Contents of the course

I Introduction to functional programming

1. Functional programming: Ideas, results, history, future
2. Types and functions
3. Case analysis, local definitions, recursion
4. Polymorphism and higher order functions
5. Structured types: Lists and user defined data types
6. Classes
7. Modules
8. Abstract data types
II Graphic programming in Haskell

• Basic I/O

• Simple graphics

• Monads

• Time permitting also: Animation, Robotics, Channels, Handles, Reactive systems

This course is partly based on material from a course given by Christian Lüth (University of Bremen) in 2002/2003.
Recommended Books


Lab classes

- Monday 2pm, Wednesday 10am, room 217. **Starts today!**
- Get password via email
- Course web page
  http://www-compsci.swan.ac.uk/~csulrich/cs-221.html
  Contains downloading information, web links to Haskell documentation, hints on how to use Haskell under Linux and Windows, lecture slides, coursework, e.t.c.
- Organizer: Monika Seisenberger csmona@swan.ac.uk
Coursework

- Coursework counts 20% towards the module.
- 5 small courseworks, 2% each, over the first 5 weeks.
- Then 1 big coursework, 10%.
- Unless stated otherwise, hand in printout of program with name, date and no. of coursework printed.
1 Functional Programming: Ideas, Results, History, Future
1.1 Ideas

- **Programs as functions**  \( f : \text{Input} \rightarrow \text{Output} \)
  - No variables — no states — no side effects
  - All dependencies explicit
  - Output depends on inputs only, not on environment

- **Abstraction**
  - Data abstraction
  - Function abstraction
  - Modularisation and Decomposition

- **Specification and verification**
  - Typing
  - Clear denotational and operational semantics
### Comparison with other programming paradigms

<table>
<thead>
<tr>
<th>Programming Paradigm</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative Programming</td>
<td>state</td>
<td>command</td>
</tr>
<tr>
<td>Logic Programming</td>
<td>specification</td>
<td>search</td>
</tr>
<tr>
<td>Functional Programming</td>
<td>expression</td>
<td>evaluation</td>
</tr>
</tbody>
</table>
1.2 Results

The main current functional languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Types</th>
<th>Evaluation</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp, Scheme</td>
<td>type free</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td>ML, CAML</td>
<td>polymorphic</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td>Haskell, Gofer</td>
<td>polymorphic</td>
<td>lazy</td>
<td>via monads</td>
</tr>
</tbody>
</table>
Haskell, the language we use in this course, is a purely functional language. Lisp and ML are not purely functional because they allow for programs with side effects.

Haskell is named after the American Mathematician Haskell B Curry (1900 – 1982).

Picture on next page copied from

http://www-groups.dcs.st-and.ac.uk/~history
1.2 Results
Some areas where functional languages are applied

• Artificial Intelligence
• Scientific computation
• Theorem proving
• Program verification
• Safety critical systems
• Web programming
• Network toolkits and applications
1.2 Results

• XML parser

• Natural Language processing and speech recognition

• Data bases

• Telecommunication

• Graphic programming

• Games

http://www.research.avayalabs.com/user/wadler/realworld
1.2 Results

Productivity and security

- Functional programming is very often more **productive** and **reliable** than imperative programming.

- Ericsson measured an improvement factor of between 9 and 25 in experiments on telephony software.

- Because of their **modularity**, **transparency** and high level of abstraction, functional programs are particularly easy to maintain and adapt.

- See [http://www.haskell.org/aboutHaskell.html](http://www.haskell.org/aboutHaskell.html)
Quicksort in Haskell

For comparison, here is the well-known quicksort program written in Haskell

\[
\text{qsort} \; \emptyset = \emptyset \\
\text{qsort} \; (x:xs) = \text{qsort} \; \text{ltx} \; ++ \; [x] \; ++ \; \text{qsort} \; \text{greqx}
\]

where

\[
\text{ltx} = \{ y \mid y \leftarrow xs, \; y < x \} \\
\text{greqx} = \{ y \mid y \leftarrow xs, \; y \geq x \}
\]

... and on the next two slides written in C.
Quicksort in C

```c
qsort(a, lo, hi) int a[], hi, lo;
{
    int h, l, p, t;
    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];
        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
```
t = a[l];
a[l] = a[h];
a[h] = t;
}
} while (l < h);

t = a[l];
a[l] = a[hi];
a[hi] = t;

qsort( a, lo, l-1 );
qsort( a, l+1, hi );
} }
1.3 History

• **Foundations 1920/30**
  ◦ Combinatory Logic and $\lambda$-calculus (Schönfinkel, Curry, Church)

• **First functional languages 1960**
  ◦ LISP (McCarthy), ISWIM (Landin)

• **Further functional languages 1970–80**
  ◦ FP (Backus); ML (Milner, Gordon), later SML and CAML; Hope (Burstall); Miranda (Turner)

• **1990: Haskell**
1.4 Future

• Functional programming more and more widespread

• Functional and object-oriented programming combined (Pizza, Generic Java)

• Extensions by dependent types (Chayenne)

• Big companies begin to adopt functional programming

• Microsoft initiative: F# = CAML into .net
2 Types and functions
Content

• How to define a function
• How to run a function
• Some basic types and functions
• Pairs and pattern matching
• Infix operators
• Computation by reduction
2.1 How to define a function

• Example

\[
\text{inc :: Int} \rightarrow \text{Int} \\
\text{inc x = x + 1}
\]
2.1 How to define a function

- **Example**
  
  \[
  \text{inc} :: \text{Int} \rightarrow \text{Int} \\
  \text{inc} \ x = x + 1
  \]

- **Explanation**
  
  - \(\text{inc} :: \text{Int} \rightarrow \text{Int}\) is the *signature* declaring \(\text{inc}\) as a function expecting an integer as input and computing an integer as output.
2.1 How to define a function

- **Example**

  \[
  \text{inc} :: \text{Int} \rightarrow \text{Int} \\
  \text{inc} \ x = x + 1
  \]

- **Explanation**
  - \text{inc} :: \text{Int} \rightarrow \text{Int} is the \textit{signature} declaring \text{inc} as a function expecting an integer as input and computing an integer as output.
  - \text{inc} \ x = x + 1 is the \textit{definition} saying that the function \text{inc} computes for any integer \( x \) the integer \( x + 1 \).
2.1 How to define a function

• Example

```haskell
inc :: Int -> Int
inc x = x + 1
```

• Explanation

○ `inc :: Int -> Int` is the *signature* declaring `inc` as a function expecting an integer as input and computing an integer as output.

○ `inc x = x + 1` is the *definition* saying that the function `inc` computes for any integer `x` the integer `x + 1`.

○ The symbol `x` is called a *formal parameter*, `Int` is the Haskell *type* of (small) integers.
2.1 How to define a function

- **Naming conventions:**
  - Functions and formal parameters begin with a *lower case* letter.
  - Types begin with an *upper case* letter.
2.1 How to define a function

- **Naming conventions:**
  - Functions and formal parameters begin with a **lower case** letter.
  - Types begin with an **upper case** letter.

- Don’t confuse the **definition**
  \[
  \text{inc } x = x + 1
  \]
  with the **assignment**
  \[
  x := x + 1
  \]
  in imperative languages.
Example of a function expecting two arguments.

addDouble :: Int -> Int -> Int
addDouble x y = 2 * (x + y)
2.1 How to define a function

- Example of a function expecting two arguments.

```haskell
addDouble :: Int -> Int -> Int
addDouble x y = 2 * (x + y)
```

- A combination of inc and addDouble

```haskell
f :: Int -> Int -> Int
f x y = inc (addDouble x y)
```
Why types?

• Early detection of errors at compile time

• Compiler can use type information to improve efficiency

• Type signatures facilitate program development

• and make programs more readable

• Types increase productivity and security
2.2 How to run a function

- **hugs** is a Haskell interpreter
  - Small, fast compilation (execution moderate)
  - Good environment for program development

- How it works
  - **hugs** reads *definitions* (programs, types, ...) from a file (*script*)
  - Command line mode: Evaluation of *expressions*
  - No definitions in command line
A hugs session

We assume that our example programs are written in a file hugsdemo1.hs (extension .hs required). After typing hugs

in a command window in the same directory where our file is, we can run the following session (black = hugs, red = we)
Prelude>
Prelude> :l hugsdemo1.hs
Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main>
2.2 How to run a function

Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
2.2 How to run a function

Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main>
Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main> f 2 3
2.2 How to run a function

Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main> f 2 3
11
Main>
2.2 How to run a function

Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main> f 2 3
11
Main> f (f 2 3) 6
Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main> f 2 3
11
Main> f (f 2 3) 6
35
Main>
Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> addDouble 2 3
10
Main> f 2 3
11
Main> f (f 2 3) 6
35
Main> :q

MMISS: Running functions Michaelmas Term 2003
At the beginning of the session hugs loads a file *Prelude.hs* which contains a bunch of definitions of Haskell types and functions. A copy of that file is available at our CS-221 page. It can be quite useful as a reference.
By typing `:?` (in hugs) one obtains a list of all available commands. Useful commands are:

- `:load <filename>`
- `:reload`
- `:type <Haskell expression>`
- `:quit`

All commands can be abbreviated by their first letter.
-- That’s how we write short comments

{-
Longer comments
can be included like this
-}
Exercise

• Define a function `square` that computes the square of an integer (don’t forget the signature).

• Use `square` to define a function `p16` which raises an integer to its 16th power.

• Use `p16` to compute the 16th powers of some numbers between 1 and 10. What do you observe? Try to explain.
2.3 Some basic types and functions

We now discuss the basic Haskell types

- Boolean values
- Numeric types: Integers and Floating point numbers
- Characters and Strings
Bolean values: Bool

- **Values** True and False

- **Predefined functions:**
  
  - `not :: Bool -> Bool`  
    negation
  
  - `&& :: Bool -> Bool -> Bool`  
    conjunction
  
  - `|| :: Bool -> Bool -> Bool`  
    disjunction

- **Example:** exclusive disjunction:

  ```haskell
  exOr :: Bool -> Bool -> Bool
  exOr x y = (x || y) && (not (x && y))
  ```
Basic numeric types

Computing with numbers

Limited precision \hspace{1cm} \leftrightarrow \hspace{1cm} \text{arbitrary precision}

constant cost \hspace{1cm} \leftrightarrow \hspace{1cm} \text{increasing cost}
Basic numeric types

Computing with numbers

Limited precision     arbitrary precision
constant cost          increasing cost

Haskell offers:

- **Int** - integers as machine words
- **Integer** - arbitrarily large integers
- **Rational** - arbitrarily precise rational numbers
- **Float** - floating point numbers
- **Double** - double precision floating point numbers
**Integers: Int and Integer**

- Some predefined functions (overloaded, also for `Integer`):
  
  \[ +, *, ^, - :: \text{Int} \to \text{Int} \to \text{Int} \]

  \[ \text{abs} :: \text{Int} \to \text{Int} -- \text{absolute value} \]

  \[ \text{div} :: \text{Int} \to \text{Int} \to \text{Int} \]

  \[ \text{mod} :: \text{Int} \to \text{Int} \to \text{Int} \]

- Comparison operators `==`, `/=`, `<=`, `<`, . . .

- **Note:** Unary minus
  
  - Different from infix operator `-`
  
  - Set brackets if in doubt: `abs (-34)`
2.3 Some basic types and functions

Floating point numbers: Float, Double

- Double precision Floating point numbers (IEEE 754 and 854)
  - Common arithmetic operations including division
    \[ / :: \text{float} \to \text{float} \to \text{float} \]
  - Logarithms, roots, exponentation, \( \pi \) (written \( \text{pi} \)) and \( e \), trigonometric functions

- conversion from and to integers:
  - \( \text{fromInt} :: \text{Int} \to \text{Double} \)
  - \( \text{fromInteger} :: \text{Integer} \to \text{Double} \)
  - \( \text{round}, \text{truncate} :: \text{Double} \to \text{Int} \) (\( \text{Integer} \))
  - Need signature to resolve overloading
Example

\[
\text{half :: Int} \rightarrow \text{Float}
\]
\[
\text{half } x = x \div 2
\]
Example

```haskell
half :: Int -> Float
half x = x / 2
```

Doesn’t work because division (/) expects two floating point numbers as arguments, but \(x\) has type \(\text{Int}\).
Example

```haskell
half :: Int -> Float
half x = x / 2
```

Doesn’t work because division (/) expects two floating point numbers as arguments, but \( x \) has type \texttt{Int}.

Solution:

```haskell
half :: Int -> Float
half x = (fromInt x) / 2
```
Characters and strings: Char, String

- Notation for characters: ’a’, . . .
- Some predefined functions:
  
  \[
  \begin{align*}
  \text{ord} & : \text{Char} \rightarrow \text{Int} \\
  \text{chr} & : \text{Int} \rightarrow \text{Char} \\
  \text{toLower} & : \text{Char} \rightarrow \text{Char} \\
  \text{toUpper} & : \text{Char} \rightarrow \text{Char} \\
  \text{isDigit} & : \text{Char} \rightarrow \text{Bool} \\
  \text{isAlpha} & : \text{Char} \rightarrow \text{Bool} \\
  : & : \text{Char} \rightarrow \text{String} \rightarrow \text{String} \quad -- \text{prefixing} \\
  ++ & : \text{String} \rightarrow \text{String} \rightarrow \text{String} \quad -- \text{concatenation}
  \end{align*}
\]
Example

rep :: String -> String
rep s = s ++ s

The expression

rep (rep "hello ")

evaluates to
Example

\[
\text{rep} :: \text{String} \rightarrow \text{String} \\
\text{rep } s = s ++ s
\]

The expression

\[
\text{rep (rep "hello ")}
\]

evaluates to

"hello hello hello hello hello hello "

If \( a \) and \( b \) are types then \((a, b)\) denotes the **cartesian product** of \( a \) and \( b \).

The elements of \((a, b)\) are all pairs \((x, y)\) where \(x\) is in \( a \) and \(y\) is in \( b \). In Prelude.hs the projection functions are defined by **pattern matching** as follows:

\[
\begin{align*}
\text{fst} & \quad :: (a, b) \to a \\
\text{fst} (x, _) & \quad = x \\
\text{snd} & \quad :: (a, b) \to b \\
\text{snd} (_, y) & \quad = y
\end{align*}
\]
2.5 Infix operators

- **Operators**: Names from special symbols !$%&/\?+^ . . .
- are written **infix**: \texttt{x \&\& y}
- otherwise they are normal functions.
- Using other functions **infix**:
  - \texttt{x ‘exOr‘ y}
  - \texttt{x ‘addDouble‘ y}
  - Note the difference: ‘exOr‘ ‘a’
- **Operators in prefix notation**:
  - \texttt{(&& x y}
  - \texttt{(+) 3 4}
2.6 Computation by reduction

- Recall our examples.
  
  \[
  \text{inc } x = x + 1 \\
  \text{addDouble } x \ y = 2 \times (x + y) \\
  f \ x \ y = \text{inc} (\text{addDouble } x \ y)
  \]

- Expressions are evaluated by reduction.

  \text{addDouble } 6 \ 4
2.6 Computation by reduction

- Recall our examples.

  \[
  \begin{align*}
  \text{inc } x &= x + 1 \\
  \text{addDouble } x \ y &= 2 \ast (x + y) \\
  f \ x \ y &= \text{inc} (\text{addDouble } x \ y)
  \end{align*}
  \]

- Expressions are evaluated by reduction.

  \[
  \text{addDouble } 6 \ 4 \to 2 \ast (6 + 4)
  \]
• Recall our examples.

\[
\begin{align*}
inc\ x &= x + 1 \\
addDouble\ x\ y &= 2 \times (x + y) \\
f\ x\ y &= inc\ (addDouble\ x\ y)
\end{align*}
\]

• Expressions are evaluated by \textit{reduction}.

\[
addDouble\ 6\ 4 \rightarrow 2 \times (6 + 4) \rightarrow 20
\]
Evaluation strategy

- From outside to inside, from left to right.

\[ f (\text{inc } 3) 4 \]
Evaluation strategy

- From outside to inside, from left to right.

\[
f \left( \text{inc} \ 3 \right) 4 \\
\rightarrow \text{inc} \left( \text{addDouble} \left( \text{inc} \ 3 \right) 4 \right) \\
\rightarrow
\]
Evaluation strategy

- From outside to inside, from left to right.

\[
\begin{align*}
f \text{(inc 3) 4} & \\
\leadsto \text{inc (addDouble (inc 3) 4)} & \\
\leadsto (\text{addDouble (inc 3) 4}) + 1 & \\
\leadsto & \\
\end{align*}
\]
Evaluation strategy

- From **outside** to **inside**, from **left** to **right**.

\[
f(\text{inc } 3) 4 \\
\leadsto \text{inc } (\text{addDouble } (\text{inc } 3) 4) \\
\leadsto (\text{addDouble } (\text{inc } 3) 4) + 1 \\
\leadsto 2*(\text{inc } 3 + 4) + 1 \\
\leadsto
\]
Evaluation strategy

- From outside to inside, from left to right.

\[
f \text{(inc 3) 4} \\
\leadsto \text{inc \ (addDouble \ (inc 3) 4)} \\
\leadsto (\text{addDouble \ (inc 3) 4}) + 1 \\
\leadsto 2*(\text{inc 3 + 4}) + 1 \\
\leadsto 2*((3 + 1) + 4) + 1 \leadsto
\]
Evaluation strategy

- From **outside** to **inside**, from **left** to **right**.

\[
\begin{align*}
f (\text{inc } 3) & 4 \\
\rightarrow & \text{inc} (\text{addDouble} (\text{inc } 3) 4) \\
\rightarrow & (\text{addDouble} (\text{inc } 3) 4) + 1 \\
\rightarrow & 2*(\text{inc } 3 + 4) + 1 \\
\rightarrow & 2*((3 + 1) + 4) + 1 \rightarrow 17
\end{align*}
\]

- **call-by-need** or **lazy evaluation**
  - Arguments are calculated only when they are needed.
  - Lazy evaluation is useful for computing with infinite data structures.
rep (rep "hello ")

~~>
rep (rep "hello ")
⇝ rep "hello" ++ rep "hello"
⇝
\[
\text{rep (rep "hello ")}
\leadsto \text{rep "hello" ++ rep "hello"}
\leadsto ("hello " ++ "hello ") ++ ("hello " ++ "hello ")
\leadsto 
\]
rep (rep "hello ")
⇒ rep "hello" ++ rep "hello"
⇒ ("hello " ++ "hello ") ++ ("hello " ++ "hello ")
⇒ "hello hello hello hellohello"
3 Case analysis, local definitions, recursion
Content

- Forms of case analysis: if-then-else, guarded equations
- Local definitions: where, let
- Recursion
- Layout
3.1 Case analysis

• If-then-else

\[
\text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{max} \ x \ y = \text{if} \ x < y \ \text{then} \ y \ \text{else} \ x
\]

• Guarded equations

\[
\text{signum} :: \text{Int} \rightarrow \text{Int} \\
\text{signum} \ x \\
| \ x < 0 \ = \ -1 \\
| \ x \ == \ 0 \ = \ 0 \\
| \ x \ > \ 0 \ = \ 1 \ -- \ or \ | \ \text{otherwise} \ = \ 1
\]
3.2 Local definitions

- **let**

  \[
  g :: \text{Float} \to \text{Float} \to \text{Float} \\
  g \ x \ y = (x^2 + y^2) / (x^2 + y^2 + 1)
  \]

  **better**

  \[
  g \ x \ y = \text{let} \ a = x^2 + y^2 \\
  \quad \text{in} \ a / (a + 1)
  \]
3.2 Local definitions

- where

\[ g :: \text{Float} \to \text{Float} \to \text{Float} \]
\[ g \ x \ y = a \div (a + 1) \quad \text{where} \]
\[ a = x^2 + y^2 \]

or

\[ g \ x \ y = a \div b \quad \text{where} \]
\[ a = x^2 + y^2 \]
\[ b = a + 1 \]

or

\[ g \ x \ y = \text{let} \ a = x^2 + y^2 \]
\[ \quad b = a + 1 \]
\[ \quad \text{in} \quad a \div b \]
• Local definition of functions

The sum of the areas of two circles with radii \( r, s \).

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{totalArea} \; r \; s = \pi \; \times \; r^2 \; + \; \pi \; \times \; s^2
\]

Use auxiliary function to compute the area of one circle

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{totalArea} \; r \; s = \\
\text{let} \; \text{circleArea} \; r = \pi \; \times \; r^2 \\
\text{in} \; \text{circleArea} \; r \; + \; \text{circleArea} \; s
\]

• Exercise: Use \textit{where} instead.
3.3 Recursion

- **Recursion** = defining a function in terms of itself.

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

Doesn’t terminate if \( n \) is negative. Therefore

```haskell
fact :: Int -> Int
fact n
  | n < 0    = error "negative argument to fact"
  | n == 0   = 1
  | n > 0    = n * fact (n - 1)
```
fib :: Int -> Int

fib n
  | n < 0       = error "negative argument to fib"
  | n == 0      = 1
  | n == 1      = 1
  | n > 0       = fib (n - 1) + fib (n - 2)

Exponential run time. Better:
fib :: Int -> Int
fib n = fst (fibpair n) where
  fibpair n
  | n < 0     = error "negative argument to fib"
  | n == 0    = (1,1)
  | n > 0     = let (k,l) = fibpair (n - 1)
                in (l,k+l)
3.4 Layout

Due to an elaborate layout Haskell programs do not need many brackets and are therefore well readable. The layout rules are rather intuitive:

• Definitions start at the beginning of a line
• The body of a definition must be indented against the function defined,
• lists of equations and other special constructs must properly line up.
4 Polymorphism and higher order functions
Contents

- Function types
- Higher order functions
- Polymorphic functions
- Lambda abstraction
4.1 Function types

Consider the function

\[
\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{add } x \; y = x + y
\]

The signature of \text{add} is shorthand for

\[
\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

We may, for example, define

\[
\text{addFive} :: \text{Int} \rightarrow \text{Int} \\
\text{addFive} = \text{add } 5
\]

\[
\text{addFive } 7 \rightarrow (\text{add } 5) \; 7 = \text{add } 5 \; 7 \rightarrow 5+7 \rightarrow 12
\]
• Int -> Int is a function type.

• Function types are ordinary types,

• they may occur as result types (like in Int -> (Int -> Int)),

• but also as argument types:

    twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
4.1 Function types

- \textbf{Int} \to \textbf{Int} is a function type.

- Function types are ordinary types,

- they may occur as result types (like in \textbf{Int} \to (\textbf{Int} \to \textbf{Int}))

- but also as argument types:

  \begin{verbatim}
  twice :: (Int -> Int) -> Int -> Int
  twice f x = f (f x)
  \end{verbatim}

\[
twice \text{ addFive } 7 \quad \Rightarrow \quad 16
\]
• **Int -> Int** is a function type.

• Function types are ordinary types,

• they may occur as result types (like in **Int -> (Int -> Int)**),

• but also as argument types:

  \[
  \text{twice} :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \\
  \text{twice} \ f \ x = f (f \ x)
  \]

  \[
  \text{twice addFive 7} \leadsto \text{addFive (addFive 7)} \leadsto
  \]
• **Int -> Int** is a function type.

• Function types are ordinary types,

• they may occur as result types (like in **Int -> (Int -> Int)**),

• but also as argument types:

\[
\text{twice} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{twice} \ f \ x = f (f \ x)
\]

\[
\text{twice addFive} \ 7 \ ⇝ \ \text{addFive} \ (\text{addFive} \ 7) \ ⇝ \ 17
\]
• `type1 -> type2 -> ... -> typen -> type`

  is shorthand for

  `type1 -> (type2 -> ... -> (typen -> type)...)`

• `f x1 x2 ... xn`

  is shorthand for

  `(...(((f x1) x2) ... ) xn`
Sections

If we apply a function of type, say

\[ f :: \text{type1} \to \text{type2} \to \text{type3} \to \text{type4} \to \text{type5} \]

to arguments, say, \( x_1 \) and \( x_2 \), then

\( x_1 \) must have type \( \text{type1} \), \( x_2 \) must have type \( \text{type2} \).

and we get

\[ f \ x_1 \ x_2 :: \text{type3} \to \text{type4} \to \text{type5} \]

The expression \( f \ x_1 \ x_2 \) is called a section of \( f \).

For example \( \text{add 5} \) is a section of \( \text{add} \).
4.1 Function types

- Applying a function $n$-times:

```haskell
iter :: Int -> (Int -> Int) -> Int -> Int
iter n f x
  | n == 0      = x
  | n > 0       = f (iter (n-1) f x)
  | otherwise   = error "negative argument to iter"
```
Applying a function \( n \)-times:

\[
\begin{align*}
\text{iter} & :: \text{Int} \to (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \\
\text{iter} \ n \ f \ x \\
& \mid n == 0 \quad = x \\
& \mid n > 0 \quad = f (\text{iter} (n-1) f x) \\
& \mid \text{otherwise} \quad = \text{error} \ "\text{negative argument to iter}" \\
\end{align*}
\]

iter 10 addFive 7 \( \leadsto \ldots \leadsto \) 57
• If a function has a function type as argument type, then it is called a **higher-order function**, 

• otherwise it is called a **first-order function**.

• *twice* and *iter* are higher-order functions whereas *add* and the functions we discussed in previous chapters were first-order functions.
4.2 Polymorphism

Another important difference between the functions

\[
\text{add} :: \text{Int} \to \text{Int} \to \text{Int}
\]
\[
\text{add} \ x \ y = x + y
\]

and

\[
\text{twice} :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})
\]
\[
\text{twice} \ f \ x = f (f \ x)
\]
is that while in the signature of \text{add} the type \text{Int} was special (we could not have replaced it by, say, \text{Char}), in the signature of \text{twice} the type \text{Int} was irrelevant; we could have replaced it by any other type.
• In Haskell we can express the latter fact by assigning `twice` a polymorphic type, that is, a type that contains type variables:

```haskell
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

`twice` then becomes a polymorphic function.

• Type variable begin with a lower case letter.

• Polymorphic functions can be used in any context where the type variables can be substituted by suitable types such that the whole expression is well typed.
Exercise

Compute the value of the expressions

\begin{itemize}
\item \texttt{twice square 2}
\item \texttt{twice twice square 2}
\item \texttt{(twice square) 2}
\end{itemize}
Exercise

Compute the value of the expressions

- \( \text{twice square } 2 \)
- \( \text{twice twice square } 2 \)

\( (\text{twice square}) \, 2 \rightarrow \text{square (square } 2) \)
Exercise

Compute the values of the expressions

\[
\text{twice square } 2
\]

\[
\text{twice twice square } 2
\]

\[
\text{(twice square) } 2 \quad \leadsto \quad \text{square (square } 2) \quad \leadsto \quad 16
\]
Exercise

Compute the value of the expressions

\[
\begin{align*}
twice \ square \ 2 & \quad \rightarrow \quad \square \ (\square \ 2) \quad \rightarrow \quad 16 \\
twice \ twice \ square \ 2 & \\
(twice \ square) \ 2 & \\
((twice \ twice) \ square) \ 2 & 
\end{align*}
\]
Exercise

Compute the value of the expressions

\[
\text{twice square } 2 \\
\text{twice twice square } 2
\]

\[
(\text{twice square}) 2 \quad \rightsquigarrow \quad \text{square (square } 2) \quad \rightsquigarrow \quad 16
\]

\[
((\text{twice twice}) \text{ square}) 2 \\
\rightsquigarrow \quad (\text{twice (twice square)}) 2
\]

MMISS: Polymorphism and higher order functions Michaelmas Term 2003
Exercise

Compute the value of the expressions

\[
\begin{align*}
&\text{twice square 2} \\
&\text{twice twice square 2} \\
&(\text{twice square}) 2 \quad \mapsto \quad \text{square (square 2)} \quad \mapsto \quad 16 \\
&((\text{twice twice}) \text{ square}) 2 \\
&\mapsto (\text{twice (twice square)}) 2) \\
&\mapsto (\text{twice square}) ((\text{twice square}) 2)
\end{align*}
\]
Exercise

Compute the value of the expressions

\[
\begin{align*}
\text{twice square } 2 & \\
\text{twice twice square } 2 & \\
\text{(twice square) } 2 & \Rightarrow \text{ square (square } 2) \Rightarrow 16 \\
\text{((twice twice) square) } 2 & \Rightarrow \text{ (twice (twice square)) } 2) \\
& \Rightarrow \text{ (twice square) ((twice square) } 2) \\
& \Rightarrow (16^2)^2 = 65536
\end{align*}
\]
Some predefined polymorphic functions

- Identity (more useful than one might think)

```haskell
id :: a -> a
id x = x
```
Some predefined polymorphic functions

- **Identity** (more useful than one might think)
  
  \[ id :: a \rightarrow a \]

  \[ id \ x = x \]

- **Projections**

  \[ fst :: (a,b) \rightarrow a \]

  \[ fst (x,_) = x \]

  \[ snd :: (a,b) \rightarrow a \]

  \[ snd (_,y) = y \]
• **Composition**

\[
(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)
\]

\[
(f \cdot g) x = f (g x)
\]
• **Composition**

\[
(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)
\]
\[
(f \cdot g) x = f (g x)
\]

• **Currying and Uncurrying**

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

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

Prove that the functions curry and uncurry are inverse to each other, that is

\[
\text{uncurry \ (curry \ f)} = f \\
\text{curry \ (uncurry \ g)} = g
\]

for all \( f :: (a, b) \to c \) and \( g :: a \to b \to c \).
Exercise

Prove that the functions \texttt{curry} and \texttt{uncurry} are inverse to each other, that is

\begin{align*}
\text{uncurry (curry } f) &= f \\
\text{curry (uncurry } g) &= g
\end{align*}

for all \( f :: (a,b) \to c \) and \( g :: a \to b \to c \).

We verify the equations for all possible arguments:

\begin{align*}
\text{uncurry (curry } f) \ (x,y) &= \
\end{align*}
Exercise

Prove that the functions `curry` and `uncurry` are inverse to each other, that is

\[
\text{uncurry } (\text{curry } f) = f \\
\text{curry } (\text{uncurry } g) = g
\]

for all \( f :: (a, b) \to c \) and \( g :: a \to b \to c \).

We verify the equations for all possible arguments:

\[
\text{uncurry } (\text{curry } f) (x, y) = \text{curry } f x y =
\]
Exercise

Prove that the functions \textit{curry} and \textit{uncurry} are inverse to each other, that is

\[
\text{uncurry} \ (\text{curry} \ f) = f \\
\text{curry} \ (\text{uncurry} \ g) = g
\]

for all \( f :: (a,b) \rightarrow c \) and \( g :: a \rightarrow b \rightarrow c \).

We verify the equations for all possible arguments:

\[
\text{uncurry} \ (\text{curry} \ f) \ (x,y) = \text{curry} \ f \ x \ y = f \ (x,y)
\]
Exercise

Prove that the functions \texttt{curry} and \texttt{uncurry} are inverse to each other, that is

\[
\text{uncurry (curry } f \text{)} = f \\
curry (\text{uncurry } g) = g
\]

for all \( f :: (a,b) \to c \) and \( g :: a \to b \to c \).

We verify the equations for all possible arguments:

\[
\text{uncurry (curry } f \text{) } (x,y) = \text{curry } f \ x \ y = f (x,y) \\
curry (\text{uncurry } g) \ x \ y =
\]
Exercise

Prove that the functions \texttt{curry} and \texttt{uncurry} are inverse to each other, that is

\[
\text{uncurry} \ (\text{curry} \ f) = f \\
\text{curry} \ (\text{uncurry} \ g) = g
\]

for all \( f : (a, b) \rightarrow c \) and \( g : a \rightarrow b \rightarrow c \).

We verify the equations for all possible arguments:

\[
\text{uncurry} \ (\text{curry} \ f) \ (x, y) = \text{curry} \ f \ x \ y = f \ (x, y) \\
\text{curry} \ (\text{uncurry} \ g) \ x \ y = \text{uncurry} \ g \ (x, y) =
\]
Exercise

Prove that the functions \texttt{curry} and \texttt{uncurry} are inverse to each other, that is

\[
\text{uncurry (curry } f) = f \\
\text{curry (uncurry } g) = g
\]

for all \( f :: (a,b) \to c \) and \( g :: a \to b \to c \).

We verify the equations for all possible arguments:

\[
\text{uncurry (curry } f) (x,y) = \text{curry } f x y = f (x,y) \\
\text{curry (uncurry } g) x y = \text{uncurry } g (x,y) = g x y
\]
Extensionality

Two functions are equal when they are equal at all arguments, that is,

\[ f = f' \text{ if and only if } f \, z = f' \, z \text{ for all } z. \]
• Until: Repeatedly applying an operation \( f \) to an initial value \( x \) until a property \( p \ x \) holds.

\[
\text{until} \quad :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]

\[
\text{until } p \ f \ x \ = \ \text{if } p \ x \ \text{then } x \ \text{else until } p \ f \ (f \ x)
\]
• Until: Repeatedly applying an operation \( f \) to an initial value \( x \) until a property \( p \ x \) holds.

\[
\text{until} :: (a -> \text{Bool}) -> (a -> a) -> a -> a
\]

\[
\text{until } p \ f \ x = \text{if } p \ x \ \text{then } x \ \text{else until } p \ f \ (f \ x)
\]

Example

For every function \( f :: \text{Integer} -> \text{Bool} \) find a number \( n \) such that \( f \ n \leq f \ (n+1) \).

\[
\text{nondecreasing} :: (\text{Int} -> \text{Int}) -> \text{Int}
\]

\[
\text{nondecreasing } f = \text{until } \text{nondec } \text{inc } 0 \ \text{where}
\]

\[
\text{nondec } n = f \ n \leq f \ (n+1)
\]

\[
\text{inc } n = n+1
\]
4.2 Polymorphism

\[
f :: \text{Int} \to \text{Int} \\
f \ x = x^2 - 6*x
\]

Main> nondecreasing f
?
Exercise

Compute for any function \( f :: \text{Int} \to \text{Int} \) and \( m \geq 0 \) the maximum of the values \( f \ 0, \ldots, f \ n \).

\[
\text{maxfun :: (Int \to Int) \to Int \to Int}
\]

\[
\text{maxfun \ f \ m}
\]

\[
| m < 0 \quad = \text{error "negative argument to maxfun"}
\]

\[
| m == 0 \quad = f \ 0
\]

\[
| m > 0 \quad = \text{max (maxfun \ f \ (m-1)) \ (f \ m)}
\]

Here we used the predefined function \( \text{max} \).
4.3 λ-Abstraction

Using \( \lambda \)-abstraction we can create anonymous functions, that is, functions without name. For example, instead of defining at top level the test function

\[
\begin{align*}
f &:: \text{Int} \rightarrow \text{Int} \\
f \ x &= x^2 - 6 \times x
\end{align*}
\]

(which is not of general interest) and then running

Main> nondecreasing \( f \)

we may simply run

Main> nondecreasing (\( \lambda x \rightarrow x^2 - 6 \times x \))
In general an expression
\[ x \to <\ldots> \]
is equivalent to
\[
\text{let } f \ x = <\ldots> \\
\text{in } f
\]
\(\lambda\)-abstraction is not needed, strictly speaking, but often very handy.
Examples of $\lambda$-abstraction

• $\lambda$-abstraction in definitions:

  \[
  \text{square} :: \text{Int} \rightarrow \text{Int} \\
  \text{square} = \lambda x \rightarrow x \times x
  \]

  (instead of \text{square} \ x = x \times x)

• Pattern matching in $\lambda$-abstraction:

  \[
  \text{swap} :: (a,b) \rightarrow (b,a) \\
  \text{swap} = \lambda (x,y) \rightarrow (y,x)
  \]

  (instead of \text{swap} \ (x,y) = (y,x))
• Avoiding local definitions by $\lambda$-abstraction (not always recommended):

\[
\text{nondecreasing} :: (\text{Int} \to \text{Int}) \to \text{Int}
\]
\[
\text{nondecreasing } f = \text{until } (\lambda n . f n \leq f (n+1)) (\lambda n . n+1) 0
\]

instead of

\[
\text{nondecreasing } f = \text{until } \text{nondec inc } 0 \text{ where}
\]
\[
\text{nondec } n = f n \leq f (n+1)
\]
\[
\text{inc } n = n+1
\]
• Higher-order \( \lambda \)-abstraction (that is, the abstracted parameter has a higher-order type):

\[
\text{quad} :: \text{Int} \rightarrow \text{Int} \\
\text{quad} = \text{twice} \ \text{square}
\]

can be defined alternatively by

\[
\text{quad} = (\lambda f \rightarrow \lambda x \rightarrow f (f \ x)) \ (\lambda x \rightarrow x\star x)
\]
The type of a $\lambda$-abstraction

If we have an expression of type

$$<\ldots> :: \text{type}_2$$

and the parameter $x$ has type $\text{type}_1$, then the lambda abstraction has type

$$\left(\lambda x \rightarrow <\ldots>\right) :: \text{type}_1 \rightarrow \text{type}_2$$
5 Structured types: Lists and user defined types
Contents

• Simple data

• Composite data: Geometric shapes

• Parametric data

• Recursive data: Lists

• Examples
5.1 Simple data

One of the simplest structured data type predefined in hugs’ Prelude is the type of boolean values:

\[
\text{data Bool} = \text{False} \mid \text{True} \\
\quad \text{deriving (Eq, Show)}
\]

This means:

- The data type \text{Bool} contains exactly the constructors \text{False} and \text{True}.

- Via ‘deriving(Eq, Show)’ the following default operations are available:
• **Eq:** Equality and inequality tests

\[ (==), (/=) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]

• **Show:** Used to print data on the terminal

\[ \text{show} :: \text{Bool} \rightarrow \text{String} \]
Pattern matching on constructors

(&&), (||) :: Bool -> Bool -> Bool
False && x = False
True && x = x
False || x = x
True || x = True

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

otherwise :: Bool
otherwise = True
Example: Days of the week

data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving(Eq, Ord, Enum, Show)

- **Ord**: Constructors are ordered from left to right

  \[(<), (\leq), (\geq), (>): \text{Day} \rightarrow \text{Day} \rightarrow \text{Bool}\]

  \[\text{max, min}: \text{Day} \rightarrow \text{Day} \rightarrow \text{Day}\]

- **Enum**: Enumeration and numbering

  \[\text{fromEnum}: \text{Day} \rightarrow \text{Int}\]

  \[\text{toEnum}: \text{Int} \rightarrow \text{Day}\]
Wed < Fri  ⇝  True
Wed < Wed  ⇝  False
Wed <= Wed  ⇝  True
max Wed Sat  ⇝  Sat
fromEnum Sun  ⇝  0
fromEnum Mon  ⇝  1
fromEnum Sat  ⇝  6
toEnum 2  ⇝  ERROR - Unresolved overloading ...
toEnum 2 :: Day  ⇝  Tue
toEnum 7 :: Day  ⇝  Program error ...
Pattern matching

Example: Testing whether a day is a work day

- Using several equations

```haskell
workDay :: Day -> Bool
workDay Sun = False
workDay Sat = False
workDay _ = True
```
• The same function using a new form of definition by cases:

```haskell
workDay :: Day -> Bool
workDay d = case d of
    Sun -> False
    Sat -> False
    _    -> True
```
• We can also use the derived ordering

\[
\text{workDay} :: \text{Day} -> \text{Bool}
\]
\[
\text{workDay} \ d = \text{Mon} \leq d \land d \leq \text{Fri}
\]

• or guarded equations

\[
\text{workDay} :: \text{Day} -> \text{Bool}
\]
\[
\text{workDay} \ d
\]
\[
| d =\text{Sat} \mid | | d =\text{Sun} \mid = \text{False}
\]
\[
|\text{otherwise} \mid = \text{True}
\]
Example: Computing the next day

- Using enumeration

```haskell
dayAfter :: Day -> Day
dayAfter d = toEnum ((fromEnum d + 1) 'mod' 7)
```

- The same with the composition operator

```haskell
dayAfter :: Day -> Day
dayAfter = toEnum . (‘mod’ 7) . inc . fromEnum
```

Exercise: Define `dayAfter` without `mod` using case analysis instead.
Summary

- **data** is a keyword for introducing a new data type with constructors.

- Names of data types and constructors begin with an upper case letter.

- Via `deriving(...)` various (overloaded) default operation can be introduced automatically.

- Definition by cases can be done via pattern matching on constructors.
5.2 Composite data: Geometric shapes

We use representations of simple geometric shapes as an example for composed data.

```
data Shape = Rectangle Float Float  
            | Circle Float  
            | RTriangle Float Float Float deriving Show
```

- **Rectangle s1 s2** is a rectangle with sides s1, s2.
- **Circle r** is a circle with radius r.
- **RTriangle s1 s2** is a right triangle with sides s1, s2 (the sides of the right angle).
Rectangle \( s_1 \ s_2 \) as a region in the plane
Circle \( r \) as a region in the plane
RTriangle s1 s2 as a region in the plane
Abstraction by type synonyms

We introduce type synonyms for the data type Float

type Radius = Float
type Side = Float

and redefine the data type of shapes

data Shape = Rectangle Side Side
            | Circle Radius
            | RTriangle Side Side deriving Show
The advantages of doing so are twofold:

- **Readability**: The intended use of the data can be read off from the code.

- **Abstraction**: If we later wish to replace the type `Float` by, say, `Double`, we need to do this only in the definitions of the types `Radius` and `Side`:

```
type Radius = Double
type Side   = Double
```
The type of a constructor

- In a data type declaration
  
  ```
  data DataType = Constr Type1 ... Typen | . . .
  ```
  
  the type of the constructor is
  
  ```
  Constr :: Type1 -> ... -> Typen -> DataType
  ```

- In our examples:

  ```
  True, False :: Bool
  Sun,...,Sat :: Day
  Rectangle, RTriangle :: Side -> Side -> Shape
  Circle :: Radius -> Shape
  ```
Areas of shapes

type Area = Float

area :: Shape -> Area
area (Rectangle s1 s2) = s1 * s2
area (Circle r) = pi * r^2
area (RTriangle s1 s2) = s1 * s2 / 2

Exercise: Use case ... of instead.
Exercise: Compute the perimeter of a shape.
Exercise: Compute the number of corners of a shape.
5.3 Polymorphic data

- A data type may be polymorphic, that is, depend on other data types in a parametric way.

- For example, the product type \((a, b)\) depends on the type parameters \(a\) and \(b\).

- Type parameters are also allowed in user defined types.
Example: Union

data Either a b = Left a | Right b
deriving (Eq, Ord, Show)

- The data type Either a b may be viewed as the union of the types a and b.

- The types of the constructors are
  
  Left :: a -> Either a b
  Right :: b -> Either a b

- Either itself can be viewed as a type constructor which constructs from given types a, b an new type Either a b.
Joining two functions with different argument types:

\[
\text{either} :: (a \to c) \to (b \to c) \to \text{Either } a b \to c
\]

\[
\text{either } f \ g \ (\text{Left } x) = f \ x
\]

\[
\text{either } f \ g \ (\text{Right } y) = g \ y
\]

The data type \text{Either } a b \text{ as well as the function } \text{either} \text{ are predefined in } \text{hugs'} \text{ Prelude.}
Example: Define a function `size` on the union of the type of strings and the type of integers. The size of a string shall be its length, the size of an integer shall be its absolute value.
Example: Define a function \texttt{size} on the union of the type of strings and the type of integers. The size of a string shall be its length, the size of an integer shall be its absolute value.

\begin{verbatim}
size :: Either String Int -> Int
size = either length abs
\end{verbatim}
Example: Define a function `size` on the union of the type of strings and the type of integers. The size of a string shall be its length, the size of an integer shall be its absolute value.

```haskell
size :: Either String Int -> Int
size = either length abs
```

```
size (Left ('Hello')) \to 5
```
Example: Define a function \texttt{size} on the union of the type of strings and the type of integers. The size of a string shall be its length, the size of an integer shall be its absolute value.

\[
\text{size} :: \text{Either String Int} \rightarrow \text{Int}
\]

\[
\text{size} = \text{either length abs}
\]

\[
\text{size (Left \text{‘‘Hello’’})} \rightarrow 5
\]

\[
\text{size (Right \text{(-3)})} \rightarrow 3
\]
Example: Define a function `size` on the union of the type of strings and the type of integers. The size of a string shall be its length, the size of an integer shall be its absolute value.

```haskell
size :: Either String Int -> Int
size = either length abs
```

```haskell
size (Left "Hello") ⇝ 5
size (Right (-3)) ⇝ 3
```

Alternatively

```haskell
size :: Either String Int -> Int
size (Left s) = length s
size (Right x) = abs x
```
5.4 Recursive data: Lists

The most common recursive data type is the type of polymorphic lists, \([a]\). One can think of \([a]\) as being defined by

\[
data [a] = [] | a : [a]
\]

This means that \([a]\) has the constructors

\[
[] :: [a] \quad -- \text{the empty list} \\
(·) :: a -> [a] -> [a] \quad -- \text{adding an element}
\]

Hence the elements of \([a]\) are either [] or of the form \(x:xs\) where \(x :: a\) and \(xs :: [a]\).
Notations for lists

Haskell allows the usual syntax for lists

\[ x_1, x_2, \ldots, x_n \] =
\[ x_1 : x_2 : \ldots : x_n : [] \] =
\[ x_1 : (x_2 : \ldots : (x_n : []) \ldots ) \]

Examples:

\[ [1,3,5+4] \] :: [Int]
\[ ['a','b','c','d'] \] :: [Char]
\[ [\text{square}, \text{inc}, \backslash x \rightarrow x^{10}] \] :: [Int -> Int]
\[ [[\text{True}, 3<2], []] \] :: [[Bool]]
\[ [(198845, "Cox"), (203187, "Wu")]] \] :: [[(Int, String)]]
Examples of functions on lists

• The sum of a list of integers

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

• Squaring all members of a list of integers

\[
\text{squareAll} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{squareAll} [] = [] \\
\text{squareAll} (x : xs) = \text{square} \; x : \text{squareAll} \; xs
\]
• The sum of squares of a list of integers

\[
\text{sumSquares} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sumSquares} \; \text{xs} = \text{sumList} \; (\text{squareAll} \; \text{xs})
\]

Alternatively

\[
\text{sumSquares} = \text{sumList} \; \circ \; \text{squareAll}
\]

<table>
<thead>
<tr>
<th>sumList ; [2,3,4]</th>
<th>\rightarrow</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>squareAll ; [2,3,4]</td>
<td>\rightarrow</td>
<td>[4,9,16]</td>
</tr>
<tr>
<td>sumSquares ; [2,3,4]</td>
<td>\rightarrow</td>
<td>29</td>
</tr>
</tbody>
</table>
Strings are lists of characters: \[ \text{String} = \text{[Char]} \]

"Hello" == ['H', 'e', 'l', 'l', 'o'] \[\mapsto\] True
Strings are lists of characters:  \[
\text{String} = [\text{Char}]
\]
"Hello" == ['H','e','l','l','o']  \implies  True

Example: Replacing a character by another

replace :: Char -> Char -> String -> String
replace old new "" = ""
replace old new (x : s) = y : replace old new s where
  y = if x == old then new else x
replace 'a' 'o' "Australia"  \implies  "Australolio"
More Examples:

- **Length of a list:**

  \[
  \text{length} :: [a] \rightarrow \text{Int} \\
  \text{length} [] = 0 \\
  \text{length} (x:xs) = 1 + \text{length} \hspace{1pt} xs
  \]

- **Flattening nested lists:**

  \[
  \text{concat} :: [[a]] \rightarrow [a] \\
  \text{concat} [] = [] \\
  \text{concat} (x:xs) = x ++ (\text{concat} \hspace{1pt} xs)
  \]
- Head and tail of a nonempty list:

\[
\text{head} :: [a] \rightarrow a \\
\text{head} (x:xs) = x \\
\text{tail} :: [a] \rightarrow [a] \\
\text{tail} (x:xs) = xs
\]

*Undefined* for the empty list.

- Taking the first \( n \) elements of a list:

\[
\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take} n \_ \mid n \leq 0 = [] \\
\text{take} \_ [] = [] \\
\text{take} n (x:xs) = x : \text{take} (n-1) xs
\]
Some predefined functions on lists

(+++) :: [a] -> [a] -> [a]  concatenating two lists
(!!) :: [a] -> Int -> a    selecting $n$-th element
concat :: [[a]] -> [a]     flattening
null :: [a] -> Bool       testing if empty list
length :: [a] -> Int      length of a list
head, last :: [a] -> a    first/last element
tail, init :: [a] -> [a]  removing first/last element
replicate :: Int -> a -> [a] $n$ copies
take :: Int -> [a] -> [a]  take first $n$ elements
drop :: Int -> [a] -> [a]  remove first $n$ elements
splitAt :: Int -> [a] -> ([a],[a]) split at $n$-th position
**reverse** :: [a] -> [a]  
reversing a list

**zip** :: [a] -> [b] -> [(a,b)]  
two lists into a list of pairs

**unzip** :: [(a, b)] -> ([a], [b])  
list of pairs into a pair of lists

**and, or** :: [Bool] -> Bool  
conjunction/disjunction

**sum** :: [Int] -> Int  
sum (overloaded)

**product** :: [Int] -> Int  
product (overloaded)

Exercise: Define these functions (use different names in order to avoid conflict with prelude).
Polygons

- We describe a polygon by the list of its vertices.
- We include polygons into the data type `Shape`:

```haskell
type Vertex = (Float, Float)

data Shape = Rectangle Side Side |
            Circle Radius |
            RTriangle Side Side |
            Polygon [Vertex] deriving Show
```
5.4 Recursive data: Lists

Polygon \([v1, v2, v3, v4, v5]\)
Areas of polygons

The area of a polygon can be computed as follows (Hudak, page 33):

- Every pair of adjacent vertices forms a trapezoid with respect to the $x$-axis.
- Starting at any vertex and working clockwise, compute these areas one-by-one, counting the area as positive if the $x$-coordinate increases, and negative if it decreases.
- The sum of these areas is then the area of the polygon.

**Coursework:** Define the function $\text{area} :: \text{Shape} \rightarrow \text{Area}$ for the data type $\text{Shape}$ including polygons.
Computing the area of Polygon \([v1, v2, v3, v4, v5]\)
Higher-order functions on lists

- Applying a function to all its elements
  \[
  \text{map} :: (a \to b) \to [a] \to [b]
  \]
  \[
  \text{map } f \ [ ] \ = \ [ ]
  \]
  \[
  \text{map } f \ (x:xs) \ = \ (f \ x) : (\text{map } f \ xs)
  \]

- Example: Squaring all elements of a list
  \[
  \text{squareAll} :: [\text{Int}] \to [\text{Int}]
  \]
  \[
  \text{squareAll } xs \ = \ \text{map } \text{square} \ x
  \]
  or, even shorter,
  \[
  \text{squareAll} \ = \ \text{map } \text{square}
  \]
• Example: Scaling shapes

scaleShape :: Float -> Shape -> Shape
scaleShape x (Rectangle s1 s2)
    = Rectangle (x * s1) (x * s2)
scaleShape x (Circle r)
    = Circle (x * r)
scaleShape x (RTriangle s1 s2)
    = RTriangle (x * s1) (x * s2)
scaleShape x (Polygon vts)
    = Polygon (map ((x1,x2) -> (x * x1,x * x2)) vts)
• ‘Folding’ lists

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]
\[
\text{foldr} \; f \; e \; [] = e
\]
\[
\text{foldr} \; f \; e \; (x:xs) = f \; x \; (\text{foldr} \; f \; e \; xs)
\]

• Example: Summing the elements of a list

\[
\text{sum} :: [\text{Int}] \to \text{Int}
\]
\[
\text{sum} \; xs = \text{foldr} \; (+) \; 0 \; xs
\]

\[
\text{sum} \; [3,12] = 3 + \text{sum} \; [12]
\]
\[
= 3 + 12 + \text{sum} \; []
\]
\[
= 3 + 12 + 0 = 15
\]
Taking parts of lists

- We had already take, drop. Now we consider similar higher-order functions.

- Longest prefix for which property holds

\[
\begin{align*}
takeWhile &:: (a \to \text{Bool}) \to [a] \to [a] \\
takeWhile\ p\ [] & = [] \\
takeWhile\ p\ (x:x:s) & = \begin{cases} \\
    x : \text{takeWhile}\ p\ xs & \text{if } p\ x \\
    [] & \text{otherwise}
\end{cases}
\end{align*}
\]
• Rest of this longest prefix

\[
dropWhile :: (a \to \text{Bool}) \to [a] \to [a]
dropWhile p \; \varepsilon \quad = \quad \varepsilon
\]
\[
dropWhile p \; x:xs
\quad \mid \; p \; x \quad = \quad dropWhile \; p \; xs
\quad \mid \; \text{otherwise} \quad = \quad x:xs
\]

• It holds: \(\text{takeWhile} \; p \; xs \; ++ \; \text{dropWhile} \; p \; xs \; == \; xs\)

• Combination of both

\[
\text{span} \quad :: \quad (a \to \text{Bool}) \to [a] \to ([a],[a])
\]
\[
\text{span} \; p \; xs \; = \; (\text{takeWhile} \; p \; xs, \; \text{dropWhile} \; p \; xs)
\]
• Order preserving insertion

\[
\text{ins} :: \text{Int} \to [\text{Int}] \to [\text{Int}]
\]
\[
\text{ins } x \text{ } xs = \text{lessx } ++ [x] ++ \text{grteqx } \text{ where}
\]
\[
(\text{lessx}, \text{grteqx}) = \text{span less xs}
\]
\[
\text{less } z = z < x
\]

• We use this to define insertion sort

\[
\text{isort} :: [\text{Int}] \to [\text{Int}]
\]
\[
\text{isort } xs = \text{foldr ins } [] \text{ } xs
\]
List comprehension

• A schema for functions on lists:
  ○ Select all elements of the given list
  ○ which pass a given test
  ○ and transform them into a result.

• Examples:
  ○ All letters in str to lower case
    
      [ toLower c | c <- str ]
    
  ○ Taking all letters from str
    
      [ c | c <- str, isAlpha c ]
    
  ○ Taking all letters from str and transforming them into lower case
    
      [ toLower c | c <- str, isAlpha c ]
• General form

\[ [E \mid c \leftarrow L, \text{test}_1, \ldots, \text{test}_n] \]

• With pattern matching:

\[
\text{addPair} :: [(\text{Int},\text{Int})] \rightarrow [\text{Int}]
\]

\[
\text{addPair}\ ls\ =\ [\ x + y \mid (x, y) \leftarrow ls ]
\]

• More than one generators possible

\[ [E \mid c_1 \leftarrow L_1, c_2 \leftarrow L_2, \ldots, \text{test}_1, \ldots, \text{test}_n] \]
• Example: Quicksort

  ○ Split a list into elements less-or-equal and greater than the first element,

  ○ sort the parts,

  ○ concatenate the results.
• Example: Quicksort

  ○ Split a list into elements less-or-equal and greater than the first element,

  ○ sort the parts,

  ○ concatenate the results.

```haskell
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) = qsort [ y | y <- xs, y <= x ]
               ++ [x] ++
               qsort [ y | y <- xs, y > x ]
```
‘Filtering’ elements

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{filter } p \ x s = [ x \mid x \leftarrow x s, p \ x ] \\
\]

or

\[
\text{filter } p \ [] = [] \\
\text{filter } p \ (x:xs) \\
\quad | \ p \ x = x:(\text{filter } p \ x s) \\
\quad | \ otherwise = \text{filter } p \ x s
\]
5.4 Recursive data: Lists

.
5.5 Examples

More examples on programming with lists and other recursive data types:

- Shopping baskets
- Modelling a library
- The sieve of Eratosthenes
- Sorting with ordering as input
- Forward composition
- Computing an Index
- Trees
### Shopping baskets

- An item consists of a name and a price. A basket is a list of items.

  ```haskell
  type Item = (String, Int)
  type Basket = [Item]
  ```

- The total price of the content of a basket.

  ```haskell
  total :: Basket -> Int
  total [] = 0
  total ((name, price):rest) = price + total rest
  or
  total basket = sum [price | (_,price) <- basket]
  ```
Modelling a library

type Person = String

type Book = String

type DBase = [(Person, Book)]

- Borrowing and returning books.

makeLoan :: DBase -> Person -> Book -> DBase
makeLoan dBase pers bk = (pers, bk) : dBase

returnLoan :: DBase -> Person -> Book -> DBase
returnLoan dBase pers bk
  = [ pair | pair <- dBase, pair /= (pers, bk) ]
• All books a person has borrowed.

books :: DBase -> Person -> [Book]
books db pers
    = [ book | (p,bk) <- db, p == pers ]

• Exercises

  o Returning all books.

  o Finding all persons that have borrowed a book.

  o Testing whether a person has borrowed any books.
Computing prime numbers

- **Sieve of Eratosthenes**
  - When a prime number $p$ is found, remove all multiples of it, that is, filter by $\left\{ n \rightarrow n \mod p \neq 0 \right\}$

  ```haskell
  sieve :: [Integer] -> [Integer]
  sieve [] = []
  sieve (p:xs) =
      p:(sieve (filter ($n \rightarrow n \mod p \neq 0) xs))
  ```

- **All primes in the interval $[1..n]$**

  ```haskell
  primes :: Integer -> [Integer]
  primes n = sieve [2..n]
  ```
Sorting with ordering as input

qsortBy :: (a -> a -> Bool) -> [a] -> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
    qsortBy ord [y| y <- xs, ord y x] ++ [x] ++
    qsortBy ord [y| y <- xs, not (ord y x)]

Sorting the library database by borrowers.

sortDB :: DBase -> DBase
sortDb = qsortBy \((p1,_) -> \((p2,_) -> p1 < p2)\)
Forward composition

• Recal the (predefined) composition operator:
  \[(.) \:: (b \to c) \to (a \to b) \to a \to c\]
  \[(f \ . \ g) x = f (g x)\]

• Sometimes it is better to swap the functions:
  \[(>\).>) \:: (a \to b) \to (b \to c) \to a \to c\]
  \[(f >\).> g) x = g (f x)\]

• Example: the length of a vector.
  \[|[x_1, \ldots, x_n]| = \sqrt{x_1^2 + \ldots + x_n^2} \cdot\]
  \[\text{len} :: [\text{Float}] \to \text{Float}\]
  \[\text{len} = (\text{map} (^2)) >\).> \text{sum} >\).> \text{sqrt}\]
Computing an index

**Problem:** Given a text

John Anderson my jo, John, \nwhen we were first aquent,  
\nyour locks were like the raven, \nyour bonnie brow as bent; \nbut now your brow is beld, John, \nyour locks are like the snow; \nbut blessings on your frosty pow, \nJohn Anderson, my jo. (Robert Burns 1789)

produce for every word the list of lines were it occurs:

`[((8],"1789"), ([1,8],"Anderson"), ([8],"Burns"), ([1,1,5,8],"John"),  
([8],"Robert"), ([2],"aquent"), ([5],"belt"), ([4],"bent"),  
([7],"blessings"), ([4],"bonnie"), ([4,5],"brow"), ([2],"first"),  
([7],"frosty"), ([3,6],"like"), ([3,6],"locks"), ([3],"raven"),  
([6],"snow"), ([2,3],"were"), ([2],"when"), ([3,4,5,6,7],"your")]`
• Specification

```haskell
  type Doc = String
  type Line = String
  type Wor = String     -- Word predefined

  makeIndex :: Doc -> [[[Int], Wor]]
```

• Dividing the problem into small steps Result type
5.5 Examples

- Dividing the problem into small steps
  
  (a) Split text into lines
• Dividing the problem into small steps

(a) Split text into lines

(b) Tag each line with its number

Result type

[Line]

[(Int, Line)]
5.5 Examples

- Dividing the problem into small steps

  (a) Split text into lines

  (b) Tag each line with its number

  (c) Split lines into words (keeping the line number)

Result type

- [(Int, Line)]

- [(Int, Wor)]
• Dividing the problem into small steps

(a) Split text into lines

(b) Tag each line with its number

(c) Split lines into words (keeping the line number)

(d) Sort by words:

Result type

[Line]

[(Int, Line)]

[(Int, Wor)]

[(Int,Wor)]
• Dividing the problem into small steps

(a) Split text into lines

(b) Tag each line with its number

(c) Split lines into words (keeping the line number)

(d) Sort by words:

(e) Collect same words with different numbers
5.5 Examples

- Dividing the problem into small steps

  (a) Split text into lines
  (b) Tag each line with its number
  (c) Split lines into words (keeping the line number)
  (d) Sort by words:
  (e) Collect same words with different numbers
  (f) Remove words with less than 4 letters

<table>
<thead>
<tr>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Line]</td>
</tr>
<tr>
<td>[(Int, Line)]</td>
</tr>
<tr>
<td>[(Int, Wor)]</td>
</tr>
<tr>
<td>[(Int,Wor)]</td>
</tr>
<tr>
<td>[([Int], Wor)]</td>
</tr>
<tr>
<td>[([Int], Wor)]</td>
</tr>
</tbody>
</table>
5.5 Examples

- Main program:

```
makeIndex :: Doc -> [(Int, Wor)]
makeIndex = -- Doc
  lines      >.> -- [Line]
numLines   >.> -- [(Int, Line)]
numWords   >.> -- [(Int, Wor)]
sortByWords >.> -- [(Int, Wor)]
amalgamate >.> -- [([Int], Wor)]
shorten    -- [([Int], Wor)]
```
• Implementing the parts:
  
  ○ Split into lines: `lines :: Doc -> [Line]` predefined
  
  ○ Tag each line with its number:

    ```
    numLines :: [Line] -> [(Int, Line)]
    numLines ls = zip [1.. length ls] ls
    ```

  ○ Split lines into words:

    ```
    numWords :: [(Int, Line)] -> [(Int, Wor)]
    Idea: Apply to each line: `words :: Line -> [Wor]` (predefined).
    Problem:
    
    ▶ recognises spaces only
    
    ▶ therefore need to replace all punctuations by spaces
  ```
numWords :: [(Int, Line)] -> [(Int, Wor)]
numWords = concat . map oneLine where
  oneLine :: (Int, Line) -> [(Int, Wor)]
  oneLine (num, line) = map (\w -> (num, w)) (splitWords line)

splitWords :: Line -> [Wor]
splitWords line =
  words [if c ‘elem’ puncts then ‘ ’ else c | c <- line ]
  where puncts = ";:.\"!()?{}-[]"
5.5 Examples

◦ Sort by words:

```
sortByWords :: [(Int, Wor)] -> [(Int, Wor)]
sortByWords = qsortBy ordWord where
  ordWord (n1, w1) (n2, w2) =
    w1 < w2 || (w1 == w2 && n1 <= n2)
```

◦ Collect same words with different numbers

```
amalgamate :: [(Int, Wor)]-> [[Int],Wor]]
amalgamate nws = case nws of
  [] -> []
  (_,w) : _ -> let ns = [ n | (n,v) <- nws, v == w ]
               other = filter (\(n,v) -> v /= w) nws
                in (ns,w) : amalgamate other
```
○ Remove words with less than 4 letters

\[
\text{shorten} :: [[\text{Int}], \text{Wor}]] \rightarrow [[\text{Int}], \text{Wor}]]
\]
\[
\text{shorten} = \text{filter} (\_ \_ \rightarrow \text{length} \ \text{wd} >= 4)
\]

Alternative definition:

\[
\text{shorten} = \text{filter} ((>= 4) . \text{length} . \text{snd})
\]
5.5 Examples

Binary Trees

- A binary tree is
  - either empty,
  - or a node with exactly two subtrees.
  - Each node carries an integer as label.

```haskell
data Tree = Null
          | Node Tree Int Tree
```

deriving (Eq, Read, Show)
Examples of trees:

- $\text{tree0} = \text{Null}$

- $\text{tree1} = \text{Node Null 3 Null}$

- $\text{tree2} = \text{Node (Node Null 5 Null) 2 (Node (Node Null 7 Null) 5 (Node Null 1 Null))}$

- $\text{tree3} = \text{Node tree2 0 tree2}$
5.5 Examples

- Creating balanced trees of depth \( n \) with label indicating depth of node:

\[
\text{balTree} :: \text{Int} \rightarrow \text{Tree} \\
\text{balTree} \ n = \begin{cases} \\
\text{Null} & \text{if } \ n == 0 \\
\text{let } t = \text{balTree} \ (n-1) & \text{else}
\end{cases}
\]
\[
\text{in } \text{Node} \ t \ n \ t
\]
• Testing whether an integer occurs in a tree:

\[
\text{member :: Tree } \to \text{ Int } \to \text{ Bool}
\]
\[
\text{member Null } _ = \text{ False}
\]
\[
\text{member } (\text{Node } l \ x \ r) \ y =
\]
\[
x == y \ || \ (\text{member } l \ y) \ || \ (\text{member } r \ y)
\]

• Traversing trees: Prefix order

\[
\text{preord :: Tree } \to \text{ [Int]}
\]
\[
\text{preord Null } = [\]
\]
\[
\text{preord } (\text{Node } l \ x \ r) = [x] ++ \text{preord } l ++ \text{preord } r
\]
• Infix and postfix order:

\[
inord :: \text{Tree} \rightarrow [\text{Int}]
\]

\[
inord \ \text{Null} = []
\]

\[
inord (\text{Node} \ l \ x \ r) = \text{inord} \ l \ ++ \ [x] \ ++ \ \text{inord} \ r
\]

\[
postord :: \text{Tree} \rightarrow [\text{Int}]
\]

\[
postord \ \text{Null} = []
\]

\[
postord (\text{Node} \ l \ x \ r) =

\quad \text{postord} \ l \ ++ \ [x] \ ++ \ \text{postord} \ r
\]

• The definitions of member, preorder, inorder, postorder are examples of

structural recursion on trees
Ordered trees

- A tree $\text{Node } l \ x \ r$ is ordered if
  - $\text{member } l \ y$ implies $y < x$ and
  - $\text{member } r \ y$ implies $x < y$ and
  - $l$ and $r$ are ordered.
• Example of an ordered tree:

\[
\begin{align*}
t &= \text{Node} (\text{Node} (\text{Node} \text{ Null} 1 \text{ Null}) \\
&\quad \quad 5 \\
&\quad \quad (\text{Node} \text{ Null} 7 \text{ Null})) \\
&\quad \quad 9 \\
&\quad (\text{Node} (\text{Node} \text{ Null} 13 \text{ Null}) \\
&\quad \quad 15 \\
&\quad \quad (\text{Node} \text{ Null} 29 \text{ Null}))
\end{align*}
\]
• For ordered trees, the test for membership is more efficient:

```haskell
member :: Tree -> Int -> Bool
member Null _ = False
member (Node l x r) y
  | y < x  = member l y
  | y == x = True
  | y > x  = member r y
```
5.5 Examples

- Order preserving insertion of a number in a tree:

```haskell
insert :: Tree -> Int -> Tree
insert Null y = Node Null y Null
insert (Node l x r) y
  | y < x  = Node (insert l y) x r
  | y == x = Node l x r
  | y > x  = Node l x (insert r y)
```

- More difficult: Order preserving deletion.
• **Order preserving deletion:**

```haskell
delete :: Int -> Tree -> Tree
delete y Null = Null
delete y (Node l x r)
   | y < x   = Node (delete y l) x r
   | y == x  = join l r
   | y > x   = Node l x (delete y r)
```

• **join** combines in an order preserving way two trees \( l, r \) provided all labels in \( l \) are smaller than those in \( r \).
join :: Tree -> Tree -> Tree
join t Null = t
join t s = Node t leftmost s1 where
  (leftmost, s1) = splitTree s
splitTree :: Tree -> (Int, Tree)
splitTree (Node Null x t) = (x, t)
splitTree (Node l x r) =
  (leftmost, Node l1 x r) where
  (leftmost, l1) = splitTree l
6 Type classes
Contents

• Overloading via type classes

• Declaring a type as an instance of a class

• Class constraints

• Some standard type classes

• User defined type classes
6.1 Overloading via type classes

• Different types with similar operations (methods) can be grouped together in a class
• Overloading: the same name for an operation can be used in different instances of the class
• Example: The class of types with equality and inequality (predefined in Haskell’s Prelude)

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

Every type `a` that is an instance of the class `Eq` is equipped with two binary boolean operations named `(==)` and `(=/=)`. 
Note: Although the intended meaning of these operations is equality respectively inequality, Haskell does not guarantee that these operations are indeed implemented accordingly!

- The types `Bool` and `Int` are both instances of the class `Eq`.

Therefore we can write expressions like `True == False` and also `5 == x`. 
class (Eq a) => Ord a where

    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a

-- Minimal complete definition: (<=) or compare
-- using compare can be more efficient for complex types

    compare x y | x==y = EQ
    | x<=y = LT
    | otherwise = GT

    x <= y = compare x y /= GT
6.1 Overloading via type classes

\[
\begin{align*}
x < y & \quad = \text{compare } x \ y \ == \ LT \\
x \geq y & \quad = \text{compare } x \ y \ /\!\!= \ LT \\
x > y & \quad = \text{compare } x \ y \ == \ GT \\
\text{max } x \ y & \quad \mid \ x \leq y \quad = \ y \\
& \quad \mid \ \text{otherwise} \quad = \ x \\
\text{min } x \ y & \quad \mid \ x \leq y \quad = \ x \\
& \quad \mid \ \text{otherwise} \quad = \ y 
\end{align*}
\]
We can declare the type `Tree` to be an instance of the class `Eq` by the following `instance declaration`:

```
instance Eq Tree where
    Null == Null       = True
    Null == Node _ _ _ = False
    Node _ _ _ == Null = False
    Node l1 n1 r1 == Node l1 n1 r1
        = l == l1 && n == n1 && r == r1
```

Note: `(=/=)` is automatically defined as

```
t1 /= t2 = not (t1 == t2)
```
• Tedious standard definitions as above can be avoided by deriving the instance automatically in the definition of the type

```
data Tree       = Null
  | Node Tree Int Tree
```

deriving (Eq)

• We could, however, also introduce another notion of equality on trees.

  For example: two trees are equal if they have the same nodes (multiple occurrences counted):
instance Eq Tree where
  t1 == t2
  = qsort (preord t1) == qsort (preord t2)

  t1 = Node (Node Null 3 Null)
       1
       (Node Null 5 Null)

  t2 = Node (Node (Node Null 1 Null) 3 Null)
       5
       Null

  t1 == t2  \implies True
6.3 Class constraints

- Example: Membership for polymorphic lists:

  ```haskell
  elem :: a -> [a] -> Bool
  elem x []     = False
  elem x (y:ys) = if x == y then True else elem x ys
  ```

  In the definition we test equality between objects of type `a` (`x == y`).

  But not all types do have an equality test!

  The type of `elem` is too general.

  We need to restrict the type variable `a` by a class constraint:
6.3 Class constraints

\[
\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

- The type variable \(a\) can be replaced by any instance of the class \(\text{Eq}\).

We may, for example, write:

\[
\text{elem } 3 \ [1,2,3]
\]

\[
\text{elem } 'A' \ ['B','C']
\]

- **Exercise**: What would be the signature of a polymorphic version of quicksort?
Also type classes may be polymorphic and have class constraints:

class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs, signum :: a -> a
    fromInteger :: Integer -> a
    fromInt :: Int -> a

    -- Minimal complete definition: All, except negate
    x - y       = x + negate y
    fromInt    = fromIntegral
negate x = 0 - x
6.4 Some standard type classes

- **Eq a** for \((==) :: a \rightarrow a \rightarrow \text{Bool}\) (Equality)
- **Ord a** for \((<=) :: a \rightarrow a \rightarrow \text{Bool}\) (Order)
- **Instances of Eq and Ord are:**
  - all basic types
  - Lists, tuples
  - **Not** for function types

- **Show a** for **show :: a \rightarrow \text{String}**
  - all basic types
  - lists, tuples
  - **Not** for function types
6.5 User defined type classes

• The user may also define their own type class.

• Example: The class of types with a distance operation

```
class Metric a where
    dist :: a -> a -> Float
```

• We may declare the type of Directions (see coursework 3) as an instance of Metric:
data Direction = North | East | South | West
deriving Eq

instance Metric Direction where
dist North South = 2
dist West East = 2
dist x y = if x == y then 0 else 1

- **Exercises:**
  - Declare other data types as instances of the class `Metric` in a reasonable way.
  - Define your own type class and instances thereof.
7 Modules
Contents

• Modules

• Exporting data types and functions

• Importing a module
7.1 Modules

- Restricting visibility by Encapsulation
- A module consists of:
  - Definitions of types, functions, classes
  - Declaration of the definitions that are visible from outside (interface).

- Syntax:

  \[
  \text{module } Name \ (\text{visible names}) \ \text{where} \ \text{body}
  \]

  - \text{visible names} may be empty
Example: Storage \((\text{Store})\)

- Type of storage parametrised by index type \(a\) and value type \(b\).
  \(\text{Store } a \ b\)

- Empty storage
  \(\text{initial} :: \text{Store } a \ b\)

- Reading a value:
  \(\text{value} :: \text{Eq } a \implies \text{Store } a \ b \rightarrow a \rightarrow \text{Maybe } b\)
    - Uses \(\text{data } \text{Maybe } b = \text{Nothing } \mid \text{Just } b\)

- Writing:
  \(\text{update} :: \text{Eq } a \implies \text{Store } a \ b \rightarrow a \rightarrow b \rightarrow \text{Store } a \ b\)
7.1 Modules

- Module declaration

```haskell
module Store(
    Store,
    initial, -- Store a b
    value,   -- Eq a => Store a b -> a -> Maybe b
    update,  -- Eq a => Store a b -> a -> b -> Store
) where

(Signature not necessary, but helpful)
```
Body of module:
Implementation of storage as a list of pairs

```
data Store a b = St [(a, b)]

initial :: Store a b
initial = St []

value :: Eq a => Store a b -> a -> Maybe b
value (St ls) a = case [b | (x, b) <- ls, a == x] of
    b : _ -> Just b
    [] -> Nothing
```

```
update :: Eq a => Store a b -> a -> b -> Store a b
update (St ls) a b = St ((a, b) : ls)
```
Alternative: Storage as a function

data Store a b = St (a -> Maybe b)

initial :: Store a b
initial = St (const Nothing)

value :: Eq a => Store a b -> a -> Maybe b
value (St f) a = f a

update (St f) a b
  = St (\x -> if x == a then Just b else f x)
Example: Shapes

module Shape (  
  Radius,        -- = Float  
  Side,          -- = Float  
  Area,          -- = Float  
  Vertex,        -- = (Float,Float)  
  Shape,         -- an algebraic data type  
  Rectangle,     -- :: Side -> Side -> Shape  
  Circle,        -- :: Radius -> Shape  
  RTriangle,     -- :: Side -> Side -> Shape  
  Polygon,       -- :: [Vertex] -> Shape  
  area,          -- :: Shape -> Area  
  perimeter      -- :: Shape -> Float  
) where
type Radius = Float
type Side = Float
type Area = Float
type Vertex = (Float, Float)

data Shape = Rectangle Side Side
  | Circle Radius
  | RTriangle Side Side
  | Polygon [Vertex]
deriving Show
area :: Shape -> Area
area (Rectangle s1 s2) = s1 * s2
area (Circle r) = pi * r^2
area (RTriangle s1 s2) = s1 * s2 / 2
area (Polygon vts) = abs (polArea vts)

polArea :: [Vertex] -> Area
polArea [] = 0
polArea [_] = 0
polArea vts@(v : _) = sum [ar trap | trap <- neighbors (vts ++ [v])]
  where
    ar ((x1,y1),(x2,y2)) = (x2 - x1) * (y2 + y1)/2
neighbors :: [a] -> [(a,a)]
neighbors xs = case xs of
  []            -> []
  (_,_)         -> []
  (x1 : x2 : vts) -> (x1,x2) : neighbors (x2 : xs)
perimeter :: Shape -> Float
perimeter shape = case shape of
  Rectangle s1 s2 -> 2 * (s1 + s2)
  Circle r -> r^2 * pi
  RTriangle s1 s2 -> s1 + s2 + sqrt (s1^2 + s2^2)
  Polygon vts -> polPeri vts

polPeri :: [Vertex] -> Float
polPeri [] = 0
polPeri [] = 0
polPeri vts@(v : _) =
  sum [len edge | edge <- neighbors (vts ++ [v])]
  where
    len ((x1,y1),(x2,y2)) = sqrt ((x1-x2)^2 + (y1-y2)^2)
7.2 Exporting data types and functions

• In a module

```
module Name (visible names) where body
```

only the data types and functions listed in *visible names* are exported, i.e., can be used outside the module.

○ In our first example the type `Store` and the functions `initial`, `value`, `update` are visible whereas the constructor `St` is invisible.

• It is often beneficial to hide implementation details (like the constructor `St`) which do not belong to the essential components of a data type.

○ This makes the module more robust against changes of the implementation (see chapter on abstract data types).
• The exported functions are listed without signature. It is recommended to add the signatures as comments.

• If the list of visible names is omitted (including brackets), then all data types and functions are exported:

\[
\text{module Name where body}
\]

• Declaration and body of a module \textit{Name} must reside in the file \textit{Name.hs}

• The name of a module must start with a \texttt{capital letter}.
7.3 Importing a module

- `import Name (identifiers)`
  - The `identifiers` declared in module `Name` are imported.
  - If `(identifiers)` is omitted, then everything is imported.

- Variant: `import qualified Name (identifiers)`
  - Identifiers are qualified by `Name`.

```haskell
import Store qualified
init = Store.initial
initial = ...
```
• **Variant**: `import Name [hiding] (identifiers)`

  - List of identifiers is **hidden**, that is, not imported.

    ```
    import Prelude hiding (foldr)
    foldr f e ls = ...
    ```

• With **qualified und hiding** naming conflicts can be resolved
8 Abstract data types
8 Abstract data types

Contents

• Abstract data types

• Sets as ADT

• Implementing finite sets with AVL trees
8.1 Abstract data types

• A module exports one or more data types together with a bunch of functions.

• Details of how data are constructed and functions are defined may be hidden.

• Such data types together with their associated functions are called abstract data types (ADTs).

• The advantage of an abstract data type is that its implementations may be changed without affecting the validity of other programs that make use of that data type.

We considered, for example, two different implementations the abstract data type Store $a \ b$. 
8.2 Sets as ADT

The following are common requirements on a data type of sets:

1. There is an empty set: \( \emptyset \)

2. We can insert an element in a set: \( \{x\} \cup s \)

3. We can test if an element is a member of a set: \( x \in s \)

4. We can compute the union of two sets: \( s \cup t \)

5. Filtering: Given a set \( s \) and a property \( p \) of elements we can compute a set containing exactly those members of \( s \) that have property \( p \): \( \{x \in s \mid p(x) = \text{True}\} \)
6. **Extensionality**: Two sets $s, t$ are equal when they contain the same elements, that is, an element is a member of $s$ if and only if it is a member of $t$:

$$s = t \iff \forall x (x \in s \iff x \in t)$$

We do not (yet) require a test for deciding equality of sets.
Implementation of sets

• As type of elements of sets we admit any type `a` that has an equality test (i.e. `a` must be an instance of the class `Eq`).

• We will begin with a discussion of two implementations of sets:

  (a) Sets as boolean functions.

  (b) Sets as lists of elements.
module Set (Set, -- type constructor
empty, -- Set a
insert, -- Eq a => a -> Set a -> Set a
member, -- Eq a => a -> Set a -> Bool
union, -- Eq a => Set a -> Set a -> Set a
sfilter, -- Eq a => (a -> Bool) -> Set a -> Set a
) where
Implementing sets as boolean functions

newtype Set a = MkSet (a -> Bool)

empty :: Set a
empty = MkSet (\x -> False)

insert :: Eq a => a -> Set a -> Set a
insert x (MkSet p) = MkSet (\y -> y == x || p x)

member :: Eq a => a -> Set a -> Bool
member x (MkSet p) = p x
union :: Eq a => Set a -> Set a -> Set a
union (MkSet p) (MkSet q) = MkSet (\x -> px || q x)

sfilter :: Eq a => (a -> Bool) -> Set a -> Set a
sfilter test (MkSet p) = MkSet (\x -> p x && test x)
Implementing sets as lists

newtype Set a = MkSet [a] deriving Show

empty :: Set a
empty = MkSet []

insert :: Eq a => a -> Set a -> Set a
insert x (MkSet xs) = MkSet (x : xs)

member :: Eq a => a -> Set a -> Bool
member x (MkSet xs) = elem x xs
union :: Eq a => Set a -> Set a -> Set a
union (MkSet xs) (MkSet ys) = MkSet (xs ++ ys)

sfilter :: Eq a => (a -> Bool) -> Set a -> Set a
sfilter test (MkSet xs) = MkSet (filter test xs)
Further functions on sets

The definitions of the following functions on lists only use what is exported by the module `Set`. Hence the definitions are independent of how sets are implemented.

```haskell
singleton :: Eq a => a -> Set a
singleton x = insert x empty

delete :: Eq a => a -> Set a -> Set a
delete x s = sfilter (/= x) s
```
meet :: Eq a => Set a -> Set a -> Set a
meet s t = sfilter ('member' s) t

minus :: Eq a => Set a -> Set a -> Set a
minus s t = sfilter (\x -> not (x 'member' t)) s

unions :: Eq a => [Set a] -> Set a
unions = foldr union empty

listToSet :: Eq a => [a] -> Set a
listToSet xs = foldr insert empty xs
Discussion

- Can we
  - define infinite sets, in particular a universal set that contains all elements of type \( a \)?
  - compute the complement of a set?
- Easy with the implementation based on boolean functions:
  ```haskell
  universal :: Set a
  universal = MkSet (\x -> True)
  
  complement :: Set a -> Set a
  complement (MkSet p) = MkSet (not . p)
  ```
- None of these are possible with sets implemented by lists.
Can we

- test whether a set is empty?
- compute the number of elements of a list?
- decide inclusion/equality between sets?

The implementation based on boolean functions inhibits the definition of any of these operations.

On the other hand there are no problems with the list implementation:
isEmpty :: Set a -> Bool
isEmpty (MkSet xs) = case xs of
  []       -> True
  _:_      -> False

card :: Eq a => Set a -> Int
card (MkSet xs) = length (removeDuplicates xs)

removeDuplicates :: Eq a => [a] -> [a]
removeDuplicates []    = []
removeDuplicates (x:xs)
  = (if x ‘elem‘ ys then [] else [x]) ++ ys
where \(ys = \text{removeDuplicates~xs}\)

\[
\text{subset} :: \text{Eq \(a\) => \text{Set \(a\)} \to \text{Set \(a\)} \to \text{Bool}}
\]
\[
\text{subset \((\text{MkSet \(xs\)}) \ (\text{MkSet \(ys\)})\)} = \text{sub} \(xs\) \(ys\) \text{ where}
\]
\[
\text{sub} \([]\) \(ys\) = \text{True}
\]
\[
\text{sub} \((x:xs)\) \(ys\) = x \ ‘\text{elem}‘ \ ys \&\& \text{sub} \(xs\) \(ys\)
\]

\[
\text{eqset} :: \text{Eq \(a\) => \text{Set \(a\)} \to \text{Set \(a\)} \to \text{Bool}}
\]
\[
\text{eqset} \(s\) \(t\) = s \ ‘\text{subset}‘ \(t\) \&\& t \ ‘\text{subset}‘ \(s\)
\]
• Can we list the members of a set?
  First attempt:

  \[
  \text{setToList} :: \text{Eq}\ a \Rightarrow \text{Set}\ a \rightarrow [a] \\
  \text{setToList}\ (\text{MkSet}\ xs) = \text{removeDuplicates}\ xs
  \]

  Does not work because equality of sets is not respected:

  \[
  \text{setToList}\ (\text{MkSet}\ [1,2]) \approx [1,2] \\
  \text{setToList}\ (\text{MkSet}\ [2,1]) \approx [2,1]
  \]

• Need order on elements:

  \[
  \text{setToList} :: \text{Ord}\ a \Rightarrow \text{Set}\ a \rightarrow [a] \\
  \text{setToList}\ (\text{MkSet}\ xs) = \text{qsort}\ (\text{removeDuplicates}\ xs)
  \]

  If we use only ordered and repetitionfree lists as representations of sets, then we can simply define:

  \[
  \text{setToList} :: \text{Ord}\ a \Rightarrow \text{Set}\ a \rightarrow [a] \\
  \text{setToList}\ (\text{MkSet}\ xs) = xs
  \]
• Conclusion (of the discussion)
  ○ Infinite sets and complements can be easily implemented with boolean functions.
  ○ Important operations on finite sets (cardinality, emptiness test) cannot be performed with boolean functions, but easily with lists.
  ○ In order to list the members of a finite set, we need an ordering on the elements.

• In the following we study finite sets over ordered elements.

• Lists are simple, but are unsuitable for representing large finite sets. We therefore use ordered trees (recall chapter 5.5).
Minimality versus Efficiency

• The interface of our first ADT of sets was minimal in the sense that we omitted a function if it could be defined by other functions (for example singleton, delete, meet, minus were defined in this way).

• The advantage of minimal declarations is that less work needs to be done when the implementation of the module is changed.
• But often functions can be implemented more efficiently by making direct use of the representation of sets: For example, our definition of `meet` has time complexity $O(n^2)$ whereas with sets represented by ordered lists one can achieve linear time.

• Therefore it is better to include important operations like `delete` and `meet` in the interface of an ADT even if they could be defined by other operations.
8.3 Implementing finite sets with AVL trees

Interface

module Set (
    Set, -- type constructor
    empty, -- Set a
    isEmpty, -- Ord a => Set a -> Bool
    insert, -- Ord a => a -> Set a -> Set a
    delete, -- Ord a => a -> Set a -> Set a
    member, -- Ord a => a -> Set a -> Bool
    setToList -- Ord a => Set a -> [a]
)
union, -- Ord a => Set a -> Set a -> Set a
meet, -- Ord a => Set a -> Set a -> Set a
minus, -- Ord a => Set a -> Set a -> Set a
sfilter, -- Ord a => (a -> Bool) -> Set a -> Set a
card, -- Ord a => Set a -> Int
subset, -- Ord a => Set a -> Set a -> Bool
eqset, -- Ord a => Set a -> Set a -> Bool
) where

We will implement only the operations up to setToList.
Finite sets — Implementation

- Efficient implementation of finite sets by *balanced ordered trees*
  - AVL-trees (Adelson, Velski, Landis) =

```haskell
type Set a = AVLTree a
```

- AVL-trees allow for $O(\log n)$ implementations of membership test, insertion and deletion.

- A tree is *balanced*, if
  - all subtrees are balanced, and
  - the heights of two subtrees differ by at most one.
• In a node we write the height of the subtree starting at that node.

```hs
data AVLTTree a = Null
                   | Node Int (AVLTTree a) a (AVLTTree a)
```

• Operations are similar to those on ordered trees (see chapter 5.5.), but balance must be preserved. In other words: Being balanced is an \textit{invariant}.

• Inserting and deleting might require \textit{rotating} subtrees.
8.3 Implementing finite sets with AVL trees

Simple Things First

- Empty set = empty tree

  \[\text{empty} :: \text{AVLTree}\ a\]
  \[\text{empty} = \text{Null}\]

- Emptiness test:

  \[\text{isEmpty} :: \text{AVLTree}\ a \rightarrow \text{Bool}\]
  \[\text{isEmpty}\ \text{Null} = \text{True}\]
  \[\text{isEmpty}\_ = \text{False}\]
Auxiliary functions

- The height of a tree
  - Select from node, don’t compute.

  \[
  \begin{align*}
  \text{ht} & : \text{AVLTree} \ a \rightarrow \text{Int} \\
  \text{ht} \ Null & = 0 \\
  \text{ht} \ (\text{Node} \ h \ _ \ _ \ _) & = h
  \end{align*}
  \]

- Creating a new node

  \[
  \begin{align*}
  \text{mkNode} & : \text{AVLTree} \ a \rightarrow \text{a} \rightarrow \text{AVLTree} \ a \rightarrow \text{AVLTree} \ a \\
  \text{mkNode} \ l \ n \ r & = \text{Node} \ h \ l \ n \ r \ \text{where} \\
  h & = 1 + \text{max} \ (\text{ht} \ l) \ (\text{ht} \ r)
  \end{align*}
  \]
Preserving balance

• Problem:
  After inserting or deleting an element, differences between heights of subtrees might become too big

• Solution:
  Rotate subtrees
8.3 Implementing finite sets with AVL trees

Rotating left

\[
\text{rotl} :: \text{AVLTree} \ a \to \text{AVLTree} \ a
\]

\[
\text{rotl} \ (\text{Node \ } _{\_} \ xt \ y \ (\text{Node \ } _{\_} \ yt \ x \ zt)) = \mkNode \ (\mkNode \ xt \ y \ yt) \ x \ zt
\]
rot \textit{r} :: AVLTree a \rightarrow AVLTree a 
rot \textit{r} (\text{Node } _{\text{ }} (\text{Node } _{\text{ }} \text{xt y yt}) \text{ x zt}) = 
mkNode \text{ xt y (mkNode yt x zt)}
Guaranteeing balance

- Case 1: Outer subtree too high
- Solution: Rotate left
 Guaranteeing balance

• Case 2: Inner subtree too high or of equal height

• Reduction to previous case by right-rotation of subtree
Guaranteeing balance

• Auxiliary function: Bias of a tree

   \text{bias} :: \text{AVLTree} \ a \rightarrow \text{Int}

   \text{bias} (\text{Node} \ _ \ \text{lt} \ _ \ \text{rt}) = \text{ht} \ \text{lt} - \text{ht} \ \text{rt}

• Two cases:
  
  o Case 1: \text{bias} < 0
  
  o Case 2: \text{bias} \geq 0

• Symmetric case (left subtree too high)
  
  o Case 1: \text{bias} > 0
  
  o Case 2: \text{bias} \leq 0
mkAVL \( xt \ y \ zt \): 

- **Assumptions:**
  - \( xt \) and \( zt \) differ in height by two at most;
  - all nodes in \( xt \) are smaller than those in \( zt \).

- **Construct new AVL-tree with node \( y \).**
8.3 Implementing finite sets with AVL trees

markdown

```
mkAVL :: AVLTree a -> a -> AVLTree a -> AVLTree a
mkAVL xt y zt
    | hx+1< hz &&
        0<= bias zt = rotl (mkNode xt y (rotr zt))
    | hx+1< hz     = rotl (mkNode xt y zt)
    | hz+1< hx &&
        bias xt<= 0 = rotr (mkNode (rotl xt) y zt)
    | hz+1< hx     = rotr (mkNode xt y zt)
    | otherwise    = mkNode xt y zt
where hx= ht xt; hz= ht zt
```
Order preserving insertion

- Insert node first, then rotate if necessary:

```haskell
insert :: Ord a => a -> AVLTree a -> AVLTree a
insert a Null = mkNode Null a Null
insert b (Node n l a r)
  | b < a = mkAVL (insert b l) a r
  | b == a = Node n l a r
  | b > a = mkAVL l a (insert b r)
```

- `mkAVL` guarantees balance.
Order preserving deletion

- Delete node first, then rotate if necessary:

```haskell
delete :: Ord a => a -> AVLTree a -> AVLTree a
delete x Null = Null
delete x (Node h l y r)
  | x < y = mkAVL (delete x l) y r
  | x == y = join l r
  | x > y = mkAVL l y (delete x r)
```

- `mkAVL` guarantees balance.
• **join** joins two trees preserving order (cf. chapter 5.5).

```
join :: AVLTree a -> AVLTree a -> AVLTree a
join xt Null = xt
join xt yt = mkAVL xt u nu where
  (u, nu) = splitTree yt
splitTree :: AVLTree a -> (a, AVLTree a)
splitTree (Node h Null a t) = (a, t)
splitTree (Node h lt a rt) =
  (u, mkAVL nu a rt) where
    (u, nu) = splitTree lt
```
Membership test

\[ \text{member} :: \text{Ord} \ a \Rightarrow \ a \rightarrow \ \text{AVLTree} \ a \rightarrow \ \text{Bool} \]
\[ \text{member} \ _ \ \text{Null} \ = \ \text{False} \]
\[ \text{member} \ x \ (\text{Node} \ _ \ \text{lt} \ a \ \text{rt}) \]
\[ \mid x < a = \text{member} \ x \ \text{lt} \]
\[ \mid x == a = \text{True} \]
\[ \mid x > a = \text{member} \ x \ \text{rt} \]
Listing a set

foldT :: (a-> b-> b)-> b-> AVLTree a-> b
foldT f e Null = e
foldT f e (Node _ l a r) =
    f a (foldT f e l) (foldT f e r)

- Listing a set: In order traversal (via fold)

setToList :: Ord a => AVLTree a -> [a]
setToList = foldT (\x t1 t2 -> t1 ++ [x] ++ t2) []