

*D*on *K*nuth

But we are in week $n - 1$ now ;-

The ideal of program optimization:

1. Write (possibly) inefficient but correct code
2. Optimize your code *and prove equivalence to correct version*
No duplication

Eliminate common subexpressions with where (or let)

Example

\[ f \ x = g \ (h \ x) \ (h \ x) \]

\[ f \ x = g \ y \ y \ \text{where} \ y = h \ x \]
The definition of a function \( f \) is tail recursive / endrekursiv if every recursive call is in “end position”,

\[ \text{\( f \) is tail recursive / endrekursiv} \]

- it is the last function call before leaving \( f \),
- nothing happens afterwards
- no call of \( f \) is nested in another function call

**Example**

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

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

\[
\begin{align*}
\text{loop: if null} \; xs & \text{ then return} \; n \\
xs & \quad := \text{tail} \; xs \\
n & \quad := \; n+1 \\
goto & \; \text{loop}
\end{align*}
\]
What does tail recursive mean for

\[ f \ x = \text{if } b \ \text{then } e_1 \ \text{else } e_2 \]

- \( f \) does not occur in \( b \)
- if \( f \) occurs in \( e_i \) then only at the outside: \( e_i = f \ldots \)

Tail recursive example:

\[ f \ x = \text{if } x > 0 \ \text{then } f(x-1) \ \text{else } f(x+1) \]

Similar for guards and case \( e \ \text{of} \):

- \( f \) does not occur in \( e \)
- if \( f \) occurs in any branch then only at the outside: \( f \ldots \)
Accumulating parameters

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

Purpose:
  • tail recursion
  • replace (++) by (:

\[
\text{length2 } \text{[]} \quad n = n \\
\text{length2 } (x:xs) \quad n = \text{length2 } xs \quad (n+1)
\]

\[
\text{length’ } xs = \text{length2 } xs \quad 0
\]

Correctness:

**Lemma** \[\text{length2 } xs \quad n = \text{length } xs + n\]

\[\implies \text{length’ } xs = \text{length } xs\]
Accumulating parameter: reverse

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

\[T_{\text{reverse}}(n) = O(n^2)\]

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

Not just tail recursive also linear:

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

\[\implies T_{\text{itrev}}(m, n) = O(m)\]
Accumulating parameter: tree flattening

data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a) = [a]
flat (Node t1 t2) = flat t1 ++ flat t2

Size measure: height of tree (height of Tip = 1)

\[
T_{flat}(1) = O(1) \\
T_{flat}(h + 1) = 2 \times T_{flat}(h) + T++(2^h, 2^h) \\
= 2 \times T_{flat}(h) + O(2^h)
\]

\[\Rightarrow T_{flat}(h) = O(h \times 2^h)\]

With accumulating parameter:

flat2 :: Tree a -> [a] -> [a]
Accumulating parameter: foldl

\[
\text{foldr } f \ z \ [] = z \\
\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs) \\
\text{foldr } f \ z \ [x_1, \ldots, x_n] = x_1 \ 'f' \ (\ldots \ 'f' \ (x_n \ 'f' \ z)\ldots)
\]

Tail recursive, second parameter accumulator:

\[
\text{foldl } f \ z \ [] = z \\
\text{foldl } f \ z \ (x:xs) = \text{foldl } (f \ z \ x) \ xs \\
\text{foldl } f \ z \ [x_1, \ldots, x_n] = (\ldots(z \ 'f' \ x_1) \ 'f' \ \ldots) \ 'f' \ x_n
\]

Relationship between foldr and foldl:

**Lemma** \( \text{foldl } f \ e = \text{foldr } f \ e \)

if \( f \) is associative and \( e \ 'f' \ x = x \ 'f' \ e \).

**Proof** by induction over \( xs \).
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: \( \text{map } g \ . \ \text{map } f = \text{map } (g \ . \ f) \)

Another example: \( \text{sum } [n..m] \)
Precompute expensive computations

search :: String -> String -> Bool
search text s =
    table_search (hash_table text) (hash s, s)

bsearch = search bible

> map bsearch ["Moses", "Goethe"]

Better:

search text = \s -> table_search ht (hash s, s)
    where ht = hash_table text

Strong hint for compiler
Lazy evaluation

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

Example: $\text{length}^2$ under lazy evaluation

In general: tail recursion not always better under 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!
15. Case Study: Parsing

Basic Parsing

Application: Parsing pico-Haskell expressions

Improved Parsing
15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

"a+b*c" $\mapsto$

```
  +
 / \
 a  *
 /  |
 b  c
```
type Parser = String -> Tree

type Parser a = String -> a

What if something is left over, e.g., "a+b*c#" ?

type Parser a = String -> (a,String)

What if there is a syntax error, e.g., "++" ?

type Parser a = String -> [(a,String)]

[] syntax error
[x] one result x
[x,y,...] multiple results, ambiguous language
Alternative parser type

For unambiguous languages:

type Parser a = String -> Maybe (a,String)
Basic parsers

one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)

Example
char 'a' "abc" = [('a','bc')]
char 'b' "abc" = []
Combining parsers

Parse anything that \( p_1 \) or \( p_2 \) can parse:

\[
(|||) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a
\]

\[
p_1 \triangledown p_2 = \backslash cs \rightarrow p_1 \, cs \text{ ++ } p_2 \, cs
\]

Example

\[
(\text{char 'b'} \triangledown \text{char 'a'}) \text{"abc"} = [\text{(‘a’,"bc")}]
\]
Combining parsers

Parse first with p1, then the remainder with p2:

\[
(***) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a,b)
\]

\[
(p1 *** p2) \text{ xs } =

\[
[((a,b),zs) | (a,ys) \leftarrow p1 \text{ xs, (b,zs) } \leftarrow \text{ p2 ys}]
\]

Example

(char 'b' *** char 'a') "bac" = [((’b’,’a’), "c")]

(one isAlpha *** one isDigit *** one isDigit) "a12" = [((’a’,(’1’,’2’)), "")]

Transforming the result

Parse with \( p \), transform result with \( f \):

\[
(\ggg) :: \text{Parser}\ a \rightarrow (a \rightarrow b) \rightarrow \text{Parser}\ b
\]

\[
p \ggg f = \lambda xs \rightarrow \{ (f \ a, ys) \mid (a, ys) \leftarrow p\ xs \}\]

Example

\[
((\text{char } 'b' \ *** \ \text{char } 'a') \ggg (\lambda (x,y) \rightarrow [x,y]))\ "bac"
= \{("ba", "c")\]

raw text for this paragraph is: Transforming the result

Parse with \( p \), transform result with \( f \):

\[
(\ggg) :: \text{Parser}\ a \rightarrow (a \rightarrow b) \rightarrow \text{Parser}\ b
\]

\[
p \ggg f = \lambda xs \rightarrow \{ (f \ a, ys) \mid (a, ys) \leftarrow p\ xs \}\]

Example

\[
((\text{char } 'b' \ *** \ \text{char } 'a') \ggg (\lambda (x,y) \rightarrow [x,y]))\ "bac"
= \{("ba", "c")\]

Parsing a list of objects

Auxiliary functions:

\[ \text{uncurry} :: (a \to b \to c) \to (a,b) \to c \]
\[ \text{uncurry } f (a,b) = f \ a \ b \]

\[ \text{success} :: a \to \text{Parser } a \]
\[ \text{success } a \ xs = [(a, xs)] \]

The parser transformer:

\[ \text{list} :: \text{Parser } a \to \text{Parser } [a] \]
\[ \text{list } p = (p \ *** \ \text{list } p) \gg \gg \text{uncurry } (:) \]
\[ \quad \quad ||| \quad \text{success } [] \]

Example

\[ \text{list } (\text{one } \text{isAlpha}) \ "ab1" \]
\[ = [("ab", "1"), ("a", "b1"), ("", "ab1")]] \]
Parsing a non-empty list of objects

\[
\text{list1} :: \text{Parser } a \rightarrow \text{Parser } [a] \\
\text{list1 } p = (p \text{ *** list } p) \gg \gg \text{ uncurry } ()
\]
Parsing identifiers

ident :: Parser String
ident = (list1(one isAlpha) *** list(one isDigit))
    >>>> uncurry (++)

Example
ident "ab0" = [("ab0",""), ("ab","0"), ("a","b0")]

Handling spaces

spaces :: Parser String
spaces = list (one isSpace)

sp :: Parser a -> Parser a
sp p = (spaces *** p) >>> snd

Example
(sp ident) " ab c" = [("ab", " c"), ("a", "b c")]

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

\[
expr ::= \text{identifier} \\
\quad | \ ( expr \ expr ) \\
\quad | \ ( \ \text{\textbackslash \ identifier} . \ expr )
\]

Examples \ a, \ (f \ x), \ (\text{\textbackslash x}. \ (f \ x))

The tree representation:

\text{data Expr = Id String | App Expr Expr | Lam String Expr}

Examples \ \text{Id "a"} \\
\quad \text{App (Id "f") (Id "x")} \\
\quad \text{Lam "x" (App (Id "f") (Id "x")})
Pico-Haskell parser

\[
\text{ch c = sp (char c)}
\]
\[
\text{id = sp ident}
\]
\[
\text{expr =}
\]
\[
\text{id \gggg Id}
\]
\[
\text{(|||}
\]
\[
\text{(ch '(' *** expr *** expr *** ch ')')}
\]
\[
\gggg (\{_,(e1,(e2,_))\}) \rightarrow \text{App e1 e2}
\]
\[
\text{(|||}
\]
\[
\text{(ch '(' *** ch '\' *** id *** ch '.' *** expr *** ch ')')}
\]
\[
\gggg (\{_,(_,x,(e,_)))\) \rightarrow \text{Lam x e}
\]
15.3 Improved Parsing

String $\xrightarrow{\text{Lexer}}$ [Token] $\xrightarrow{\text{Parser}}$ Tree

Example

data Token =
    LParant | RParant | BSlash | Dot | Ident String
"(\x1 . x2)" $\xrightarrow{\text{Lexer}}$
    [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]

Why?

- Lexer based on regular expressions
  $\implies$ lexer can be more efficient than general parser
- Lexer can already remove spaces and comments
  $\implies$ simplifies parsing
Generalizing the implementation

So far:

```haskell
type Parser a = String -> [(a,String)]
```

Now:

```haskell
type Parser a b = [a] -> [(b,[a])]  
```

None of the parser combinators `***`, `|||`, `>>>` change, only their types become more general!

So far:

```haskell
(***) :: Parser a -> Parser b -> Parser (a,b)
```

Now:

```haskell
(***) :: Parser a b -> Parser a c -> Parser a (b,c)
```
Some literature:

- Chapter 8 of Hutton’s *Programming in Haskell*
- Section 17.5 in Thompson’s Haskell book (3rd edition)
- Many papers on functional parsers