



Functional Programming

Week 1 – Organisation and Introduction

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Organization

Lecture (VO 2)

- LV-Number: 703024
- lecturer: René Thiemann
consultation hours: Tuesday 10:00–11:00 in 3M09 (ICT building)
or at <https://webconference.uibk.ac.at/b/ren-mxw-peh-o9v>
- time and place: Monday, 12:15 – 14:00 in HS B or online
- course website: <http://cl-informatik.uibk.ac.at/teaching/ws21/fp/>
- lecture will be in German with English slides
- slides are available online and contain links
- online registration required until February 5, 2022
- lecture will be recorded and streamed;
videos and live-streams accessible in OLAT-VO;
live-questions will be taken via ARSnova, session 25123751



ARSnova

- login via browser

<https://arsnova.uibk.ac.at/mobile/#id/25123751>

- main feature: ask questions

The image displays three screenshots of the ARSnova mobile application:

- Session Screen:** Shows a green header "Session". Below it is a session ID "25123751" and a button "Session betreten". A sidebar on the left lists "Besuchte Sessions" with items like "Demo Session (86156280)" and "Funktionaler Programmierung VO (25123751)".
- FP VO Screen:** Shows a green header "FP VO". Below it is a session ID "Session: 25 12 37 51" and a message "Ich habe eine Frage" with a question mark icon.
- Feedback Screen:** Shows a green header "Feedback". It includes a note "Ihr Feedback bleibt anonym". There are buttons for "Thema" (Topic) and "Folie 2". The "Text" field contains the question "Wie finde ich den OLAT Kurs?". At the bottom, there is a note: "Hinweis: Die Lehrperson kann Ihre Fragen und Kommentare auf der Twitter Wall anzeigen." and two buttons: "Vorschau" (Preview) and "Abschicken" (Send).

Schedule

lecture 1	October	5	lecture 8	November	29
lecture 2	October	11	lecture 9	December	6
lecture 3	October	18	lecture 10	December	13
lecture 4	October	25	lecture 11	January	10
lecture 5	November	8	lecture 12	January	17
lecture 6	November	15	lecture 13	January	24
lecture 7	November	22	Q & A	January	31

- no lecture on November 1 (all saints day)
- on January 31 no new content is presented

Proseminar (PS 1)

- LV-Number: 703025
- new exercise sheets available online on Tuesday or Wednesday
- solved exercises must be entered in OLAT-PS
 - mark which exercises have been solved (Kreuzliste)
 - upload solution: program source (everyone), everything (groups 10–12)
 - deadline: 6 am before PS on Wednesday
- solutions will be presented in proseminar groups
- first exercise sheet: today
- proseminar starts on October 6 or October 13
- proseminar on October 6
 - voluntary, discussion of basic topics (command line, ...)
 - might be in different room (not in HSB 5), check LFU online
- **attendance is obligatory starting from October 13**
- registration deadline was in September
- exercise sheets will be English, seminar groups in German

Proseminar Groups

- in total 12 groups, cf. [LFU online](#)
- two kinds of proseminars
 - physical meetings: Bachelorstudium, Lehramtsstudium
 - virtual meetings: Erweiterungsstudium
- all groups are completely full
- change of groups only possible if you find exchange partner
 - go to [OLAT-PS](#) and use “Tauschbörse”
 - several swaps have already been conducted
 - exchanges are only possible until Friday this week

Tutorium

- opportunity to **ask questions** about topics of lecture and exercises
- presentation of **more examples**
- no new topics, no influence on grades, no solutions to exercises
- attendance voluntary
- tutor: Benedikt Dornauer
- Monday 18:15–19:00
 - starts next week
 - HS B
 - stream + recordings in **OLAT-VO**
 - online questions via **ARSnova** (session 52444256)

Weekly Schedule

- Monday 12:15 – 14:00: lecture on topic n
- Monday 18:15 – 19:00: tutorium on topic $n - 1$ or n
- Tuesday or Wednesday: exercise sheet n on topic n available
- Wednesday 6 am: deadline for upload of solution of ex. sheet $n - 1$
- Wednesday afternoon: proseminalars on exercise sheet $n - 1$
- ...

Grading

- separate grades for lecture and proseminar
- lecture
 - grading solely via exam
 - 1st exam on February 7, 2022
 - online registration required from January 1 – 31 via [LFU online](#)
(deregistration still possible in February)
 - 2nd and 3rd exam in March and September (tentative)
 - **it suffices to pass one of the three exams**
- proseminar
 - 80 %: scores from weekly exercises
 - 20 %: presentation of solutions

Literature



slides

- no other topics will appear in exam ...
- ... but topics need to be understood thoroughly
 - read and write functional programs
 - apply presented techniques on new examples
 - not only knowledge reproduction



Richard Bird. Introduction to Functional Programming using Haskell, 2nd Edition, Prentice Hall.

Introduction

(Functional) Programming

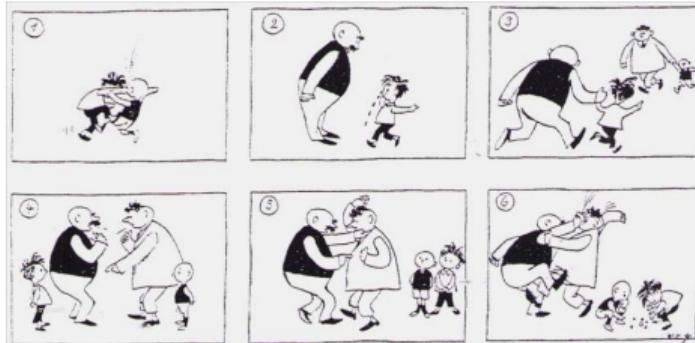
- task: solve problems
 - sort a list
 - generate a website
 - navigate from Innsbruck to Cologne
- distinguish between **data** ...
 - input [1, 5, 2] and output [1, 2, 5]
 - query “search for ‘functional programming’” and resulting website
 - map of Europe, two locations and route
- ... and **programs**
 - control over how data should be processed
 - mostly written by humans
- usually computers are used for executing a program on some input, but computation can also be done on paper or in **mind**

How to Learn Programming

- + read, study and write programs (many)
- + actively attend lecture and proseminar
- + try to solve exercises (alone or discuss in small teams)
- copy solutions from other students or from the internet

Algorithms and Programs

story (language agnostic)



algorithm (prog. language agn.)

- task: determine the maximum of m and a list of numbers
- if list is empty, result is m
- otherwise, change m to max. of head of list and m
- continue with tail of list

text (language dependent)

- Tom and Paul were struggling until
...
• Thomas und Paul rauften solange bis
...
• 토마스와 파울은 싸우고 있었는데...

program (language dependent)

```
maxlist m [] = m
• maxlist m (x : xs) =
    maxlist (max m x) xs
while (list != null) {
    m = max(m, list.head);
    list = list.next;
}
return m;
```

Different Programming Styles

- Imperative Programming (VO Introduction to Programming)
 - state is mapping of variables to data
 - assignments instruct computer to update state
 - example
 - consider assignment `x := x + 5;`
 - if in a state `x` stores value 7
 - then after executing assignment `x` stores value 12
- Functional Programming (this lecture)
 - define functions (mathematical: same input implies same output)
 - new results (of function invocations) are computed,
but there is no notion of state that can be updated
 - example
 - consider function definition `add5 x = x + 5` where `x` is parameter;
 - function invocation `add5 7`
 - is evaluated, e.g., `add5 7 = 7 + 5 = 12`
 - 7 is not changed into 12, there is no state with variable `x`
- Logic Programming, Object Oriented Programming, ...

Different Programming Styles

- fact: most programming languages are of equal power
- demand for different styles still reasonable
 - each style has its own **distinguishing features** and limitations
(like in real languages: translate “Ohrwurm” or “Internetbrowser”)
 - good programmer should know about alternatives:
choose suitable style and language depending on problem and context
- advantages of functional programming
 - **intuitive** evaluation mechanism
 - suitable for **verification**
 - **expressive** language features
 - suitable for **parallelization**
- disadvantages of functional programming
 - more difficult to model **state**, **side-effects**, and **I/O**
 - not main-stream in industry, but getting more popular

Different Functional Programming Languages

- combinatory logic (Moses Schönfinkel 1924, Haskell Curry 1930): foundation of FP
- λ -calculus (Alonzo Church 1936): foundation of FP
- LISP (John McCarthy, 1958): List Processing
- ML (Robin Milner, 1973): Meta Language, several dialects
- Erlang (Ericsson, 1987): distributed computing
- **Haskell** (Paul Hudak and Philip Wadler, 1990): language in this course
- F# (Microsoft, 2002) and Scala (Martin Odersky, 2003): combine different programming styles, including FP

Syntax and Semantics

- **syntax** of a (programming) language defines valid sentences (programs)
 - “This is a proper English sentence.”
 - “this one not propper”
 - computers refuse programs that contain syntactical errors!
- **semantics** defines the meaning of valid sentences / programs
 - “Clean your room!”
 - `let xs = 1 : 1 : zipWith (+) xs (tail xs) in take 9 xs`
- we will learn both syntax and semantics of Haskell



Haskell Scripts

```
-- This script is stored in file example.hs

add5 x = x + 5

{- the following function takes a temperature in
degree Fahrenheit and converts it into Celsius -}

fahrenheitToCelsius f = (f - 32) * 5 / 9
```

- a Haskell script (= program) has file extension .hs
- a script is a collection of (several) function definitions
- comments are just for humans, ignored by computer
- single-line and multi-line comments
 - single: -- everything right of -- is a comment
 - {- multi-line comments can deactivate
 - multi: areaRectangle width height = width * height
 - parts of script easily -}

Writing Haskell Scripts

```
-- This script is stored in file example.hs
```

```
add5 x = x + 5
```

```
fahrenheitToCelsius f = (f - 32) * 5 / 9
```

- coloring

- when entering a Haskell script, one does **not** add colors in a text editor
- **syntax highlighting**: often editors for computer programs automatically add colors to simplify reading; quickly distinguish
 - comments, keywords, names of functions, names of parameters, ...

- function- and parameter-names (`add5`, `x`, ...)

- always start with a lowercase letter, may contain digits
- convention: long names use camelCase (`fahrenheitToCelsius`, ...)

- white-space (spaces, tabs, newlines, ...)

- in Haskell white-space matters
- for the moment, start every new line without blanks
- the following script is not accepted

```
add5 x = x + 5
```

```
fahrenheitToCelsius f = (f - 32) * 5 / 9
```

Functional Programming – Sessions

- starting a session is like activating your calculator
- we use `ghci`, an interpreter for Haskell

```
rene$ ghci                      -- start the interpreter
Prelude> 42                      -- enter a value
42
Prelude> 5 * (3 + 4)              -- evaluate an expression
35
Prelude> :load example.hs        -- load script from file
[1 of 1] Compiling Main ( example.hs, interpreted )
Ok, 1 module loaded.             -- script was accepted
*Main> fahrenheitToCelsius 95   -- invoke our function
35.0
*Main> :quit
```

Workflow for Functional Programming

- define functions in script
- load script (will compile script or deliver error message)
 - parse error: `5 +` (argument missing)
 - type error: `5 + "five"` (cannot add number and text)
 - error-messages are sometimes cryptic
- enter expression and start evaluation to get result (**read-eval-print loop, REPL**)
 - **result**: (canonical representation of some) value which cannot be further simplified,
e.g., `42`, `"hello"`, `[7,1,3]`, ...
but not `5 + 7`, `fahrenheitToCelsius 8`, ...
 - evaluation uses
 - **built-in** functions (`+`, `*`, `:`, `++`, `head`, `tail`, ...), defined in **Prelude**
 - **user-defined** functions (`fahrenheitToCelsius`, ...) from script-files

Compare FP to Calculator

- enter expression and let it compute result
- restricted to numbers and built-in functions

Comparison: FP vs Calculator

- task: convert many temperatures from Fahrenheit to Celsius: 8, 9, 300, ...
- calculator: enter the following expressions
 - $(8 - 32) * 5/9$
 - $(9 - 32) * 5/9$
 - $(300 - 32) * 5/9$
 - ...

(quite tedious: enter same formula over and over again)
- FP
 - write one program: `fahrenheitToCelsius f = (f - 32) * 5 / 9`
 - just evaluate the function on the various inputs
 - `fahrenheitToCelsius 8`
 - `fahrenheitToCelsius 9`
 - `fahrenheitToCelsius 300`
 - ...

(concise, readable, easy: just invoke function)
 - or just: `map fahrenheitToCelsius [8,9,300,...]`
- program(s): a recipe to turn inputs into desired outputs

Summary

- Haskell scripts are stored in .hs-files
- functional programming: specify functions (input-output-behaviour)
- ghci loads scripts and evaluates expressions
- next lecture: beyond numbers — structured data



Functional Programming

Week 2 – Tree Shaped Data and Datatypes

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- algorithm (can be informal) vs. program (concrete prog.-language)
- Haskell script (code, program, ...), e.g., `program.hs`
`fahrenheitToCelsius f = (f - 32) * 5 / 9`
consists of function definitions that describe input-output behaviour
- function- and parameter-names have to start with lowercase letters
- read-eval-print loop:
load script, enter expressions and let these be evaluated

```
$ ghci program.hs
... welcome message ...
Main> fahrenheitToCelsius (3 + 20) - 7
-12.0
Main> ... further expressions ...
...
Main> :q
```

Structured Data

Different Representations of Data

- some (abstract) element can be represented in various ways
- example: numbers

• roman:	XI
• decimal:	11
• binary:	1011
• English:	eleven
• tally list:	

- fact: algorithms depend on concrete representation
- example: addition

- decimal + binary: process digits of both numbers from right to left

$$\begin{array}{r} 7823 \\ + 909 \\ \hline 8732 \end{array}$$

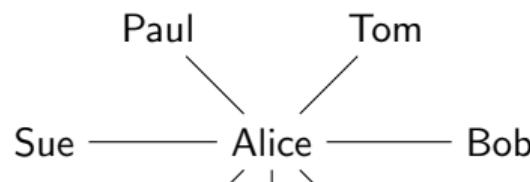
- tally list: just write the two numbers side-by-side $(||| + || = ||||)$
- roman: algorithm? $(IV + IX = XIII)$
- English: not well-suited $(\text{twentynine} + \text{two} = \text{thirtyone})$
- in Haskell: numbers are built-in, representation not revealed to user

Different Representations of Data – Continued

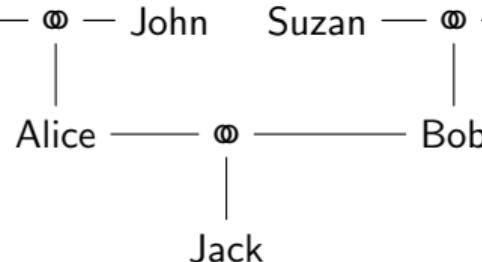
- representation must be chosen appropriately
- example: person
 - photographer:



- social analysis:



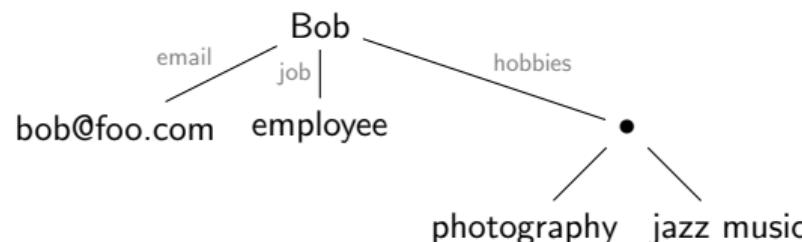
- advertising: Bob (bob@foo.com, employee, hobbies: photography, jazz music, ...)
- genealogist: Carmen — ♂ — John Suzan — ♂ — Jack



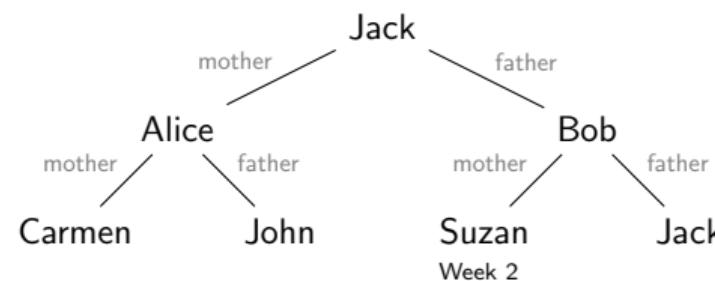
Tree Shaped Data

- in functional programming most of the data is **tree shaped**
- a **tree**
 - has exactly one **root** node
 - can have several subtrees; nodes without subtrees are **leaves**
 - nodes and edges can be labeled
- in computer science, trees are usually displayed upside-down
- examples from previous slide

- advertizing:



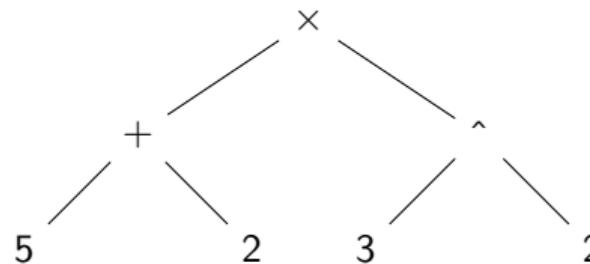
- genealogist:



Expressions = Trees

- mathematical expressions can be represented as trees
- example

- expression in textual form: $(5 + 2) \times 3^2$
- expression as tree



- remarks
 - the process of converting text into tree form is called **parsing**
 - operator precedences ($^$ binds stronger than \times b.s.t. $+$) and parentheses are only required for parsing
 - parsing $(5 + 2) \times (3^2)$ results in tree above
 - $5 + 2 \times 3^2$ and $((5 + 2) \times 3)^2$ represent other trees
 - algorithm of calculator
 - convert textual input into tree
 - evaluate the tree bottom-up, i.e., start at leaves and end at root

Programs = Trees

- programs can be represented as trees, too: **abstract syntax tree**

- example

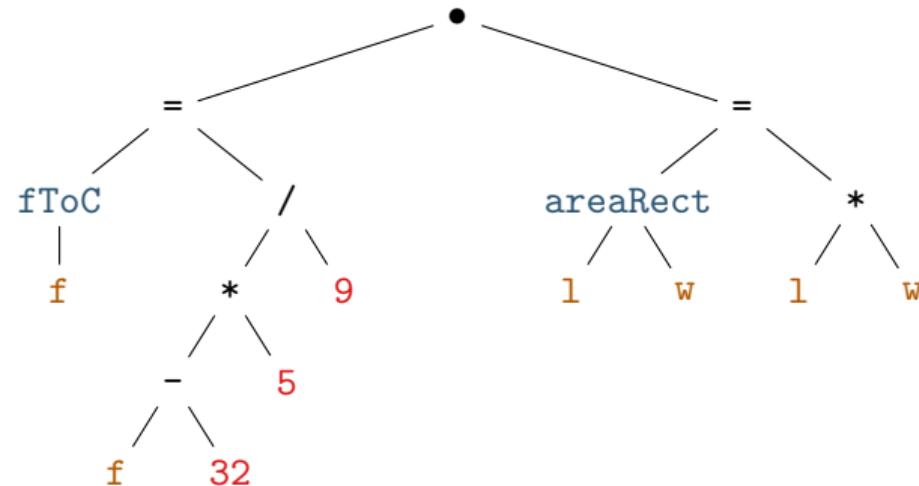
- program in textual form

```
-- some comment
```

```
fToC f = (f - 32) * 5 / 9
```

```
areaRect l w = l * w
```

- abstract syntax tree (draft)



- comments and parentheses are no longer present in syntax tree

Tree Shaped Data

- many programs deal with tree shaped data
- examples
 - calculator evaluates expression tree
 - compiler translates abstract syntax tree into machine code
 - search engine translates query into HTML (tree shaped)
 - contact application manages tree shaped personal data
 - file systems are organised as trees
- trees as **mental model** or representation of data is often suitable
- good news: processing tree shaped data is well-supported in functional programming
- next lecture: define functions on trees
- this lecture: restriction of trees via **types**

Types

Types

- functions are often annotated by their domain and codomain, e.g.,
 - $(!): \mathbb{N} \rightarrow \mathbb{N}$
 - $(/): \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \rightarrow \mathbb{R}$
 - $\log_2: \mathbb{R}_{>0} \rightarrow \mathbb{R}$
- domain and codomain provide useful information
 - domain: what are allowed inputs to a function
 - codomain: what are potential outputs of the function
- aim: specify domains and codomains of (Haskell-)functions
- notions
 - elements or values
 - maths: 5, 8, π , $-\frac{3}{4}$, ...
 - Haskell: 5, 8, 3.141592653589793, -0.75, ..., "hello", 'c', ...
 - sets of elements to specify domain or codomain, in Haskell: types
 - maths: \mathbb{N} , \mathbb{Z} , \mathbb{Q} , \mathbb{R} , $\mathbb{Q} \setminus \{0\}$, ...
 - Haskell: Integer, Double, String, Char, ...

Typing Judgements

- in maths, we write statements like $7 \in \mathbb{Z}$, $7 \in \mathbb{R}$, $0.75 \notin \mathbb{Z}$
- in Haskell we can also express that a value or expression has a certain type via **typing judgements**
 - format: `expression :: type`
 - examples
 - `7 :: Integer` or `7 :: Double`
 - `'c' :: Char`
- that an expression indeed has the specified type is checked by the Haskell compiler
 - if an expression has not the given type, a type error is displayed
 - examples
 - `7 :: String` or `0.75 :: Integer` or `'c' :: String`
 - `(7 :: Integer) :: Double`
 - remarks
 - unlike in maths where $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q}$, in Haskell the types `Integer` and `Double` are **not** subtypes of each other
 - although some expressions can have both types (e.g., `7 + 5`), in general numbers of different types have to be converted explicitly
 - once a typing judgement is applied, the type of that expression is fixed

Typing of Haskell Expressions

- not only values but also functions have a type, e.g.,

- `(/)` :: `Double` → `Double` → `Double`
- `(+)` :: `Integer` → `Integer` → `Integer`
- `(+)` :: `Double` → `Double` → `Double`
- `head` :: `String` → `Char`

remarks

- a function can have multiple types, e.g., `(+)`
- limited expressivity, e.g. `(/)` :: `Double` → `Double \ {0}` → `Double` not allowed
- type checking enforces that in all function applications,
type of arguments matches input-types of function
- example: consider expressions `expr1 / expr2`
 - recall: `(/)` :: `Double` → `Double` → `Double`
 - it will be checked that both `expr1` and `expr2` have type `Double`
 - type of the overall expression `expr1 / expr2` will then be `Double`
- examples
 - `5 + 3 / 2`
 - `5 + '3'` or `5.2 + 0.8` :: `Integer`



Static Typing

- Haskell performs static typing
- **static** typing: types will be checked before evaluation
(by contrast, dynamic typing checks types during evaluation)
- when loading Haskell script
 - check types of all function definitions `someFun x ... z = expr`:
check that lhs `someFun x ... z` has same type as rhs `expr`
 - consequence: expressions cannot change their type during evaluation
- when entering expression in REPL: type check expression before evaluation
- benefits
 - no type checking required during evaluation
 - no type errors during evaluation

Built-In Types – A First Overview

- numbers
 - **Integer** – arbitrary-precision integers
 - **Int** – fixed-precision integers with at least 29 bits (-100, 0, 999)
 - **Float** – single-precision floating-point numbers (-12.34, 5.78e36)
 - **Double** – double-precision floating-point numbers
- characters and text
 - **Char** – a single character ('a', 'Z', ' ')
 - **String** – text of arbitrary length ("", "a", "The answer is 42.")
 - some characters have to be **escaped** via the backslash-symbol \:
 - '\t' and '\n' – tabulator and new-line
 - '\"' and '\' – double- and single quote
 - '\\' – the backslash character
 - example: in the program

```
text = "Please say \"hello\"\nwhenever you enter the room"  
the string text corresponds to the following two lines:  
  
Please say "hello"  
whenever you enter the room
```
- **Bool** – yes/no-decisions or truth-values (**True**, **False**)

Datatypes

Current State

- each value and function in Haskell has a type
- types are used to define input and output of function
- example: `fahrenheitToCelsius :: Double -> Double`
- built-in types for numbers, strings, and truth values
- missing: how to define types that describe tree shaped data?
- solution: definition of (algebraic) **datatypes**

Datatype Definitions

- recall: a tree consists of a (labelled) root and 0 or more subtrees
- a **datatype** definition defines a set of trees by specifying all possible labelled roots together with a list of allowed subtrees
- Haskell scripts can contain many **datatype definitions** of the form

```
data TName =  
    CName1 type1_1 ... type1_N1  
  | ...  
  | CNameM typeM_1 ... typeM_NM  
deriving Show
```

where

- **data** is a Haskell keyword to define a new datatype
- **TName** is the name of the new type; **type-names** always start with capital letters
- **CName1, ..., CNameM** are the labels of the permitted roots;
these are called **constructors** and have to start with capital letters
- **typeI_J** can be any Haskell type, including **TName** itself
- **|** is used as separator between different constructors
- **deriving Show** is required for displaying values of type **TName**

Example Datatype Definition – Date

```
data Date = -- name of type
  DMY      -- name of constructor
    Int     -- day
    Int     -- month
    Integer -- year
```

deriving Show

- here, there is only one constructor: `DMY`
- for day and month the precision of `Int` is sufficient
- the values of the type `Date` are exactly trees of the form



- in Haskell, these trees are built via the constructor `DMY`; `DMY` is a function of type `Int -> Int -> Integer -> Date` that is **not evaluated**
- example value of type `Date`: `DMY 11 10 2021`

Example Datatype Definition – Person

```
data Person = -- name of type
    Person      -- constructor name can be same as type name
    String      -- first name
    String      -- last name
    Bool        -- married
    Date        -- birthday
```

deriving Show

- reuse of previously defined types is permitted, in particular Date
- this leads to trees with more than one level of subtrees
- example program that defines a person (and an auxiliary date)

```
today = DMY 11 10 2021
```

```
myself = Person "Rene" "Thiemann" True today
```

-- is the same as

```
myself = Person "Rene" "Thiemann" True (DMY 11 10 2021)
```

Example Datatype Definition – Vehicle

```
data Brand = Audi | BMW | Fiat | Opel deriving Show
data Vehicle =
    Car Brand Double -- horsepower
  | Bicycle
  | Truck Int -- number of wheels
deriving Show
```

- `Brand` just defines 4 car brands; all "trees" of type `Brand` consist of a single node; such datatypes are called **enumerations**
- there are three kinds of `Vehicles`, each having a different list of types
- example expressions of type `Vehicle`:

Car Fiat (60 + 1)

Car Audi 149.5

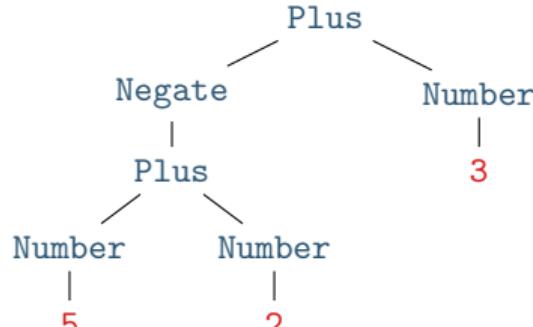
Bicycle

Truck (-7) -- types don't enforce all sanity checks

Example Datatype Definition – Expr

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr  
deriving Show
```

- type Expr models arithmetic expressions with addition and negation
- Expr is a **recursive** datatype: Expr is defined via Expr itself
- recursive datatypes contain values (trees) of arbitrary large height
 - expression $-(5 + 2)) + 3$ in Haskell (as value of type Expr):
`Plus (Negate (Plus (Number 5) (Number 2))) (Number 3)`
 - expression as tree



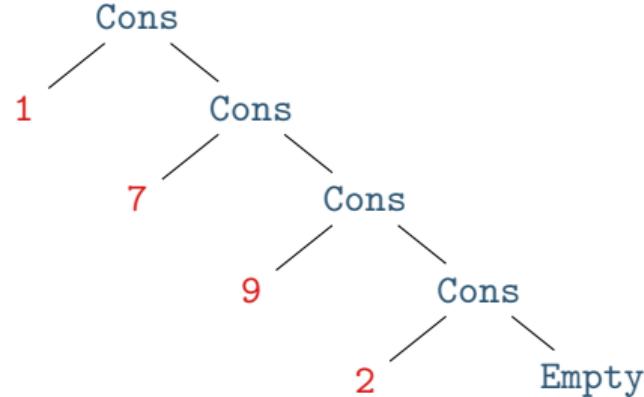
Example Datatype Definition – Lists

- lists are just a special kind of trees, e.g., lists of integers

```
data List =  
    Empty  
  | Cons Integer List  
deriving Show
```

- example representation of list [1, 7, 9, 2]

- in Haskell: Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))
- as tree:



Summary

- mental model: data = tree shaped data
- type = set of values; restricts shape of trees
- built-in types for numbers and strings
- user-definable datatypes, e.g., for expressions, lists, persons

```
data TName =  
    CName1 type1_1 ... type1_N1  
    | ...  
    | CNameM typeM_1 ... typeM_NM  
deriving Show
```

- missing: function definitions on trees



Functional Programming

Week 3 – Functions on Trees

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- data = tree shaped data
- every value, expression, function has a **type**
- type of `lhs` and `rhs` has to be equal in function definition `lhs = rhs`
- **built-in types**: Int, Integer, Float, Double, String, Char, Bool
- user defined **datatypes**

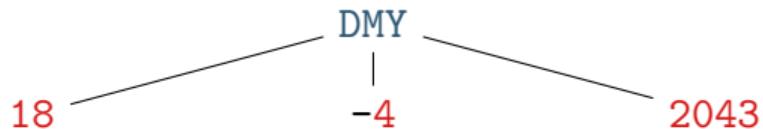
```
data TName =
    CName1 type1_1 ... type1_N1
  | ...
  | CNameM typeM_1 ... typeM_NM
deriving Show
```

- **constructor** `CNameI :: typeI_1 -> ... -> typeI_NI -> TName`
is a function that is not evaluated
- `TName` is **recursive** if some `typeI_J` is `TName`
- names of types and constructors start with uppercase letters

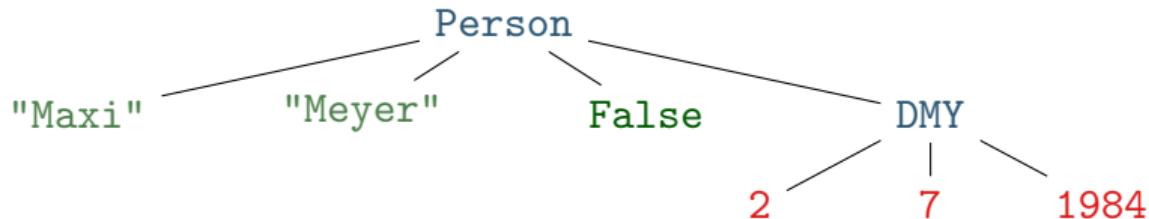
Examples of Nonrecursive Datatype Definitions

```
data Date = DMY Int Int Integer deriving Show  
data Person = Person String String Bool Date deriving Show
```

- values of type Date are trees such as



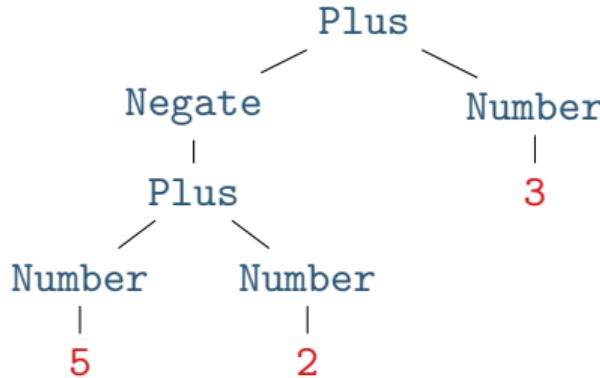
- values of type Person are trees such as



Example of Recursive Datatype Definition – Expr

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr  
deriving Show
```

- expression $-(5 + 2)) + 3$ in Haskell (as value of type Expr):
`Plus (Negate (Plus (Number 5) (Number 2))) (Number 3)`
- expression as tree



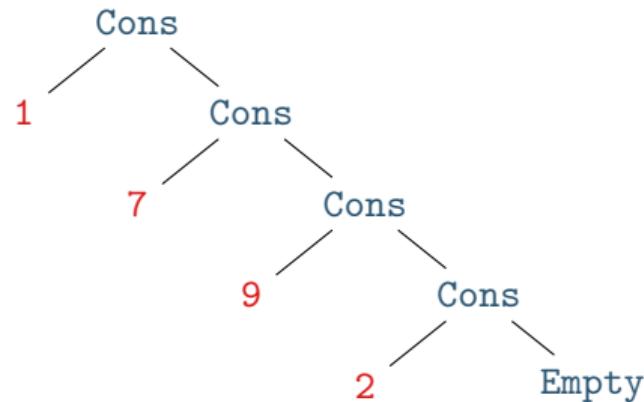
Example of Recursive Datatype Definition – Lists

- lists are just a special kind of trees, e.g., lists of Integers

```
data List =  
    Empty  
  | Cons Integer List  
deriving Show
```

- example representation of list [1, 7, 9, 2]

- in Haskell: Cons 1 (Cons 7 (Cons 9 (Cons 2 Empty)))
- as tree:



Function Definitions Revisited

Function Definitions and Expressions

- so far all functions definitions have been of the shape

`funName x1 ... xN = expr`

where

- `x1 ... xN` are parameter names;
a function can have arbitrary many parameters (including zero)
- `expr` is an **expression**, i.e., a mathematical expression consisting of
 - literals: `5, 3.4, 'a', "hello", ...`
 - function applications: `pi, square expr, average expr1 expr2, ...`
 - constructor applications: `True, Number expr, Cons expr1 expr2, ...`
 - operator applications: `- expr, expr1 + expr2, ...`
 - parenthesis
- remark: function and constructor application binds stronger than operator applications
 $(\text{square } 2) + 4 = \text{square } 2 + 4 \neq \text{square } (2 + 4)$

- this lecture: **extend shape of function definitions**,
in particular to define functions on tree shaped data

Creating New Values – Expr Example

- creation of new values is easily possible using constructors
- example: consider `Expr` datatype

```
data Expr = Number Int | Plus Expr Expr | Negate Expr
```

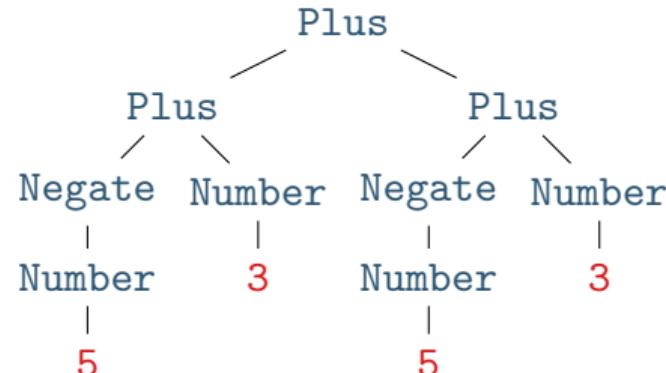
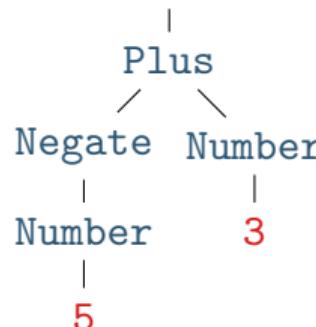
(in the remainder of the lecture "`deriving Show`" is omitted)

- task: define a function for doubling, i.e., multiplication by 2
- solution:

```
doubleNum  x = x + x      -- doubling a number
```

```
doubleExpr e = Plus e e    -- doubling an expression
```

- evaluation: `doubleExpr` =



Creating New Values – Person Example

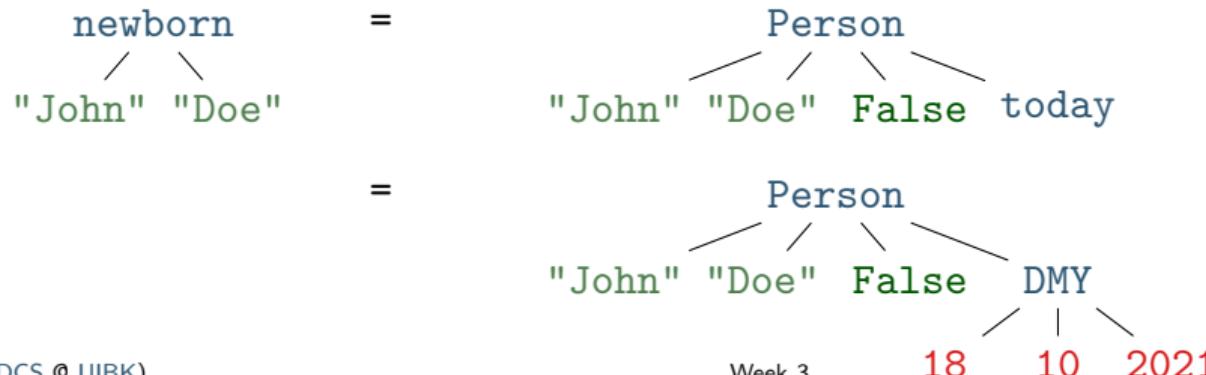
- consider Person datatype of last lecture

```
data Date = DMY Int Int Integer  
data Person = Person String String Bool Date
```

- task: define a function that takes first- and lastname and creates a (value of type) Person representing a newborn with that name
- solution:

```
today = DMY 18 10 2021  
newborn fName lName = Person fName lName False today
```

- evaluation



Function Definitions using Patterns

- so far all functions definitions have been of the shape
`funName x1 ... xN = expr`
where `x1 ... xN` is a list of **parameters**
- in these definitions we cannot inspect the structure of the input
- aim: define **functions depending on structure** of input
- example using vehicle datatype (with cars, bicycles and trucks)
 - task: convert a vehicle into a string
 - algorithm:
 - if the input is a **car with *x* PS**, then return "a car with *x* PS"
 - if the input is a **bicycle**, then return "a bicycle"
 - if the input is a **truck with *x* wheels**, then return "a(n) *x*-wheel truck"
- in Haskell, structure of trees are described by **patterns**
- the question whether some input tree fits a pattern is called **pattern matching**

Patterns

- a **pattern** is an expression of one of the following forms
 - **x** parameter name as in a function definition
 - **_** underscore
 - **CName pat₁ ... pat_N** constructor application with patterns **pat₁ ... pat_N** as arguments
 - **x@pat** parameter name followed by @ and pattern

where

- all parameter names occur at most once
- numbers, strings, and characters can be interpreted as constructors
- parentheses might be required for nested patterns
- examples
 - **Car brand ps** an arbitrary car
 - **Car _ ps** an arbitrary car (no interest in brand)
 - **Car BMW 100** a BMW with exactly 100 PS
+ is not a constructor ✗
 - **Car _ (50 + 50)**
 - **Person "John" lName _ _** a person whose first name is John
 - **p@(Person _ _ _ (DMY 18 10 _))** a person **p** to congratulate
duplicate parameters ✗
 - **Person name name _ _**

Pattern Matching

- pattern matching is an algorithm that determines whether an expression matches a pattern
- during pattern matching a substitution of parameter names to expressions is created, written as $x_1/\text{expr}_1, \dots, x_N/\text{expr}_N$
(here, / is not the division operator but the substitute operator)
- pattern matching algorithm for pattern **pat** and expression **expr**
 - **pat** is parameter **x**: matching succeeds, substitution is x/expr
 - **pat** is **_**: matching succeeds, empty substitution
 - **pat** is $x@\text{pat}_1$: matching succeeds if **pat₁** matches **expr**;
add x/expr to resulting substitution
 - **pat** is **CName pat₁ ... pat_N**:
 - if **expr** is **OtherCName ...** with $\text{CName} \neq \text{OtherCName}$ then match fails
 - if **expr** is **CName expr₁ ... expr_N** then
match **expr₁** with **pat₁**, ..., match **pat_N** with **expr_N**;
if all of these matches succeed then succeed with merged substitution, otherwise match fails
 - otherwise, first evaluate **expr** until outermost constructor is fixed
- remark: algorithm itself is described via pattern matching

Pattern Matching – Examples

- matching expression `Car BMW (20 + 80)` with some patterns
 - pattern `x`: success with substitution `x / Car BMW (20 + 80)`
 - pattern `Car brand ps`: success with substitution `brand / BMW, ps / (20 + 80)`
 - pattern `Car brand _`: success with substitution `brand / BMW`
 - pattern `Car Audi _`: failure
 - pattern `Car _ 100`: success with empty substitution, triggers evaluation
- matching expression `Person "Liz" "Ball" True (DMY 18 10 1970)` with some patterns
 - pattern `Person "John" 1Name _ _`: fails
 - pattern `p@(Person _ _ _ (DMY 18 10 _))`: success with substitution `p / Person "Liz" "Ball" True (DMY 18 10 1970)`

Function Definitions with Pattern Matching

- so far all functions definitions have been of shape

funName $x_1 \dots x_N = \text{expr}$

- now add two generalizations

- a function definition has the shape

funName $p_1 \dots p_N = \text{expr}$ (*)

where all parameters in patterns $p_1 \dots p_N$ occur at most once

- there can be several equations for the same function
- evaluation of funName $\text{expr}_1 \dots \text{expr}_N$ via function equation (*)
 - if p_1 matches $\text{expr}_1, \dots, p_N$ matches expr_N via some substitutions, then the equation is applicable and funName $\text{expr}_1 \dots \text{expr}_N$ is replaced by rhs expr with the merged substitution applied
 - otherwise, (*) is not applicable
- evaluation of funName $\text{expr}_1 \dots \text{expr}_N$
 - apply first equation that is applicable (tried from top to bottom)
 - if no equation is applicable, abort computation with error

Function Definitions – Example on Person

```
data Date = DMY Int Int Integer  
data Person = Person String String Bool Date  
data Option  = Some Integer | None
```

- task: change the last name of a person

```
changeName lName (Person fName _ m b) =  
    Person fName lName m b
```

remark: data is never changed but newly created

- task: compute the age of a person in years, if it is his or her birthday, otherwise return nothing

```
ageYear (Person _ _ _ (DMY 18 10 y)) = Some (2021 - y)
```

```
ageYear _ = None
```

remark: here the order of equations is important

- task: create a greeting for a person

```
greeting p@(Person name _ _ _) = gHelper name (ageYear p)  
gHelper n None = "Hello " ++ n  
gHelper n (Some a) = "Hi " ++ n ++ ", you turned " ++ show a
```

remark: (++) concatenates two strings, show converts values to strings

Merging Substitutions and Equality

- consider the following code for testing equality of two values

```
equal x x = True
```

```
equal _ _ = False
```

- consider evaluation of equal 5 7

- first argument: x matches 5, obtain substitution x / 5

- second argument: x matches 7, obtain substitution x / 7

- merging these substitutions is not possible: x / ???

- Haskell avoids problem of non-mergeable substitutions by the distinct-parameter-restriction in lhss, i.e., above definition is not allowed in Haskell

- correct solution for testing on equality

- use (==), a built-in operator to compares two values of the same type, the result will be of type Bool

- for comparison of user-defined datatypes, replace deriving Show by deriving (Show, Eq)

- examples: 5 == 7, "Peter" == name, . . . , but not "five" == 5

Function Definitions – Example on Bool

- consider built-in datatype `data Bool = True | False`

- consider function for conjunction of two Booleans

```
conj True  b = b
```

```
conj False _ = False
```

- example evaluation (numbers are just used as index)

```
conj1 (conj2 True False) (conj3 True True)
```

-- check which equation is applicable for conj1

-- first equation triggers evaluation of first argument of conj1 (True)

-- check which equation is applicable for conj2

-- first equation is applicable with substitution b/False

```
= conj1 False (conj3 True True)
```

-- now see that only second equation is applicable for conj1

```
= False
```

- remark: many Boolean functions are predefined, e.g.,

`(&&)` (conjunction), `(||)` (disjunction),

`(/=)` (exclusive-or), `not` (negation)

Function Definitions by Case Analysis

- design principle for functions:

define (several) equation(s) to cover all possible shapes of input

- example

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
weekend Sat = True
```

```
weekend Sun = True
```

```
weekend _ = False
```

- example: first element of a list

```
data List = Empty | Cons Integer List
```

```
first (Cons x xs) = x
```

```
first Empty = error "first on empty list"
```

- **error** takes a string to deliver sensible error message upon evaluation

- without second defining equation, `first Empty` results in generic "non-exhaustive patterns" exception

Recursive Function Definitions

- example: length of a list

```
len Empty = 0
```

```
len (Cons x xs) = 1 + -- the length of the list xs
```

- potential problem: we would like to apply a function that we are currently defining
- this is allowed in programming and called **recursion**:
a function definition that invokes itself

```
len Empty = 0
```

```
len (Cons x xs) = 1 + len xs -- len xs is recursive call
```

- make sure to have smaller arguments in recursive calls
- evaluation is as before

$$\begin{aligned} & \text{len}(\text{Cons } 1 (\text{Cons } 7 (\text{Cons } 9 \text{ Empty}))) \\ &= 1 + (\text{len}(\text{Cons } 7 (\text{Cons } 9 \text{ Empty}))) \\ &= 1 + (1 + (\text{len}(\text{Cons } 9 \text{ Empty}))) \\ &= 1 + (1 + (1 + (\text{len} \text{ Empty}))) \\ &= 1 + (1 + (1 + 0)) = 1 + (1 + 1) = 1 + 2 = 3 \end{aligned}$$

Recursive Function Definitions – Example Append

- task: append two lists, e.g., appending [1, 5] and [3] yields [1, 5, 3]
- solution: pattern matching and recursion on first argument

```
append Empty ys = ys
```

```
append (Cons x xs) ys = Cons x (append xs ys)
```

- example evaluation

```
append (Cons 1 (Cons 3 Empty)) (Cons 2 (Cons 7 Empty))
= Cons 1 (append (Cons 3 Empty) (Cons 2 (Cons 7 Empty)))
= Cons 1 (Cons 3 (append Empty (Cons 2 (Cons 7 Empty))))
= Cons 1 (Cons 3 (Cons 2 (Cons 7 Empty)))
```

Recursive Function Definitions – Evaluating Expr

- consider datatype for expressions

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr
```

- task: evaluate expression
- solution:

```
eval (Number x)      = x  
eval (Plus e1 e2)   = eval e1 + eval e2  
eval (Negate e)     = - eval e
```

Recursive Function Definitions – Expr to List

- consider datatype for expressions

```
data Expr =  
    Number Integer  
  | Plus Expr Expr  
  | Negate Expr
```

- task: create list of all numbers that occur in expression
- solution:

```
numbers (Number x)      = Cons x Empty  
numbers (Plus e1 e2)   = append (numbers e1) (numbers e2)  
numbers (Negate e)     = numbers e
```

Summary

- function definitions by case analysis via **pattern matching**
 - patterns describe shapes of trees
 - multiple defining equations allowed, tried from top to bottom
- function definitions can be **recursive**
 - `funName ... = ... (funName ...) ... (funName ...) ...`
 - arguments in recursive call should be smaller than in lhs



Functional Programming

Week 4 – Polymorphism

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- function definitions by pattern matching
 - allow for several equations for each function
 - equations are tried from top to bottom
- patterns
 - `x, _, CName pat1 ... patN, x@pat`
 - parameter names must be distinct
 - patterns describe shape of inputs
- recursion
 - in a defining equation of function `f` one can use `f` already in the rhs
$$f \text{ pat1 ... patN} = \dots (f \text{ expr1 ... exprN}) \dots$$
 - the arguments in each **recursive call** should be smaller than in the lhs

List Examples

- task 1: append two lists, e.g., appending [1, 5] and [3] yields [1, 5, 3]
- solution: pattern matching and recursion on first argument

```
data List = Empty | Cons Integer List
append Empty ys      = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

interpretation of the second equation

- first append the remaining list `xs` and `ys` (`append xs ys`), afterwards insert `x` in front of the result
- the first element of the resulting list is `x`, so the result is `Cons x ...`, and for `...` just append `xs` and `ys`
- task 2: determine last element of list
- solution: consider three cases (empty, singleton, list with at least two elements)

```
lastElem (Cons _ xs@(Cons _ _)) = lastElem xs
lastElem (Cons x _)              = x      -- here the order of eq. matters
lastElem Empty                   = error "empty list has no last element"
```

Example – Datatypes Expr and List

- consider datatype for expressions

```
data Expr = Number Integer | Plus Expr Expr | Negate Expr
```

- task: create list of all numbers that occur in expression
- solution

```
numbers (Number x)      = Cons x Empty
numbers (Plus e1 e2)    = append (numbers e1) (numbers e2)
numbers (Negate e)      = numbers e
```

- remarks
 - `numbers :: Expr -> List`
 - the rhs of the first equation must be `Cons x Empty` and not just `x`:
the result must be a list of numbers
 - `numbers` (and also `append`) is defined via **structural recursion**:
invoke the function recursively for each recursive argument of a datatype
(`e1` and `e2` for `Plus e1 e2`, and `e` for `Negate e`, but not `x` of `Number x`)

Decomposition and Auxiliary Functions

- during the definition of new functions, often some functionality is missing
- task: define a function to remove all duplicates from a list
- solution:

```
remdups Empty = Empty
remdups (Cons x xs) = Cons x (remove x (remdups xs))
-- subtask: define "remove x xs" to delete each x from list xs
remove x Empty = Empty
remove x (Cons y ys) = rHelper (x == y) y (remove x ys)
rHelper True _ xs = xs
rHelper False y xs = Cons y xs
```

- remarks
 - solution above uses structural recursion: `remdups (Cons x xs)` invokes `remdups xs`
 - alternative solution with non-structural recursion: replace 2nd equation by
`remdups (Cons x xs) = Cons x (remdups (remove x xs))`

Parametric Polymorphism

Limitations of Datatype Definitions

- task: define a datatype for lists of **numbers** and a function to compute their length

```
data IntList = EmptyIL | ConsIL Integer IntList  
lenIL EmptyIL      = 0  
lenIL (ConsIL _ xs) = 1 + lenIL xs
```

- task: define a datatype for lists of **strings** and a function to compute their length

```
data StringList = EmptySL | ConsSL String StringList  
lenSL EmptySL      = 0  
lenSL (ConsSL _ xs) = 1 + lenSL xs
```

- observations

- the datatype and function definitions are nearly identical:
only difference is type of elements (**Integer/String**) and type/function/constructor names
- creating a copy for each new element type is not desirable for many reasons
 - writing the same functionality over and over again initially is tedious and error-prone
 - changing the implementation later on is even more tedious – integrate changes for every element type
- aim: define one **generic** list datatype and functions on these generic lists – **polymorphism**

Two Kinds of Polymorphism

- **parametric polymorphism**
 - key idea: provide **one definition** that can be used in various ways
 - examples
 - a datatype definition for arbitrary lists (parametrized by type of elements)
 - a datatype definition for arbitrary pairs (parametrized by two types)
 - ...
 - a function definition that works on parametric lists, pairs, ...;
examples: length, append two lists, first component of pair, ...
- **ad-hoc polymorphism**
 - key idea: provide similar functionality under **same name** for different types
 - examples
 - `(==)` is equality operator; different implementations for strings, integers, floats, ...
 - `(+)` is addition operator; different implementations for integers, floats, ...
 - `minBound` gives smallest value for bounded types; different implementations for `Int`, `Char`, ...
 - advantage: uniform access (instead of `==Int`, `==String`, `==Double`)

Type Variables

- definition of polymorphic types and functions requires **type variables**
- type variables
 - start with a lowercase letter; usually a single letter is used, e.g., **a**, **b**, ...
 - a type variable represents any type
 - type variables can be substituted by (more concrete) types
- type **ty1** is **more general** than **ty2** if **ty2** can be obtained from **ty1** by a type substitution
- **important:** it is allowed to replace generic types with more concrete ones;
whenever **expr :: ty1** and **ty1** is more general than **ty2** then **expr :: ty2**
- types **ty1** and **ty2** are **equivalent** if **ty1** is more general than **ty2** and vice versa
- examples
 - **a** is more general than any other type
 - **a -> b -> a** is more general than **Int -> Char -> Int**, **a -> Bool -> a**, **c -> c -> c**
 - **a -> b -> a** is equivalent to **b -> a -> b**
 - **a -> b -> a** is not more general than **a -> b -> c**
 - **someFun x y = x** is a function with type **a -> b -> a**
 - **otherFun True x = x** is a function with type **Bool -> a -> a**

Types Revisited

- already known: definition of (basic) Haskell expressions and patterns
- now: definition of **types**
- prerequisite: **type constructors** (`TConstr`)
 - similar to (value-)constructors (`Cons`, `True`, ...)
 - start with uppercase letter
 - have a fixed arity
 - different to constructors: type constructors are used to construct types
- a **Haskell type** has one of the following three shapes
 - `a` a type variable
 - `TConstr ty1 ... tyN` a type constructor of arity `N` applied to `N` types
 - `(ty)` parentheses are allowed
- examples (type constructors of arity 0: `Char`, `Bool`, `Integer`, ...; arity 2: `->`)
 - `->` without the two arguments is not a type
 - `a -> Int` – type of functions that take an arbitrary input and deliver an `Int`
 - `Bool -> (a -> Int)` – type of f. that take a `Bool` and deliver a f. of type `a -> Int`
 - `Bool -> a -> Int` – same as above (!), `->` associates to the right
 - `(Bool -> a) -> Int` – take a f. of type `Bool -> a` as input, deliver an `Int`

Type Constraints and Predefined Type Classes

- often a type variable **a** needs to be constrained to belong to a certain **type class**
 - a type **a** for which (+), (-), (*) is defined: type class **Num a**
 - a type **a** for which (/) is defined: type class **Fractional a**
 - a type **a** for which (==), (/=) is defined: type class **Eq a**
 - a type **a** for which (<), (<=), ... is defined: type class **Ord a**
 - a type **a** for which **show :: a -> String** is defined: type class **Show a**
- notation of type constraints in Haskell via =>
- examples

```
f x y = x          -- f :: a -> b -> a
g x y = x + y - 3 -- g :: Num a => a -> a -> a
h x y = "cmp is " ++ show (x < y) -- h :: Ord a => a -> a -> String
i x = "result: " ++ show (x + 3)   -- i :: (Num a, Show a) => a -> String
```

- type substitutions need to respect type constraints

- **g False True** is not allowed since **Bool** is not an instance of **Num**
- **i (5 :: Int)** is allowed since **Int** is an instance of both **Num** and **Show**

Datatypes with Parametric Polymorphism

- previous definition

```
data TName =  
    CName1 type1_1 ... type1_N1  
  | ...  
  | CNameM typeM_1 ... typeM_NM
```

- new definition

```
data TConstr a1 ... aK =  
    CName1 type1_1 ... type1_N1  
  | ...  
  | CNameM typeM_1 ... typeM_NM
```

- new definition is more general (K can be zero)
- $a1 \dots aK$ have to be distinct type variables
- $TConstr$ is a new type constructor with arity K
- $a1 \dots aK$ can be used in any of the types $type1_J$, but no other type variables
- $CName1 :: type1_1 \rightarrow \dots \rightarrow type1_N1 \rightarrow TConstr a1 \dots aK$, etc.

Examples using Parametric Polymorphism

Parametric Lists

```
data List a = Empty | Cons a (List a)
```

- List is unary type constructor
- example types
 - List a – list of arbitrary elements
 - List Integer – list of integers
 - List Bool – list of Booleans
 - List (List Integer) – list whose elements are lists of integers
- type of constructors
 - Empty :: List a
 - Cons :: a -> List a -> List a
- example programs

```
len :: List a -> Int      -- parametric function definition
len Empty = 0
len (Cons _ xs) = 1 + len xs
first :: List a -> a
first (Cons x _) = x
```

Parametric Lists Continued

```
data List a = Empty | Cons a (List a)
```

- function definitions can enforce certain type restrictions
 - example: replace all occurrences of *x* by *y* in a list

```
replace :: Eq a => a -> a -> List a -> List a
replace _ _ Empty = Empty
replace x y (Cons z zs) = rHelper (x == z) y z (replace x y zs)
rHelper True y _ xs = Cons y xs
rHelper False _ z xs = Cons z xs
```

- type constraint *Eq a* is required since list elements are compared via `==`
- function definitions can enforce a concrete element type
 - example: replace all occurrences of 'A' by 'B' in a list

```
replaceAB :: List Char -> List Char
replaceAB xs = replace 'A' 'B' xs
```

Lists in Haskell

- the list type from previous two slides is actually predefined in Haskell
- only difference: names
 - instead of `List a` one writes `[a]`
 - instead of `Empty` one writes `[]`
 - instead of `Cons x xs` one writes `x : xs` (and `:` is called “Cons”)
 - in total

```
data [a] = [] | a : [a]
```

- list constructor `(:)` associates to the right:
 $1 : 2 : 3 : [] = 1 : (2 : (3 : []))$
- special list syntax for finite lists: $[1, 2, 3] = 1 : 2 : 3 : []$
- example: append on Haskell lists

```
append :: [a] -> [a] -> [a]
```

```
append [] ys      = ys
```

```
append (x : xs) ys = x : append xs ys
```

Tuples

- tuples are a frequently used datatype to provide several outputs at once; example: a division-with-remainder function should return two numbers, the quotient and the remainder
- it is easy to define various tuples in Haskell

```
data Unit = Unit           -- tuple with 0 entries
data Pair a b = Pair a b   -- tuple with 2 entries
data Triple a b c = Triple a b c -- tuple with 3 entries
```

- example: find value of key 'y' in list of key/value-pairs

```
findY :: [Pair Char a] -> a
findY []             = error "..."
findY (Pair 'y' v : _) = v
findY (_ : xs)       = findY xs
```

remark: one would usually define a function to search for arbitrary keys, but this is a task in your homework: Assignment ~ [Pair String Integer]

Tuples in Haskell

- tuples are predefined in Haskell (so there is no need to define `Pair`, `Triple`, ...)
- for every $n \neq 1$ Haskell provides:
 - a type constructor `(, ...,)` (with n entries)
 - a (value) constructor `(, ...,)` (with n entries)
- examples
 - `Pair a b` and `Triple a b c` are equivalent to `(a,b)` and `(a,b,c)`
 - `(5, True, "foo") :: (Int, Bool, String)`
 - `() :: ()`
 - `(5)` is just the number `5`, so no 1-tuple
 - `(1,2,3)` is neither the same as `((1,2),3)` nor as `(1,(2,3))`
- example program from previous slide using predefined tuples

```
findY :: [(Char, a)] -> a
findY []           = error "..."
findY (('y', v) : _) = v
findY (_ : xs)     = findY xs
```

Type Synonyms

- Haskell offers a mechanism to create **synonyms of types** via the keyword **type**
type TConstr a1 ... aN = ty
 - **TConstr** is a fresh name for a type constructor
 - **a1 ... aN** is a list of type variables
 - **ty** is a type that may contain any of the type variables
 - there is no new (value-)constructor
 - **ty** may not include **TConstr** itself, i.e., no recursion allowed
- example

```
data PersonDT = Person (String, Integer) -- name & year of birth
type PersonTS = (String, Integer)


- the types PersonTS and (String, Integer) are identical,  
    ((("Jane", 1980) :: PersonTS) :: (String, Integer)) :: PersonTS
- the types PersonDT is different to both (String, Integer) and PersonTS;  
    ("Bob", 2002) is of type PersonTS, but not of type PersonDT;  
    Person ("Bob", 2002) is of type PersonDT, but not of type PersonTS

```

Type Synonyms – Applications, Strings

- example applications of type synonyms

- avoid creation of new datatypes: `type Person = (String, Integer)`

- increase readability of code

```
type Month = Int
```

```
type Day    = Int
```

```
type Year   = Int
```

```
type Date  = (Day, Month, Year)
```

```
createDate :: Day -> Month -> Year -> Date
```

```
createDate d m y = (d,m,y)
```

-- createDate is logically equivalent to the following function,
-- but the type synonyms help to make the code more readable

```
createDate :: Int -> Int -> Int -> (Int, Int, Int)
```

```
createDate x y z = (x,y,z)
```

- in Haskell: `type String = [Char]`

- in particular "hello" is identical to `['h', 'e', 'l', 'l', 'o']`

- all functions on lists can be applied to `Strings` as well, e.g. `(++) :: [a] -> [a] -> [a]`

```
data Maybe a = Nothing | Just a
```

- `Maybe` is predefined Haskell type to specify optional results
- example application: safe division without runtime errors

```
divSafe :: Double -> Double -> Maybe Double
```

```
divSafe x 0 = Nothing
```

```
divSafe x y = Just (x / y)
```

```
data Expr = Plus Expr Expr | Div Expr Expr | Number Double
```

```
eval :: Expr -> Maybe Double
```

```
eval (Number x) = Just x
```

```
eval (Plus x y) = plusMaybe (eval x) (eval y)
```

```
eval (Div x y) = divMaybe (eval x) (eval y)
```

```
plusMaybe (Just x) (Just y) = Just (x + y)
```

```
plusMaybe _ _ = Nothing
```

```
divMaybe (Just x) (Just y) = divSafe x y
```

```
divMaybe _ _ = Nothing
```

```
data Either a b = Left a | Right b
```

- Either is predefined Haskell type for specifying alternative results
- example application: model optional values with error messages

```
divSafe :: Double -> Double -> Either String Double  
divSafe x 0 = Left ("don't divide " ++ show x ++ " by 0")  
divSafe x y = Right (x / y)
```

```
data Expr = Plus Expr Expr | Div Expr Expr | Number Double
```

```
eval :: Expr -> Either String Double
```

```
eval (Number x) = Right x
```

```
eval (Plus x y) = plusEither (eval x) (eval y)
```

```
eval (Div x y) = divEither (eval x) (eval y)
```

```
divEither (Right x) (Right y) = divSafe x y
```

```
divEither e@(Left _) _ = e -- new case analysis required
```

```
divEither _ e = e
```

```
plusEither ... = ...
```

Algebraic Datatypes

- datatype definitions are called **algebraic** because they can be purely constructed via products and sums
 - products: in math $A \times B$, in Haskell `(a, b)`
 - sums: in math $A \uplus B$, in Haskell `Either a b`
- it can be shown that in datatype definitions it would be sufficient to permit only one constructor with only one argument; idea:
 - encode many arguments into a single tuple, i.e., nested products
 - encode the different constructors using nested sums
- example encoding of list datatype
 - definition: `data List a = ListC (Either () (a, List a))`
 - encoding of `Empty`: `ListC (Left ())`
 - encoding of `Cons x xs`: `ListC (Right (x, xs))`
- application of such a construction:
compiler development where datatype definitions can be simplified by frontend;
backend only has to handle definitions of shape `data TConstr a1 ... aN = CName ty`

Summary

- usage of type variables and parametric polymorphism
 - datatypes with type variables
 - polymorphic functions (`f :: (Eq a, Show b) => a -> a -> b -> String, ...)`)
- predefined datatypes
 - lists `[a]`
 - tuples `(..., ..., ...)`
 - option type `Maybe a`
 - sum type `Either a b`
- predefined type classes
 - arithmetic except division: `Num a`
 - arithmetic including division: `Fractional a`
 - equality between elements: `Eq a`
 - smaller than and greater than: `Ord a`
 - conversion to `Strings`: `Show a`
- type synonyms via `type`



Functional Programming

Week 5 – Expressions, Recursion on Numbers

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- type variables: `a`, `b`, ... represent any type
- parametric polymorphism
 - one implementation that can be used for various types
 - polymorphic datatypes, e.g., `data List a = Empty | Cons a (List a)`
 - polymorphic functions, e.g., `append :: List a -> List a -> List a`
 - type constraints, e.g., `sumList :: Num a => List a -> a`
- predefined types: `[a]`, `Maybe a`, `Either a b`, `(a1, ..., aN)`
- predefined type classes
 - arithmetic except division: `Num a`
 - arithmetic including division: `Fractional a`
 - equality between elements: `Eq a`
 - smaller than and greater than: `Ord a`
 - conversion to `Strings`: `Show a`

This Lecture

- type synonyms
- expressions revisited
- recursion involving numbers

Type Synonyms

Type Synonyms

- Haskell offers a mechanism to create **synonyms of types** via the keyword **type**
type TConstr a1 ... aN = ty

- TConstr** is a fresh name for a type constructor
- a1 ... aN** is a list of type variables
- ty** is a type that may contain any of the type variables
- there is no new (value-)constructor
- ty** may not include **TConstr** itself, i.e., no recursion allowed

new apart constructor

- example

```
data PersonDT = Person (String, Integer) -- name & year of birth  
type PersonTS = (String, Integer)
```

- the types **PersonTS** and **(String, Integer)** are identical,
((("Jane", 1980) :: PersonTS) :: (String, Integer)) :: PersonTS
- the types **PersonDT** is different from both **(String, Integer)** and **PersonTS**;
("Bob", 2002) is of type **PersonTS**, but not of type **PersonDT**;
Person ("Bob", 2002) is of type **PersonDT**, but not of type **PersonTS**

Type Synonyms – Applications, Strings

- example applications of type synonyms

- avoid creation of new datatypes: `type Person = (String, Integer)`

- increase readability of code

```
type Month = Int
```

```
type Day = Int
```

```
type Year = Int
```

```
type Date = (Day, Month, Year)
```

type Day nb - [(nb)]

```
createDate :: Day -> Month -> Year -> Date
```

```
createDate d m y = (d, m, y)
```

-- createDate is logically equivalent to the following function,
-- but the type synonyms help to make the code more readable

```
createDate :: Int -> Int -> Int -> (Int, Int, Int)
```

```
createDate x y z = (x, y, z)
```

- in Haskell: `type String = [Char]`

- in particular "hello" is identical to `['h', 'e', 'l', 'l', 'o']`

- all functions on lists can be applied to `Strings` as well, e.g. `(++) :: [a] -> [a] -> [a]`

Expressions Revisited

Function Definitions Revisited

- current form of function definitions

```
f :: ty          -- optional type definition  
f pat11 ... pat1M = expr1    -- first defining equation  
...  
f pat1M ... patNM = exprN    -- last defining equation
```

where expressions consist of literals, variables, and function- or constructor applications

- observations

- case analysis only possible via patterns in left-hand sides of equations
- case analysis on right-hand sides often desirable
- work-around via auxiliary functions possible
- better solution: extension of expressions

if-then-else

- most primitive form of case analysis: if-then-else
- functionality: return one of two possible results, depending on a Boolean value

```
ite :: Bool -> a -> a -> a
```

```
ite True x y = x
```

```
ite False x y = y
```

zum gehört zu Maybe

- example application: lookup a value in a key/value-list

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

```
lookup x ((k,v) : ys) = ite (x == k) (Just v) (lookup x ys)
```

```
lookup _ _ = Nothing
```

- if-then-else is predefined: `if ... then ... else ...`

```
lookup x ((k,v) : ys) = if x == k then Just v else lookup x ys
```

- there is no if-then (without the else) in Haskell:

what should be the result if the Boolean is false?

- in C ?:

- remark: also `lookup` is predefined in Haskell;

Prelude content (functions, (type-)constructors, type classes, ...) is typeset in **green**

Vorlesung

Case Analysis via Pattern Matching

- observation: often case analysis is required on computed values
- implementation possible via auxiliary functions
- example: evaluation of expressions with meaningful error messages

```
data Expr a = Var String | ... -- Numbers, Addition, ...
eval :: Num a => [(String,a)] -> Expr a -> a
eval ass ...      = ...           -- all the other cases
eval ass (Var x) = aux (lookup x ass) x -- case analysis on lookup x ass
aux (Just i) _ = i
aux _ x = error ("assignment does not include variable " ++ x)
```

- disadvantages
 - local values need to be passed as arguments to auxiliary function (here: `x`)
 - **pollution of name space** by auxiliary functions
`(aux, aux1, aux2, auX, helper, fHelper, ...)`
- note: if-then-else is not sufficient for above example

Case Expressions

- **case expressions** support arbitrary pattern matching directly in right-hand sides

```
case expr of
    pat1 -> expr1
    ...
    patN -> exprN
```

- match **expr** against **pat1** to **patN** top to bottom
- if **patI** is first match, then case-expression is evaluated to **exprI**
- example from previous slide without auxiliary function

```
eval ass (Var x) = case lookup x ass of
    Just i -> i
    _ -> error ("assignment does not include variable " ++ x)
```

Variablen können noch den eingesetzten aufnehmen, wieder verwendet werden

The Layout Rule

- problem: define groups (of patterns, of function definitions, ...)
- items that start in same column are grouped together
- by increasing indentation, single item may span multiple lines
- groups end when indentation decreases
- script content is group, start nested group by `where`, `let`, `do`, or `of`
- **ignore layout:** enclose groups in '{' and '}' and separate items by ;'

Examples

with layout:

```
and b1 b2 = case b1 of
  True -> case b2 of
    True -> True
    False -> False
  False -> False
```

without layout:

```
and b1 b2 = case b1 of
  { True -> case b2 of
    { True -> True; False -> False };
    False -> False }
```

White-Space in Haskell

- because of layout rule, white-space in Haskell matters
(in contrast to many other programming languages)
- avoid tabulators in Haskell scripts
(tab-width of editor vs. Haskell-compiler)

Example

```
and1 b1 b2 = case b1 of
  True -> case b2 of
    True -> True
    False -> False
```

```
and2 b1 b2 = case b1 of
  True -> case b2 of
    True -> True
    False -> False
```

```
ghci> and1 True False
False
```

```
ghci> and2 True False
*** error: non-exhaustive patterns
```

The `let` Construct

- `let`-expressions are used for **local** definitions

- syntax

```
let
```

pat

 fname pat1 ... patN

in expr

Pattern Variablen nur...

= expr
= expr

*funktion und Variablen
nur innerhalb verwendet werden*

-- definition by pattern matching
-- function definition
-- result

- each `let`-expression may contain several definitions (order irrelevant)
- definitions result in new variable-bindings and functions
 - may be used in every expression `expr` above
 - are **not visible outside** `let`-expression

Number of Real Roots via `let` Construct

```
-- Prelude type and function for comparing two numbers
data Ordering = EQ | LT | GT
compare :: Ord a => a -> a -> Ordering

-- task: determine number of real roots of ax^2 + bx + c
numRoots a b c = let
    disc = b^2 - 4 * a * c      -- local variable
    analyse EQ = 1              -- local function
    analyse LT = 0
    analyse GT = 2
  in analyse (compare disc 0)
```

The `where` Construct

- `where` is similar to `let`, used for **local definitions**

- syntax

```
f pat1 .. patM = expr          -- defining equation (or case)
  where pat                  = expr -- pattern matching
        fname pat1 .. patN = expr -- function definitions
```

- each `where` may consist of several definitions (order irrelevant)
- local definitions introduce new variables and functions
 - may be used in every expression `expr` above
 - are **not visible outside** defining equation / case-expression
- remark: in contrast to `let`, when using `where` the defining equation of `f` is given first

```
numRoots a b c = analyse (compare disc 0) where
  disc = b^2 - 4 * a * c    -- local variable
  analyse EQ = 1            -- local function
  analyse LT = 0
  analyse GT = 2
```

Guarded Equations

- defining equations within a function definition can be **guarded**
- syntax:

```
fname pat1 ... patM "=
  | cond1 = expr1
  | cond2 = expr2
  |
  where ... -- optional where-block
```

where each **condI** is a Boolean expression

- whenever **condI** is first condition that evaluates to **True**, then result is **exprI**
- next defining equation of **fname** considered, if no condition is satisfied

```
numRoots a b c
  | disc > 0    = 2
  | disc == 0   = 1
  | otherwise   = 0           -- otherwise = True
  where disc = b^2 - 4 * a * c -- disc is shared among cases
```

mathematische Analogie

Example: Roots

- task: compute the sum of the roots of a quadratic polynomial
- solution with potential runtime errors

```
roots :: Double -> Double -> Double -> (Double, Double)
```

```
roots a b c
```

```
| a == 0 = error "not quadratic"  
| d < 0 = error "no real roots"  
| otherwise = ((- b - r) / e, (- b + r) / e)
```

```
where d = b * b - 4 * a * c  
      e = 2 * a  
      { r = sqrt d}
```

Hilfsmittel

```
sumRoots :: Double -> Double -> Double -> Double
```

```
sumRoots a b c = let  
    (x, y) = roots a b c -- pattern match in let  
    in x + y
```

- note: non-variable patterns in `let` are usually only used if they cannot fail; otherwise, use `case` instead of `let`

Example: Roots (Continued)

- task: compute the sum of the roots of a quadratic polynomial

- solution with explicit failure via `Maybe`-type

```
roots :: Double -> Double -> Double -> Maybe (Double, Double)
```

```
roots a b c
```

```
| a == 0 = Nothing
```

```
| d < 0 = Nothing
```

```
| otherwise = Just ((- b - r) / e, (- b + r) / e)
```

```
where d = b * b - 4 * a * c
```

```
    e = 2 * a
```

```
    r = sqrt d
```

```
sumRoots :: Double -> Double -> Double -> Maybe Double
```

```
sumRoots a b c =
```

```
  case roots a b c of           -- case for explicit error handling
```

```
    Just (x, y) -> Just (x + y) -- nested pattern matching
```

```
    n -> Nothing               -- can't be replaced by n -> n! (types)
```

Recursion on Numbers

Recursion on Numbers

- recursive function

`f pat1 ... patN = ... (f expr1 ... exprN) ...`

where input arguments should somehow be larger than arguments in recursive call:

`(pat1, ..., patN) > (expr1, ..., exprN)` -- for some relation `>`

- decrease often happens in one specific argument (the i -th argument always gets smaller)
 - so far the decrease in size was always w.r.t. **tree size**
 - length of list gets smaller
 - arithmetic expressions ([Expr](#)) are decomposed, i.e., number of constructors is decreased
 - if argument is a number (tree size is always 1), then still recursion is possible;
example: the **value** of number might decrease
 - frequent cases
 - some number i is decremented until it becomes 0 (while $i \neq 0 \dots i := i - 1$)
 - some number i is incremented until it reaches some bound n (while $i < n \dots i := i + 1$)

\triangleq while in C (i++)

Example: Factorial Function

- mathematical definition: $n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1, 0! = 1$

- implementation D: count downwards

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

- in every recursive call the value of **n** is decreased

- `factorial n` does not terminate if **n** is negative (hit Ctrl-C in ghci to stop computation)

- implementation U: count upwards, use accumulator (here: **r** stores accumulated (r)esult)

```
factorial :: Integer -> Integer
```

```
factorial n = fact 1 1 where
```

```
fact r i
```

```
| i <= n = fact (i * r) (i + 1)
```

```
| otherwise = r
```

- in every recursive call the value of **n - i** is decreased

- implementation U is equivalent to imperative program (with local variables **r** and **i**)

Example: Combined Recursion

- recursion on trees and numbers can be combined
- example: compute the n -th element of a list

1st element = 0

~~nth :: Int -> [a] -> a~~

`nth 0 (x : _) = x` -- indexing starts from 0

`nth n (_ : xs) = nth (n - 1) xs` -- decrease of number and list-length

`nth _ _ = error "no nth-element"`

- example: take the first n -elements of a list

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

`take _ [] = []`

`take n (x:xs)`

`| n <= 0 = []`

`| otherwise = x : take (n - 1) xs` -- decrease of number and list-length

- remarks

- both `take` and n -th element (!!) are predefined

- `drop` is predefined function that removes the first n -elements of a list

- equality: `take n xs ++ drop n xs == xs`

Example: Creating Ranges of Values

- task: given lower bound l and upper bound u , compute list of numbers $[l, l + 1, \dots, u]$
- algorithm: increment l until $l > u$ and always add l to front of list

```
range l u
| l <= u = l : range (l + 1) u
| otherwise = []
```

- remark: (a generalized version of) `range l u` is predefined and written `[l .. u]`
- example: concise definition of factorial function
 - `factorial n = product [1 .. n]`
where `product :: Num a => [a] -> a` computes the product of a list of numbers

Summary

- type synonyms via `type`
- expressions with local definitions and case analysis
- recursion on numbers



Functional Programming

Week 6 – Type Classes

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- layout rule: define blocks via indentation or via { ...; ...; ... }
- case-expressions: perform pattern matching in right-hand sides of defining equations
`case expr of { pat -> expr; ... ; pat -> expr }`
- local definitions with `let` and `where`

`let { pat = expr; fName pat ... pat = expr } in expr`

`fName pat ... pat = expr`
`where pat = expr`
`fName pat .. pat = expr`

- guarded equations

`fName pat ... pat`
| `cond = expr`
| `... = ...` -- + optional where-block

- recursion on numbers

Type Classes – Definition

Type Classes so Far

- brief introduction that there are type classes, e.g., `Num a`, `Eq a`, ...
- type classes are used to provide **uniform access** to functions that can be implemented differently for each type
- example
 - `(<)` :: `Ord a` => `a -> a -> Bool` is name of function for comparing two elements
 - each of the following types have a different implementation of `(<)`
 - `(<)` :: `Int -> Int -> Bool`
 - `(<)` :: `Char -> Char -> Bool`
 - `(<)` :: `Bool -> Bool -> Bool`
 - `(<)` :: `Ord a => [a] -> [a] -> Bool`
 - `(<)` :: `(Ord a, Ord b) => (a, b) -> (a, b) -> Bool`
- upcoming: **definition** of type classes
 - understand definition of existing type classes
 - specify new type classes
- upcoming: **instantiating** type classes
 - define an implementation for some type and some type class

Type Classes – Definition

- type classes are defined via the keyword `class`:

```
class TCName a where
    fName :: ty      -- type ty + description of fName
    ...
    lhs = rhs        -- optional default implementation
    ...
```

where

- `TCName` is a name for the type class, starting with uppercase letter
- `a` is a single type variable
- there are (several) type definitions for functions – without defining equations!
- for each function `fName` there should be some **informal description**
- there can be default implementations for each specified function `fName`
- when adding a type constraint `TCName a => ...`, then **all** functions `fName` are available
- defining a type class instance for some type requires implementation of all functions
- exception: functions that have default implementation can, but do not have to be implemented

Type Classes – Example Equality

```
class Equality a where
    equal :: a -> a -> Bool      -- equality
    different :: a -> a -> Bool   -- inequality
-- properties:
--   equal x x should evaluate to True
--   equal and different should be symmetric
--   exactly one of equal x y and different x y should be True
equal x y = not (different x y)  -- default implementation
different x y = not (equal x y)  -- default implementation
```

- if type constraint `Equality b` is added to type of function `f`, then both `equal :: b -> b -> Bool` and `different :: b -> b -> Bool` can be used in defining equation of `f`
- if concrete type `Ty` is an instance of `Equality`, then both `equal :: Ty -> Ty -> Bool` and `different :: Ty -> Ty -> Bool` can be used without adding type constraint
- in order to make some type an instance of `Equality`, at least one of `equal`, `different` has to be implemented for that type

Operator Syntax, Type Class Eq

- `Eq` is already predefined type class for equality
- only difference to `Equality`: operators are used instead of function names
- in Haskell every **operator can be turned into a function and vice versa**
 - parentheses turn arbitrary operator `&` into function name `(&)`
 - `a & b` is the same as `(&) a b`
 - `(&) :: ty` is used to specify the type of an operator
 - backticks turn some arbitrary function name `fName` into an operator ``fName``
 - `fName a b` is the same as `a `fName` b`
- consequence: in the following definition `(==)` and `(/=)` are just function names

```
class Eq a where
  (==) :: a -> a -> Bool    -- equality
  (/=) :: a -> a -> Bool    -- inequality
  x == y = not (x /= y)      -- default implementation
  x /= y = not (x == y)      -- default implementation
```

<http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html#t:Eq>

Type Class Hierarchies

- type classes can be defined hierarchically via type constraints
- syntax: `class (TClass1 a, ..., TClassN a) => TClassNew a where ...`
- consequences
 - type constraint `TClassNew a` implicitly adds type constraints `(TClass1 a, ..., TClassN a)`
 - when adding type constraint `TClassNew a`, all functions that are defined in one of `TClassNew, TClass1, ..., TClassN` become available
 - an instantiation of `TClassNew` for some type is only possible if that type is already an instance of all of `TClass1, ..., TClassN`
 - default implementations in `TClassNew` can make use of functions of `TClass1, ..., TClassN`

Example: Type Class Ord

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering -- data Ordering = LT | EQ | GT
    (<) :: a -> a -> Bool
    (≤) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (≥) :: a -> a -> Bool
    max :: a -> a -> a
    min :: a -> a -> a
    x < y = x ≤ y && x /= y
    x > y = y < x
    ...
    ...
```

- minimal complete definition: `compare` or `(≤)`
- note: default definition refers to `Eq` function `(/=)`
- <http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html#t:Ord>

Type Class Instances

- many types are instances of `Eq` and `Ord`
- examples

- `Eq Int`
- `Eq Char, Eq Integer, Eq Bool, ...`
- `Eq a => Eq [a]`

meaning: `Int` is an instance of `Eq`

- `Eq a => Eq (Maybe a), (Eq a, Eq b) => Eq (Either a b), ...`
- `Eq (), (Eq a, Eq b) => Eq (a,b), ...` for tuples of at most 15 entries
- `Ord Bool, Ord Char, Ord Integer, Ord Double, ...`
- `Ord a => Ord [a], (Ord a, Ord b) => Ord (Either a b), ...`
- `Ord (), (Ord a, Ord b) => Ord (a,b), ...` for tuples of at most 15 entries
- `Ord a => Ord [(String, Either (a,Int) [Double])]`
- functions are not instances of `Eq` and `Ord`: `Eq (Int -> Int)` does not hold

Type Class Hierarchy for Numbers

Type Class Num

- Num a provides basic arithmetic operations

- specification

```
(+)      :: a -> a -> a
(*)      :: a -> a -> a
(-)      :: a -> a -> a
abs      :: a -> a
signum   :: a -> a
fromInteger :: Integer -> a
negate   :: a -> a
```

- minimal complete definition: nearly everything, only negate or (-) can be dropped
- number literals are available for instances of Num class: 4715 :: Num a => a
- instances: Int, Integer, Float, Double

The Fractional Class – Division

- definition: `class Num a => Fractional a where ...`
- excerpt of functions
`(/)` $:: a \rightarrow a \rightarrow a$
- used for fractional literals: `5.72 :: Fractional a => a`
- instances: `Float, Double`

The Integral Class – Division with Remainder

- definition: `class (Num a, Ord a) => Integral a where ...`
- excerpt of functions
 - `toInteger` $:: a \rightarrow Integer$
 - `div` $:: a \rightarrow a \rightarrow a$
 - `mod` $:: a \rightarrow a \rightarrow a$
- instances: `Int, Integer`

Different behaviour when dividing by 0

- check: `1 `div` 0, 1 / 0, 1 / (-0), 0 == -0`

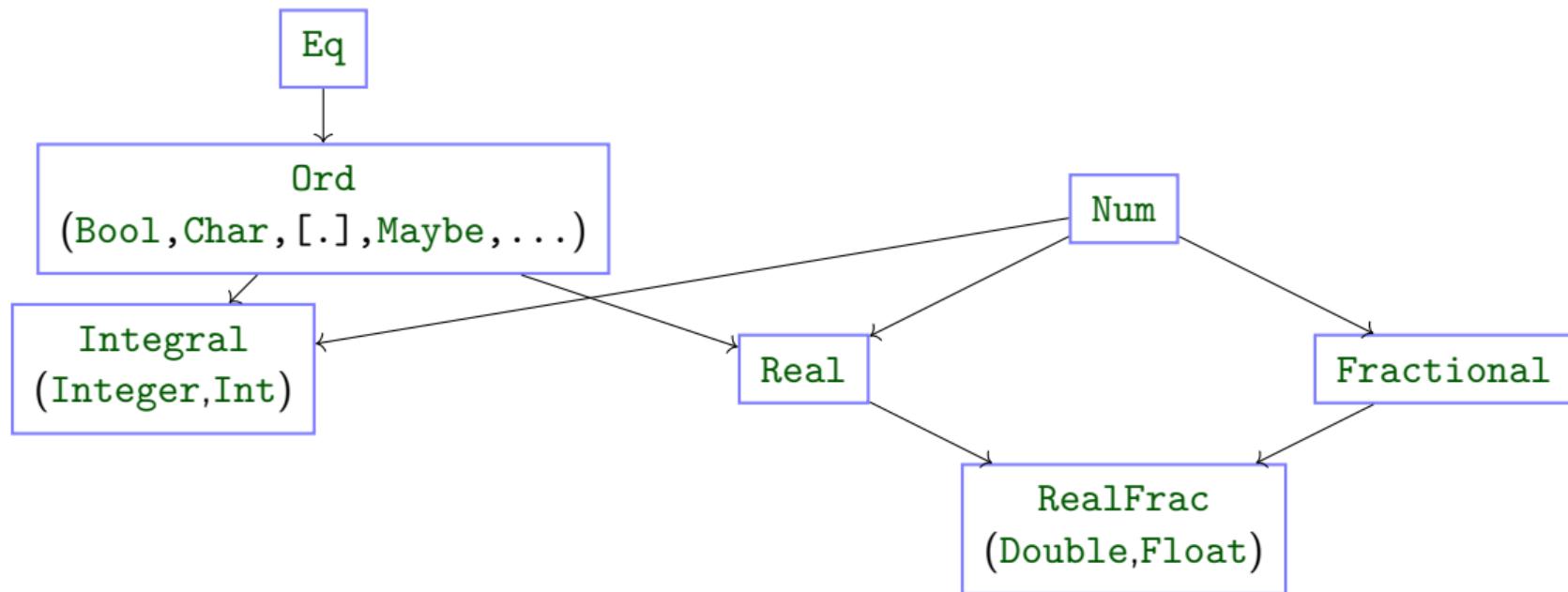
The RealFrac Class – Truncation

- definition: `class (Real a, Fractional a) => RealFrac a where ...`
- excerpt of functions
 - `floor :: Integral b => a -> b`
 - `ceiling :: Integral b => a -> b`
 - `round :: Integral b => a -> b`
- instances: `Float, Double`

Conversion of Numbers

- from integral to arbitrary number type
 - `fromIntegral :: (Integral a, Num b) => a -> b`
 - `fromIntegral x = fromInteger (toInteger x)`
- from real fractional numbers to integral numbers
 - `fractionalPart :: RealFrac a => a -> a`
 - `fractionalPart x = x - fromInteger (floor x)`

Excerpt of Class Hierarchy



- documentation under
<http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html>

Type Class Instantiation

Instantiating a Type Class

- so far: definitions of type classes, list of existing instantiations
- now: define own instances; syntax is as follows

```
instance (optional type constraints) => TClass (TConstr a1 .. aN) where  
    ... -- implementation of functions
```

where

- `a1 .. aN` are distinct type variables
- these may be used in a type constraint
- the implementation has to provide the implementations for each function `f :: ty` within the definition of `TClass a`
 - however, `f` has to be implemented for type `ty'` which is obtained by replacing `a` by `TConstr a1 .. aN` in `ty`
 - functions `f` that have a default implementation can be omitted
 - whenever a type constraint is used, then the implementation may use the functions of that type class
- writing `deriving TClass` in data type definition triggers generation of default instance; supported for type classes `Eq`, `Ord`, `Show`

Example: Complex Numbers

```
data Complex = Complex Double Double -- real and imaginary part

-- remark: we do not write deriving (Eq, Show),
-- but implement these instances on our own

instance Eq Complex where
    Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
    -- for (/=) use default implementation

instance Show Complex where
    show (Complex r i)
        | i == 0 = show r
        | r == 0 = show i ++ "i"
        | i < 0 = show r ++ show i ++ "i"
        | otherwise = show r ++ "+" ++ show i ++ "i"
```

Example: Complex Numbers Continued

```
instance Num Complex where
    Complex r1 i1 + Complex r2 i2 = Complex (r1 + r2) (i1 + i2)
    Complex r1 i1 * Complex r2 i2 =
        Complex (r1 * r2 - i1 * i2) (r1 * i2 + r2 * i1)
    fromInteger x = Complex (fromInteger x) 0
    negate (Complex r i) = Complex (negate r) (negate i)
    abs c = Complex (absComplex c) 0
    signum c@(Complex r i)
        | c == 0 = 0
        | otherwise = Complex (r / a) (i / a)
            where a = absComplex c

-- auxiliary functions must be defined outside
-- the class instantiation
absComplex (Complex r i) = sqrt (r^2 + i^2)
```

Example: Polymorphic Complex Numbers

```
data Complex a = Complex a a -- polymorphic: type a instead of Double

instance Eq a => Eq (Complex a) where
    Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
    -- comparing r1 and r2 (i1 and i2) requires equality on type a

-- for Show not only Show a is required, but also Ord a and Num a
instance (Show a, Ord a, Num a) => Show (Complex a) where
    show (Complex r i)
        | i == 0 = show r
        | r == 0 = show i ++ "i"
        | i < 0 = show r ++ show i ++ "i"
        | otherwise = show r ++ "+" ++ show i ++ "i"

instance (Floating a, Eq a) => Num (Complex a) where ...
    -- sqrt :: Floating a => a -> a, Floating a implies Num a
```

Example: Polymorphic Complex Numbers in Action

```
> (Complex 0 1)^2
```

```
-1.0
```

```
> 2 + 5 :: Complex Float
```

```
7.0
```

```
> abs (Complex 1 3) :: Complex Float
```

```
3.1622777
```

```
> abs (Complex 1 3) :: Complex Double
```

```
3.1622776601683795
```

```
> 2 * Complex 7 2.5
```

```
14.0+5.0i
```

```
> 2.4 * Complex 7 2.5
```

```
error: No instance for (Fractional (Complex Double))
```

Limitations of Type Class Instantiations

- instantiation: `instance ... => TClass (TConstr a1 .. aN)`
- the type variables **cannot** be replaced by more concrete types
 - example: the following instantiation is not permitted

-- show Boolean lists as bit-strings: "011" vs. "[False,True,True]"
`instance Show [Bool] where`
 `show (b : bs) = (if b then '1' else '0') : show bs`
 `show [] = ""`

- workaround via separate function: `showBits :: [Bool] -> String`
- each combination of type class and type can have **at most one instance**

- example: the following instantiation is not permitted

-- case-insensitive comparison of characters
`import Data.Char`
`instance Ord Char where` -- clashes with existing Ord Char instance
 `c <= d = toUpper c <= toUpper d`

- workaround: parametrise sorting algorithm, ... by order instead of using (\leq)

Summary

- several type classes are already defined in Prelude
- hierarchy of type classes for numbers
- new type classes can be user defined; content:
 - list of function names with types
 - description of what these functions should do
 - optionally: default implementations for some of the functions
- new type class instantiations can be added,
where both type class and type can either be user defined or predefined
- for each combination of type class and type, there can be **at most one implementation**
- conversion between operators and functions: (+) (25 `div` 3) 2



Functional Programming

Week 7 – Higher-Order Functions

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- type class definitions

```
class (...) => TCName a where
    fName :: ty      -- type ty + description of fName
    ...
    lhs = rhs        -- optional default implementation
    ...
```

- type class instantiations

```
instance (...) => TCName (TConstr a1 .. aN) where
    ... -- implementation of functions
```

- examples

- classes: Eq a, Num a, Integral a, RealFrac a, ...
- instances: Integral Int, Eq a => Eq (Maybe a), (Ord a, Ord b) => Ord (a,b), ...

- documentation:

<http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html>

- switch between operators and function names: (+) and `div`

Higher-Order Functions

Functions and Values

- functions take values as input and produce output values
 - values so far: numbers, characters, pairs, lists, user defined datatypes, ...
 - examples
 - `lookup :: Eq a => a -> [(a,b)] -> Maybe b`
 - `elem :: Eq a => [a] -> Bool`
- important extension: **functions are values**
- result: **higher-order functions**
 - functions can take other functions as input, e.g.,
`nTimes :: (a -> a) -> Int -> a -> a`
-- `nTimes f n x = f(...(f x))`
 - the result of a function can be a function, e.g.,
`compose :: (b -> c) -> (a -> b) -> (a -> c)`
-- `compose f g` is the function that takes an `x` and results in `f(g(x))`
- observations
 - higher-order functions are quite natural to define, e.g., `compose f g x = f (g x)`
 - higher-order functions are useful to avoid code duplication

Partial Application

- question: how to construct values that are functions?
- possible answer: **partial application**
- note: type constructor for functions (\rightarrow) associates to the right, cf. [lecture 4, slide 10](#)

$a \rightarrow b \rightarrow c \rightarrow d$ is identical to $a \rightarrow (b \rightarrow (c \rightarrow d))$

- note: function application associates to the left

$f \text{ expr1 expr2 expr3}$ is identical to $((f \text{ expr1}) \text{ expr2}) \text{ expr3}$

- example with parentheses added

```
average :: Double -> (Double -> Double)
(average x) y = (x + y) / 2
```

- partial application: `average` is applied on less than two arguments
- example **expressions**

- `average :: Double -> (Double -> Double)` no arguments applied
- `average 3 :: Double -> Double` 1 argument applied
- `(average 3) 5 :: Double` first 1 argument applied, then another one
- `average 3 5 :: Double` same as above

Sections, flip

- **sections** are a special form of partial applications in combination with operators &
- **(expr &)** is the same as **(&) expr**
- **(& expr)** is a function that takes an **x** and returns **x & expr**
- **(& expr)** is the same as **flip (&) expr**
 - **flip** is a predefined function that swaps the arguments of a binary function

```
flip :: (a -> b -> c) -> (b -> a -> c)
-- same as (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

- exception: **(- expr)** is not **flip (-) expr** but just the negated value of **expr**
- examples

- **(> 3)** test whether a number is larger than 3
- **(3 >)** test whether 3 is larger than a number
- **(3 -)** subtract something from 3
- **(- 3)** the number -3

Example: nTimes

```
nTimes :: (a -> a) -> Int -> a -> a
```

```
nTimes f n x
```

```
| n == 0 = x  
| otherwise = f (nTimes f (n - 1) x)
```

- observations

- `nTimes` uses standard recursion on numbers
- in the last line `f` is used twice
 - once as parameter of `nTimes`, where in `nTimes f` no argument is applied to `f`
 - once as the function which is applied to an argument: `otherwise = f (...)`

- application: implement other functions in more concise way

```
tower :: Integer -> Int -> Integer -- tower x n = x ^ (x ^ ... (x ^ 1))
```

```
tower x n = nTimes (x ^) n 1           -- n exponentiations with basis x
```

```
replicate :: Int -> a -> [a]          -- replicate n x = [x, ..., x]
```

```
replicate n x = nTimes (x :) n []       -- n insertions of x
```

Partial Application and Evaluation

- if defining equation of `f` is of shape `f pat1 ... patN` with `N` arguments, then evaluation of `f expr1 ... exprM` can only happen, if `M ≥ N`
- example `nTimes` and `tower`

```
nTimes f n x
| n == 0 = x
| otherwise = f (nTimes f (n - 1) x)
tower x n = nTimes (x ^) n 1
```

```
tower 4 2
= nTimes (4 ^) 2 1          -- (4 ^) cannot be evaluated!
= 4 ^ (nTimes (4 ^) 1 1)    -- evaluate second argument of ^
= 4 ^ (4 ^ (nTimes (4 ^) 0 1)) -- again, argument evaluation
= 4 ^ (4 ^ 1)
= 4 ^ 4
= 256
```

Partial Application and Evaluation, Continued

- if defining equation of f is of shape $f \ pat_1 \dots pat_N$ with N arguments, then evaluation of $f \ expr_1 \dots expr_M$ can only happen, if $M \geq N$
- example with $M > N$

```
selectFunction :: Bool -> (Int -> Int) -- same as Bool -> Int -> Int
selectFunction True  = (* 3)
selectFunction False = abs
```

```
selectFunction False (-2) -- M > N
= abs (-2)
= 2
```

- restriction: all defining equations of a function must have same number of arguments
- consequence: the following code is not allowed, although it would make sense

```
selectFunction' :: Bool -> Int -> Int
selectFunction' True = (* 3)
selectFunction' False x = 2 - x
```

Currying

- most of the time we defined functions in **curried form** (Haskell B. Curry, M. Schönfinkel)

$$f :: ty_1 \rightarrow \dots \rightarrow ty_N \rightarrow ty$$

- alternative is **tupled form**

$$f :: (ty_1, \dots, ty_N) \rightarrow ty$$

- observations

- partial application is only possible with curried form
- tupled form has advantage when passing logically connected values around

```
type Date = (Int, Int, Int)
```

```
differenceDate :: Date -> Date -> Int -- number of days between two dates  
-- but not: Int -> Int -> Int -> Int -> Int -> Int -> Int
```

- argument order is relevant in curried form: partial application only possible from left to right

- divide 1000 by something:
- division by 1000:
- alternative using `flip`:

```
div 1000  
let f x = div x 1000 in f  
flip div 1000
```

- rule of thumb: put arguments that are unlikely to change to the left

Anonymous Functions: λ abstractions

- example: apply n -times the function that given an x computes $3 \cdot (x + 1)$

- one possibility: local definition of a function

```
example :: Num a => Int -> a -> a
```

```
example = let f x = 3 * (x + 1) in nTimes f
```

-- this is equivalent to

```
example n y = let f x = 3 * (x + 1) in nTimes f n y
```

- annoying: creation of function names, here **f**

- alternative: creation of anonymous function via **λ abstraction**

- syntax: λ pat1 ... patN -> expr λ is written as \backslash in Haskell
- equivalent to: `let f pat1 ... patN = expr in f` for some fresh name **f**

```
example = nTimes (\ x -> 3 * (x + 1))
```

- difference between lambda abstractions and local function definitions

- recursion not expressible via lambda abstractions
- lambda abstractions do not require new function names

Example Higher-Order Functions and Applications

Generalize Common Programming Patterns

- consider the following tasks
 - multiply all list elements by 2
 - convert all characters in a string to upper case
 - compute a list of email addresses from a list of students
- possible implementation

```
multTwo [] = []
multTwo (x : xs) = 2 * x : multTwo xs

toUpperList [] = []
toUpperList (c : cs) = toUpper c : toUpperList cs

eMails [] = []
eMails (s : ss) = getEmail s : eMails ss
```

- observation: all of these functions are similar
- abstract version: apply some function on each list element
- aim: program the abstract version only once (will be a higher-order function), and then just instantiate this function for each task

The `map` Function

- `map` applies a function on each list element

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

- solve tasks from previous slide easily

```
multTwo = map (2 *)
toUpperList = map toUpper
eMails = map getEmail
```

- example evaluation

```
toUpperList "Hi"
= map toUpper "Hi"
= toUpper 'H' : map toUpper "i"
= 'H' : toUpper 'i' : map toUpper ""
= 'H' : 'I' : ""
= "HI"
```

The filter Function

- `filter` selects all elements of a list that satisfy some condition

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x : xs)
| f x =  x : filter f xs
| otherwise = filter f xs
```

- example applications

```
-- test whether some element is included in a list
```

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

```
-- the well known lookup function
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x xs = case filter (\ (k,_) -> x == k) xs of
[] -> Nothing
(_ ,v) : _ -> Just v
```

Application: Quicksort

- quicksort is an efficient sorting algorithm
- main idea: partition a non-empty list into small and large elements and sort recursively
- straight-forward implementation

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =    -- x is pivot element
    quicksort (filter (=< x) xs) ++ [x] ++ quicksort (filter (> x) xs)
```

- implementation might be tuned in several ways
 - use `partition` :: `(a -> Bool) -> [a] -> ([a], [a])` once instead of `filter` twice
 - parametrize order
 - `quicksortBy :: (a -> a -> Bool) -> [a] -> [a]`
 - `quicksort = quicksortBy (=<)`
 - take `random` pivot element, cf. lecture Algorithms and Data Structures

The Function Composition Operator (.)

- function composition is a higher-order function (in Haskell: (.)
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $(f . g) = \lambda x \rightarrow f(g x)$
- it takes two functions as input and returns a function
- in Haskell, function composition is often used to chain several function applications without explicit arguments
- example: given a number, first add 5, then compute the absolute value, then multiply it by 7, and finally convert it into a string and determine its length
- without composition: many parenthesis, not very readable
`\ x -> length (show ((abs (x + 5)) * 7))`
- written conveniently with function composition
`length . show . (* 7) . abs . (+ 5)`

Collection View

- often lists are used to encode collections of elements
- then one can process the whole collection via `map`, `filter`, `sum`, ... without looking at the position of the list elements
- list index function (!!) is rarely used in these applications
- in particular: do **not** write the following kind of loop

```
for (int i = 0; i < length; i++) {  
    xs[i] = someFun(xs[i]);  
}
```

as functional program

```
map (\ i -> someFun (xs !! i)) [0 .. length xs - 1]
```

but instead just write

```
map someFun xs
```

- the bad program needs $\sim \frac{1}{2}n^2$ evaluation steps for a list of length n : **lists \neq arrays!**

Application: Names of Good Students

- given a list of students, compute a sorted list of all names of students whose average grade is 2 or better
- implementation

```
data Student = ...
avgGrade :: Student -> Double
...
getName :: Student -> String
...
```

```
goodStudents :: [Student] -> [String]
goodStudents = qsort . map getName . filter (\ s -> avgGrade s <= 2)
```

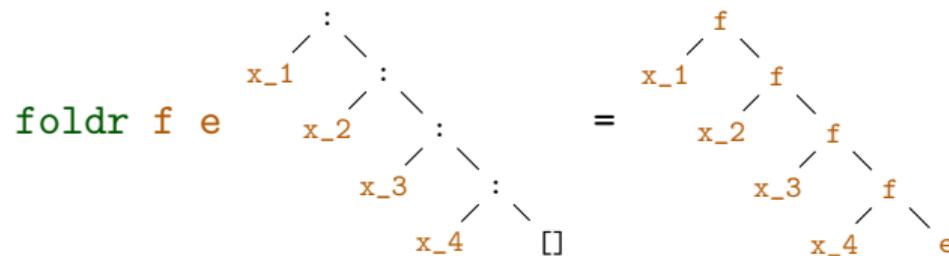
The foldr Function

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

`foldr f e [] = e`

`foldr f e (x : xs) = x `f` (foldr f e xs)`

- `foldr f e` captures structural recursion on lists
 - `e` is the result of the base case
 - `f` describes how to compute the result given the first list element and the recursive result
- `foldr f e` replaces `:` by `f` and `[]` by `e`



```
foldr f e [x_1, x_2, x_3, x_4]
= x_1 `f` (x_2 `f` (x_3 `f` (x_4 `f` e)))
```

Expressiveness of foldr

- `foldr f e` replaces `:` by `f` and `[]` by `e`;

```
foldr f e [x_1, x_2, x_3, x_4]
= x_1 `f` (x_2 `f` (x_3 `f` (x_4 `f` e)))
```
- `foldr f e` captures structural recursion on lists
- consequence: all function definitions that use structural recursion on lists can be defined via `foldr`
- example definitions via `foldr`

```
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
xs ++ ys = foldr (:) ys xs
length = foldr (\ _ -> (+ 1)) 0
map f = foldr ((:) . f) []
all f = foldr ((&&) . f) True
```

map via foldr in Detail

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

```
foldr f e [] = e
```

```
foldr f e (x : xs) = f x (foldr f e xs)
```

```
map f = foldr ((:) . f) []
```

```
map f [x_1, x_2, x_3]
```

```
= foldr ((:) . f) [] (x_1 : x_2 : x_3 : [])
```

```
= ((:) . f) x_1 (foldr ((:) . f) [] (x_2 : x_3 : []))
```

```
= (:) (f x_1) (foldr ((:) . f) [] (x_2 : x_3 : []))
```

```
= f x_1 : foldr ((:) . f) [] (x_2 : x_3 : [])
```

```
= ... = f x_1 : f x_2 : foldr ((:) . f) [] (x_3 : [])
```

```
= ... = f x_1 : f x_2 : f x_3 : foldr ((:) . f) [] []
```

```
= f x_1 : f x_2 : f x_3 : []
```

```
= [f x_1, f x_2, f x_3]
```

Variants of foldr

```
-- foldr from previous slide
```

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

```
foldr f e [x_1, x_2, x_3] = x_1 `f` (x_2 `f` (x_3 `f` e))
```

```
-- foldr without starting element, only for non-empty lists
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

```
foldr1 f [x_1, x_2, x_3] = x_1 `f` (x_2 `f` x_3)
```

```
-- application: maximum of list elements
```

```
maximum = foldr1 max
```

```
-- foldl, apply function from left-to-right
```

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

```
foldl f e [x_1, x_2, x_3] = ((e `f` x_1) `f` x_2) `f` x_3
```

```
-- application: reverse
```

```
reverse = foldl (flip (:)) []
```

Summary

- higher-order functions
 - functions may have functions as input
 - functions may have functions as output
- partial application
 - n -ary function is value
 - applying n -ary function on 1 argument results in $n - 1$ -ary function
 - sections are special syntax for partially applied operators
- λ -abstraction is anonymous function
- process lists that encode a collection via `map`, `filter`, ...
- `foldr` captures structural recursion on lists, very expressive



Functional Programming

Week 8 – List Comprehension, Calendar Application

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- partial application: if `f` has type `a -> b -> c -> d`, then build expressions

`f :: a -> b -> c -> d`

`f expr :: b -> c -> d`

`f expr expr :: c -> d`

- sections: `(x >)` and `(> x)`

- λ -abstractions: `\ pat -> expr`

- higher-order functions

- functions are values

- functions can take functions as input or return functions as output

- example higher-order functions

`(.) :: (b -> c) -> (a -> b) -> (a -> c)`

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

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

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

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

More Library Functions

Take, Drop, Take-While, Drop-While

- `take :: Int -> [a] -> [a]` and `drop :: Int -> [a] -> [a]`
 - `take n xs` takes the leftmost `n` elements of `xs`
 - `drop n xs` drops the leftmost `n` elements of `xs`
 - if `n >= length xs` then `take n xs = xs` and `drop n xs = []`
 - examples
 - `take 3 "hello" = "hel"` • `drop 2 "hello" = "llo"`
 - `take 4 [1,2] = [1,2]`
 - identity: `take n xs ++ drop n xs = xs`
- `takeWhile :: (a -> Bool) -> [a] -> [a]` and
`dropWhile :: (a -> Bool) -> [a] -> [a]`
 - `takeWhile p xs` takes elements from left of `xs` while `p` is satisfied
 - `dropWhile p xs` drops elements from left of `xs` while `p` is satisfied
 - identity: `takeWhile p xs ++ dropWhile p xs = xs`
- combinations – more efficient versions of the following definitions
 - `splitAt n xs = (take n xs, drop n xs)`
 - `span p xs = (takeWhile p xs, dropWhile p xs)`

Example Application: Separate Words

- task: write function `words :: String -> [String]` that splits a string into words
- example: `words "I am fine. " = ["I", "am", "fine.]`
- implementation:

```
words s = case dropWhile (== ' ') s of
    "" -> []
    s1 -> let (w, s2) = span (/= ' ') s1
           in w : words s2
```

- notes
 - non-trivial recursion on lists
 - `words` is already predefined
 - `unwords :: [String] -> String` is inverse which inserts blanks
 - similar functions to split at linebreaks or to insert linebreaks
 - `lines :: String -> [String]`
 - `unlines :: [String] -> String`
 - identities
 - `words (unwords ss) = ss`, if the strings in `ss` contain no blanks
 - `lines (unlines ss) = ss`, if the strings in `ss` contain no newlines

Combining Two Lists

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
 $\text{zipWith } f \ [x_1, \dots, x_m] \ [y_1, \dots, y_n] = [x_1 `f` y_1, \dots, x_{\min\{m,n\}} `f` y_{\min\{m,n\}}]$
 - resulting list has length of shorter input
 - above equality is not Haskell code, think about recursive definition yourself
- specialization `zip`
-- `(,)` :: `a` -> `b` -> `(a, b)` is the pair constructor
`zip :: [a] -> [b] -> [(a, b)]`
`zip = zipWith (,)`
- inverse function: `unzip :: [(a, b)] -> ([a], [b])`
- examples
 - `zip [1, 2, 3] ['a', 'b'] = [(1, 'a'), (2, 'b')]`
 - `zipWith (*) [1, 2] [3, 4, 5] = [1*3, 2*4] = [3, 8]`
 - `zipWith drop [1, 0] ["ab", "cde"]`
 $= [\text{drop 1 "ab"}, \text{drop 0 "cde"}]$
 $= ["b", "cde"]$
 - `unzip [(1, 'c'), (2, 'b'), (3, 'a')] = ([1, 2, 3], "cba")`

Application: Testing whether a List is Sorted

```
isSorted :: Ord a => [a] -> Bool  
isSorted xs = all id $ zipWith (≤) xs (tail xs)
```

- `id :: a -> a` is the identify function `id x = x`;
used as “predicate” whether a Boolean is `True`
- `($)` is application operator with low precedence, `f $ x = f x`,
used to avoid parentheses
- example:

```
isSorted [1,2,5,3]  
= all id $ zipWith (≤) [1,2,5,3] [2,5,3]  
= all id [1 ≤ 2, 2 ≤ 5, 5 ≤ 3]  
= all id [True, True, False]  
= id True && id True && id False && True  
= False
```

Table of Precedences

precedence	operators	associativity
9	!! , .	left (!!), right (.)
8	^, ^^, **	right
7	* , / , `div`	left
6	+ , -	left
5	: , ++	right
4	== , /= , < , <= , > , >=	none
3	&&	right
2		right
1	>> , >>=	left
0	\$	right

- reminder: associativity determines parentheses between operators of same precedence
 $x : y ++ z = x : (y ++ z)$ $x - y + z = (x - y) + z$
- all of ^, ^^, ** are for exponentiation: difference is range of exponents
- operators (>>) and (>>=) will be explained later

List Comprehension

List Comprehension

- list comprehension is similar to set comprehension in mathematics
- concise, readable definition
 - sum of even squares up to 100: $\sum\{x^2 \mid x \in \{0, \dots, 100\}, \text{even}(x)\}$
- examples of list comprehension in Haskell

```
evenSquares100 = sum [ x^2 | x <- [0 .. 100], even x]
```

```
prime n = n >= 2 && null [ x | x <- [2 .. n - 1], n `mod` x == 0]
```

```
pairs n = [ (i, j) | i <- [0..n], even i, j <- [0..i]]
```

```
> pairs 5
```

```
[(0,0),(2,0),(2,1),(2,2),(4,0),(4,1),(4,2),(4,3),(4,4)]
```

List Comprehension – Structure

```
foo zs = [ x + y + z |  
          x <- [0..20],  
          even x,  
          let y = x * x,  
              y < 200,  
          Just z <- zs]
```

- list comprehension is of form $[e \mid Q]$ where
 - e is Haskell expression, e.g., $x + y + z$
 - Q is the **qualifier**, a possibly empty comma-separated sequence of
 - **generators** of form $pat \leftarrow expr$ where the expression has a list type, e.g., $x <- [0..20]$ or $Just z <- zs$;
 - e and later parts of qualifier may use variables of pat
 - **guards**, i.e., Boolean expressions, e.g., $even x$ or $y < 200$
 - **local declarations** of form $let decls$ (no $in!$);
 e and later parts of qualifier may use variables and functions introduced in $decls$
- if Q is empty, we just write $[e]$

List Comprehension – Translation

```
[ x + y | x <- [0..20], even x, let y = x * x, y < 200]
```

- list comprehension is of form $[e \mid Q]$ where qualifier is list of guards, generators and local definitions
- list comprehension is syntactic sugar, it is translated using the predefined function

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

```
concatMap f = concat . map f
```

- guards:

```
[e | b, Q] = if b then [e | Q] else []
```

- local declaration:

```
[e | let decls, Q] = let decls in [e | Q]
```

- generators for exhaustive patterns (e.g., variable or pair of variables):

```
[e | pat <- xs, Q] = concatMap (\ pat -> [e | Q]) xs
```

- generator (general case):

```
[e | pat <- xs, Q] = concatMap  
  (\ x -> case x of { pat -> [e | Q]; _ -> [] } )  
  xs          -- where x must be a fresh variable name
```

List Comprehension – Translation Examples

- translations

`[e | b, Q] = if b then [e | Q] else []`

`[e | let decls, Q] = let decls in [e | Q]`

`[e | pat <- xs, Q] = concatMap (\ pat -> [e | Q]) xs`

- examples

`[s | (s, g) <- xs, g == 1]`

`= concatMap (\ (s, g) -> [s | g == 1]) xs`

`= concatMap (\ (s, g) -> if g == 1 then [s] else []) xs`

`[y + z | x <- xs, let y = x * x, z <- [0 .. y]]`

`= concatMap (\ x -> [y + z | let y = x * x, z <- [0 .. y]]) xs`

`= concatMap (\ x -> let y = x * x in [y + z | z <- [0 .. y]]) xs`

`= concatMap (\ x -> let y = x * x in`

`concatMap (\ z -> [y + z]) [0 .. y]) xs`

List Comprehension – Order of Generators

- consider `[(i, j) | i <- [0..2], j <- [0..1]]`
- possible outcomes
 - `[(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]`
 - `[(0,0),(1,0),(2,0),(0,1),(1,1),(2,1)]`

outer counter is `i`

outer counter is `j`

- translation reveals correct result

```
[ (i, j) | i <- [0..2], j <- [0..1] ]
= concatMap (\ i -> [(i, j)] | j <- [0..1] ]) [0 .. 2]
= [(0, j)] | j <- [0..1]] ++
  [(1, j)] | j <- [0..1]] ++
  [(2, j)] | j <- [0..1]] ++
= concatMap (\ j -> [(0, j)]) [0..1] ++
  concatMap (\ j -> [(1, j)]) [0..1] ++
  concatMap (\ j -> [(2, j)]) [0..1] ++
=([(0,0)] ++
  [(0,1)] ++
  []) ++
  ([ (1,0)] ++
  [(1,1)] ++
  []) ++
  ([ (2,0)] ++
  [(2,1)] ++
  [])) ++
= [(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]
```

Example Application – Pythagorean Triples

- (x, y, z) is Pythagorean triple iff $x^2 + y^2 = z^2$
- task: find all Pythagorean triples within given range

```
ptriple x y z = x^2 + y^2 == z^2
ptriples n = [ (x,y,z) |
  x <- [1..n], y <- [1..n], z <- [1..n], ptriple x y z]
```

- problem of duplicates because of symmetries

```
> ptriples 5
[(3,4,5),(4,3,5)]
```

- solution eliminates symmetries, also more efficient

```
ptriples n = [ (x,y,z) |
  x <- [1..n], y <- [x..n], z <- [y..n], ptriple x y z]
```

```
> ptriples 5
[(3,4,5)]
```

Application – Printing a Calendar

Printing a Calendar

- given a month and a year, print the corresponding calendar
- example: December 2021

Mo	Tu	We	Th	Fr	Sa	Su
					1	2
					3	4
					5	
6	7	8	9	10	11	12
...						

- decomposition identifies two parts
 - construction phase (computation of days, leap year, ...)
 - layout and printing
- we concentrate on printing, assuming machinery for construction

```
type Month = Int
```

```
type Year = Int
```

```
type Dayname = Int -- Mo = 0, Tu = 1, ..., So = 6
```

-- monthInfo returns name of 1st day in m. and number of days in m.

```
monthInfo :: Month -> Year -> (Dayname, Int)
```

The Picture Analogan

pictures:

- atomic part: **pixel**
- **height** and **width**
- **white** pixel

strings:

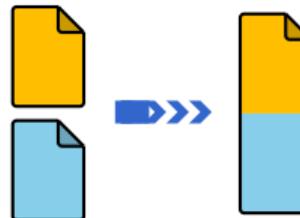
- atomic part: **character**
- number of **rows** and **columns**
- **blank** character

Auxiliary Types

```
type Height  = Int
type Width   = Int
type Picture = (Height, Width, [[Char]])
```

- consider (**h**, **w**, **rs**)
- **rs** :: [[**Char**]] – “list of rows”
- invariant 1: length of **rs** is height **h**
- invariant 2: all rows (that is, lists in **rs**) have length **w**

Stacking Pictures Above Each Other



Stacking Two Picture Above Each Other

```
above :: Picture -> Picture -> Picture
(h, w, css) `above` (h', w', css')
| w == w'      = (h + h', w, css ++ css')
| otherwise     = error "above: different widths"
```

Stacking Several Pictures Above Each Other

```
stack :: [Picture] -> Picture
stack = foldr1 above
```

Spreading Pictures Beside Each Other



Spreading Two Pictures Beside Each Other

```
beside :: Picture -> Picture -> Picture
(h, w, css) `beside` (h', w', css')
| h == h'    = (h, w + w', zipWith (++) css css')
| otherwise = error "beside: different heights"
```

Spreading Several Pictures Beside Each Other

```
spread :: [Picture] -> Picture
spread = foldr1 beside
```

Tiling Several Pictures

```
tile :: [[Picture]] -> Picture
tile = stack . map spread
```

Creating Pictures

- single ‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

- rows

```
row :: String -> Picture
row r = (1, length r, [r])
```

- blank

```
blank :: Height -> Width -> Picture
blank h w = (h, w, blanks)
where
    blanks = replicate h (replicate w ' ')
```

Constructing a Month

- as indicated, assume function

```
monthInfo :: Month -> Year -> (Dayname, Int)
```

where daynames are 0 (Monday), 1 (Tuesday), ...

```
daysOfMonth :: Month -> Year -> [Picture]
```

```
daysOfMonth m y =
```

```
  map (row . rjustify 3 . pic) [1 - d .. numSlots - d]
```

```
where
```

```
(d, t) = monthInfo m y
```

```
numSlots = 6 * 7 -- max 6 weeks * 7 days per week
```

```
pic n = if 1 <= n && n <= t then show n else ""
```

```
rjustify :: Int -> String -> String
```

```
rjustify n xs
```

```
| l <= n = replicate (n - l) ' ' ++ xs
```

```
| otherwise = error ("text (" ++ xs ++ ") too long")
```

```
where l = length xs
```

Tiling the Days

- `daysOfMonth` delivers list of 42 single pictures (of size 1×3)
- missing: layout + header for final picture (of size 7×21)

```
month :: Month -> Year -> Picture
```

```
month m y = above weekdays . tile . groupsOfSize 7 $ daysOfMonth m y
  where weekdays = row " Mo Tu We Th Fr Sa Su"
```

```
-- groupsOfSize splits list into sublists of given length
```

```
groupsOfSize :: Int -> [a] -> [[a]]
```

```
groupsOfSize n [] = []
```

```
groupsOfSize n xs = ys : groupsOfSize n zs
```

```
  where (ys, zs) = splitAt n xs
```

Printing a Month

- transform a Picture into a String

```
showPic :: Picture -> String  
showPic (_, _, css) = unlines css
```

- show result of month m y as String

```
showMonth :: Month -> Year -> String  
showMonth m y = showPic $ month m y
```

- display final string via putStr :: String -> IO () to properly print newlines and drop double quotes

```
> showMonth 12 2021
```

```
" Mo Tu We Th Fr Sa Su\n          1 2 3 4 5\n      6 ..."
```

```
> putStr $ showMonth 12 2021
```

```
Mo Tu We Th Fr Sa Su
```

```
    1 2 3 4 5
```

```
 6 7 8 9 10 11 12
```

```
13 14 15 16 17 18 19
```

```
20 21 22 23 24 25 26
```

```
27 28 29 30 31
```

Summary

- further useful functions on lists

<code>take, drop, splitAt,</code>	-- split at position
<code>takeWhile, dropWhile, span,</code>	-- split via predicate
<code>zipWith, zip, unzip,</code>	-- (un)zip two lists
<code>(\$),</code>	-- application operator
<code>concatMap</code>	-- map with concat combined

- table of operator precedences
- list comprehension

- concise description of lists, similar to set comprehension in mathematics
- can automatically be translated into standard expressions based on `concatMap`
- example:

$$[(x, y, z) \mid x \leftarrow [1..n], y \leftarrow [x..n], z \leftarrow [y..n], x^2 + y^2 == z^2]$$

- calendar application



Functional Programming

Week 9 – Calendar Application, Scope, Modules

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture – Library Functions

`take, drop :: Int -> [a] -> [a]`

`splitAt :: Int -> [a] -> ([a], [a])`

`takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]`

`span :: (a -> Bool) -> [a] -> ([a], [a])`

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

`zip :: [a] -> [b] -> [(a, b)]`

`unzip :: [(a, b)] -> ([a], [b])`

`words, lines :: String -> [String]`

`unwords, unlines :: [String] -> String`

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

`($) :: (a -> b) -> a -> b`

Last Lecture – List Comprehension

- list comprehension
 - shape: `[(x,y,z) | x <- [1..n], let y = x ^ 2, y > 100, Just z <- f y]`
 - consists of guards, generators, local declarations
 - translated via `concatMap`
- examples

```
prime n = n >= 2 && null [ x | x <- [2 .. n - 1], n `mod` x == 0]
```

```
ptriples n = [ (x,y,z) |
  x <- [1..n], y <- [x..n], z <- [y..n], x^2 + y^2 == z^2]
```

Last Lecture – Printing a Calendar

- given a month and a year, print the corresponding calendar
- example: December 2021

Mo	Tu	We	Th	Fr	Sa	Su
					1	2
				3	4	5
6	7	8	9	10	11	12
...						

- we concentrate on printing, assuming machinery for construction

```
type Month    = Int
```

```
type Year     = Int
```

```
type Dayname  = Int -- Mo = 0, Tu = 1, ..., So = 6
```

```
-- monthInfo returns name of 1st day in m. and number of days in m.
```

```
monthInfo :: Month -> Year -> (Dayname, Int)
```

Design and Functionality for Representing Character-Pictures

```
type Height = Int
type Width = Int
type Picture = (Height, Width, [[Char]])
```

```
above :: Picture -> Picture -> Picture
stack :: [Picture] -> Picture
```

```
beside :: Picture -> Picture -> Picture
spread :: [Picture] -> Picture
```

```
tile :: [[Picture]] -> Picture
tile = stack . map spread
```

Finalizing the Calendar

Creating Pictures

- single ‘pixels’

```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```

- rows

```
row :: String -> Picture
row r = (1, length r, [r])
```

- blank

```
blank :: Height -> Width -> Picture
blank h w = (h, w, blanks)
where
    blanks = replicate h (replicate w ' ')
```

Constructing a Month

- as indicated, assume function

```
monthInfo :: Month -> Year -> (Dayname, Int)
```

where daynames are 0 (Monday), 1 (Tuesday), ...

```
daysOfMonth :: Month -> Year -> [Picture]
```

```
daysOfMonth m y =
```

```
  map (row . rjustify 3 . pic) [1 - d .. numSlots - d]
```

```
where
```

```
(d, t) = monthInfo m y
```

```
numSlots = 6 * 7 -- max 6 weeks * 7 days per week
```

```
pic n = if 1 <= n && n <= t then show n else ""
```

```
rjustify :: Int -> String -> String
```

```
rjustify n xs
```

```
| l <= n = replicate (n - l) ' ' ++ xs
```

```
| otherwise = error ("text (" ++ xs ++ ") too long")
```

```
where l = length xs
```

Tiling the Days

- `daysOfMonth` delivers list of 42 single pictures (of size 1×3)
- missing: layout + header for final picture (of size 7×21)

```
month :: Month -> Year -> Picture
```

```
month m y = above weekdays . tile . groupsOfSize 7 $ daysOfMonth m y
  where weekdays = row " Mo Tu We Th Fr Sa Su"
```

```
-- groupsOfSize splits list into sublists of given length
```

```
groupsOfSize :: Int -> [a] -> [[a]]
```

```
groupsOfSize n [] = []
```

```
groupsOfSize n xs = ys : groupsOfSize n zs
```

```
  where (ys, zs) = splitAt n xs
```

Printing a Month

- transform a Picture into a String

```
showPic :: Picture -> String  
showPic (_, _, css) = unlines css
```

- show result of month m y as String

```
showMonth :: Month -> Year -> String  
showMonth m y = showPic $ month m y
```

- display final string via putStr :: String -> IO () to properly print newlines and drop double quotes

```
> showMonth 12 2021
```

```
" Mo Tu We Th Fr Sa Su\n          1 2 3 4 5\n      6 ..."
```

```
> putStr $ showMonth 12 2021
```

```
Mo Tu We Th Fr Sa Su  
      1 2 3 4 5  
      6 7 8 9 10 11 12  
    13 14 15 16 17 18 19  
    20 21 22 23 24 25 26  
    27 28 29 30 31
```

Scope

Scope

- consider program (1 compile error)

```
radius = 15
```

```
area radius = pi^2 * radius
```

```
squares x = [ x^2 | x <- [0 .. x]]
```

```
length [] = 0
```

```
length (_ : xs) = 1 + length xs
```

```
data Rat = Rat Integer Integer
```

```
createRat n d = normalize $ Rat n d where normalize ... = ...
```

- scope

- resolve ambiguities
- defines which names of variables, functions, types, ... are visible at a given program position
- controlling scope to structure larger programs (imports / exports)

Scope of Names

```
radius = 15
```

```
area radius = pi^2 * radius
```

- in the following we assume that `name_i` in the real code is always just `name` and the `_i` is used for addressing the different occurrences of `name`
- renamed Haskell program

```
radius_1 = 15
```

```
area_1 radius_2 = pi_1^2 * radius_3
```

- scope of names in right-hand sides of equations
 - is `radius_3` referring to `radius_2` or `radius_1`?
 - what is `pi_1` referring to?
- rule of thumb for searching `name`: search **inside-out**
 - think of abstract syntax tree of expression
 - whenever you pass a `let`, `where`, `case`, or function definition where `name` is **bound**, then refer to that **local** name
 - if nothing is found, then search **global** function `name`, also in Prelude
- `radius_3` refers to `radius_2`, `pi_1` to `Prelude.pi`

Local Names in Case-Expressions

- general case: `case expr of { pat1 -> expr1; ...; patN -> exprN }`
 - each `patI` binds the variables that occur in `patI`
 - these variables can be used in `exprI`
 - the newly bound variables of `patI` bind stronger than any previously bound variables
- example Haskell expression

```
case xs_1 of                      -- renamed Haskell expression
  [] -> xs_2
  (x_1 : xs_3) -> case xs_4 ++ ys_1 of
    [] -> ys_2
    (x_2 : xs_5) -> x_3 : xs_6 ++ ys_3
```

- `x_3` refers to `x_2` (since `x_2` is further inside than `x_1`)
- `xs_6` refers to `xs_5` (since `xs_5` is further inside than `xs_3`)
- `xs_4` refers to `xs_3`
- `xs_1`, `xs_2`, `ys_1`, `ys_2`, and `ys_3` are not bound in this expression
(the proper references need to be determined further outside)

Local Names in Let-Expressions

```
let {  
    pat1 = expr1; ...; patN = exprN;  
    f1 pats1 = fexpr1; ...; fM patsM = fexprM  
} in expr
```

- all variables in `pat1` ... `patN` and all names `f1` ... `fM` are bound
 - these can be used in `expr`, in each `exprI` and in each `fexprJ`
 - variables of `patsJ` bind strongest, but only in `fexprJ`
- `let (x_1, y_1) = (y_2 + 1, 5) -- renamed Haskell expression`
- ```
 f_1 x_2 = x_3 + g_1 y_3 id_1
 g_2 y_4 f_2 = f_3 $ g_3 x_4 f_4
```
- `in (f_5, g_4, x_5, y_5)`
- `y_2`, `y_3` and `y_5` refer to `y_1`
  - `x_3` refers to `x_2` since `x_2` binds stronger than `x_1`
  - `x_4` and `x_5` refer to `x_1`
  - `f_3` and `f_4` refer to `f_2` since `f_2` binds stronger than `f_1`
  - `g_1`, `g_3` and `g_4` refer to `g_2`
  - `f_5` refers to `f_1`
  - `id_1` is not bound in this expression

# Global Function Definitions

- general case:

`fname pats = expr`

- all variables in `pats` are bound locally and can be used in `expr`
- `fname` is **not** locally bound, but added to global lookup table
- all variables/names in `expr` without local reference will be looked up in global lookup table
- lookup in global table does not permit ambiguities

- `radius_1 = 15` -- renamed Haskell program

`area_2 radius_2 = pi_1^2 * radius_3`

`length_1 [] = 0`

`length_2 (_:xs_1) = 1 + length_3 xs_2`

- `radius_1`, `area_2` and `length_1/2` are stored in global lookup table
- global lookup table has ambiguity: `length_1/2` vs. `Prelude.length`
- `pi_1` is not locally bound and therefore refers to `Prelude.pi`
- `radius_3` refers to local `radius_2` and not to global `radius_1`
- `xs_2` refers to `xs_1`
- `length_3` is not locally bound and because of mentioned ambiguity, this leads to a compile error

## Global vs. Local Definitions

```
length :: [a] -> Int
-- choose definition 1,
length = foldr (\ _-> (1 +)) 0
-- definition 2,
length =
 let { length [] = 0; length (x : xs) = 1 + length xs }
 in length
-- or definition 3
length [] = 0
length (_ : xs) = 1 + length xs
```

- definitions 1 and 2 compile since there is no `length` in the rhs that needs a global lookup
- in contrast, definition 3 does not compile
- still definitions 1 and 2 result in ambiguities in global lookup table  
→ study Haskell's module system

# Modules

# Modules

- so far

- Haskell program is a **single** file, consisting of several definitions
- all global definitions are visible to user

```
-- functions on rational numbers
```

```
data Rat = Rat Integer Integer -- internal definition of datatype
```

```
normalize (Rat n d) = ... -- internal function
```

```
createRat n d = normalize $ Rat n d -- function for external usage
```

```
...
```

```
-- application: approximate pi to a certain precision
```

```
piApprox :: Integer -> Rat
```

```
piApprox p = ...
```

- motivation for modules

- structure programs into smaller **reusable** parts without copying
- distinguish between **internal and external** definitions
  - clear interface for users of modules
  - maintain invariants
  - improve maintainability

## Modules in Haskell

```
-- first line of file ModuleName.hs
module ModuleName(exportList) where
-- standard Haskell type and function definitions
```

- each **ModuleName** has to start with uppercase letter
- each module is usually stored in separate file **ModuleName.hs**
- if Haskell file contains no **module** declaration, ghci inserts module name **Main**
- **exportList** is comma-separated list of function-names and type-names, these functions and types will be accessible for users of the module
- if (**exportList**) is omitted, then everything is exported
- for types there are different export possibilities
  - **module** Name(**Type**) exports **Type**, but no constructors of **Type**
  - **module** Name(**Type**(...)) exports **Type** and its constructors

## Example: Rational Numbers

```
module Rat(Rat, createRat, numerator, denominator) where
data Rat = Rat Integer Integer
normalize = ...
createRat n d = normalize $ Rat n d
numerator (Rat n d) = n
...
instance Num Rat where ...
instance Show Rat where ...
```

- external users know that a type `Rat` exists
- they only see functions `createRat`, `numerator` and `denominator`
- they don't have access to constructor `Rat` and therefore cannot form expressions like `Rat 2 4` which break invariant of cancelled fractions
- they can perform calculations with rational numbers since they have access to `(+)` of class `Num`, etc., in particular for the instance `Rat`
- for the same reason, they can display rational numbers via `show`

## Example: Application

```
module PiApprox(piApprox, Rat) where
-- Prelude is implicitly imported
-- import everything that is exported by module Rat
import Rat
-- or only import certain parts
import Rat(Rat, createRat)
-- import declarations must be before other definitions
piApprox :: Integer -> Rat
piApprox n = let initApprox = createRat 314 100 in ...
```

- there can be multiple `import` declarations
- what is imported is not automatically exported
  - when importing `PiApprox`, type `Rat` is visible, but `createRat` is not
  - if application requires both `Rat` and `PiApprox`, import both modules:  
`import PiApprox`  
`import Rat`

# Resolving Ambiguities

```
-- Foo.hs
module Foo where pi = 3.1415
```

```
-- Problem.hs
module Problem where

import Foo

pi = 3.1415
area r = pi * r^2
```

- problem: what is `pi` in definition of `area`? (global name)
- lookup map is ambiguous: `pi` defined in `Prelude`, `Foo`, and `Problem`
- ambiguity persists, even if definition is identical
- solution via `qualifier`: disambiguate by using `ModuleName.name` instead of `name`
  - write `area r = Problem.pi * r^2` in `Problem.hs`  
(or `area r = Prelude.pi * r^2`)

## Qualified Imports

```
module Foo where pi = 3.1415
module SomeLongModuleName where fun x = x + x

module ExampleQualifiedImports where

-- all imports of Foo have to use qualifier
import qualified Foo
-- result: no ambiguity on unqualified "pi"

import qualified SomeLongModuleName as S
-- "as"-syntax changes name of qualifier

area r = pi * r^2
myfun x = S.fun (x * x)
```

# Summary

## Summary

- calendar application
- scoping rules determine visibility of function names and variable names
- larger programs can be structured in **modules**
  - explicit **export-lists** to distinguish internal and external parts
  - advantage: changes of internal parts of module **M** are possible without having to change code that imports **M**, as long as exported functions of **M** have same names and types
  - if no module name is given: **Main** is used as module name
  - further information on modules  
<https://www.haskell.org/onlinereport/modules.html>



# Functional Programming

Week 10 – Input and Output, Connect Four

René Thiemann    Philipp Anrain    Marc Bußjäger    Benedikt Dornauer    Manuel Eberl  
Christina Kohl    Sandra Reitinger    Christian Sternagel

Department of Computer Science

# Last Lecture

- scoping rules determine visibility of function names and variable names
  - larger programs should be structured in modules
    - explicit export-lists to distinguish internal and external parts
    - import of modules instead of copying code
    - qualified imports and qualifiers are useful for resolving name conflicts
    - defaults
      - if program does not contain module declaration, `module Main where` is added
      - `import Prelude` is implicitly added, if no other imports of `Prelude` are present
  - example

# Input and Output in Haskell

# I/O: Input and Output

- aim: communicate with the user
  - ask user for inputs
  - print answers
  - **outside** the GHCI read-eval-print-loop
  - stand-alone programs that neither require ghc-installation nor Haskell knowledge of user
- I/O is not restricted to text-based user-I/O
  - reading and writing of files  
(e.g., compiler translates .hs to .exe, or .tex to .pdf)
  - reading and writing into memory  
(mutable state, arrays)
  - reading and writing of network channels  
(e.g., web-server and internet-browser)
  - start other programs and communicate with them
  - play/record sound, capture mouse-movements, ...

## An Initial Example

- ```
main = do                      -- file: welcomeIO.hs
    putStrLn "Greetings! Please tell me your name."
    name <- getLine
    putStrLn $ "Welcome to Haskell's IO, " ++ name ++ "!"
```
- compile it with GHC (not GHCI) via

```
$ ghc --make welcomeIO.hs
```
- and run it

```
$ ./welcomeIO                  # welcomeIO.exe on Windows
Greetings! Please tell me your name.
Homer                         # this was typed in
Welcome to Haskell's IO, Homer!
```
- notes
 - `putStrLn` – prints string followed by newline
 - `getLine` – reads line from standard input
 - new syntax: `do` and `<-`

I/O and the Type System

- consider

```
ghci> :l welcomeIO.hs
```

```
ghci> :t putStrLn
```

```
putStrLn :: String -> IO ()
```

```
ghci> :t getLine
```

```
getLine :: IO String
```

```
ghci> :t main
```

```
main :: IO ()
```

- **IO a** is type of I/O actions delivering results of type **a**
(in addition to their I/O operations)
- examples

- **String -> IO ()** – after supplying a string, we obtain an I/O action

(in case of **putStrLn**, “printing”)

- **IO ()** – just perform I/O

(in case of **main**, run our program)

- **IO String** – do some I/O and deliver a string

(in case of **getLine**, user-input)

Combining I/O Actions

- I/O actions can be combined
- core building block: **bind** (syntax $\gg=$)
 $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- consider $act1 \gg= \lambda\ x \rightarrow act2$
 - on evaluation, this expression first performs action $act1$
 - the result of action $act1$ is stored in x
 - afterwards action $act2$ is performed (which may depend on x)
 - in total, both actions are performed and the result is that of $act2$
- ignoring results: $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b, a1 \gg a2 = a1 \gg= \lambda\ _ \rightarrow a2$
- example

```
putStrLn "Hi. What's your name?" >>      -- ignore result, which is ()  
getLine >>= \ name ->                      -- store result in variable name  
let answer = "Hello " ++ name in              -- no I/O in this line  
putStrLn answer                                -- final result from putStrLn: ()
```

- the type of overall expression is $IO\ ()$, that of the last I/O action `putStrLn answer`
- execution of actions is sequential, like in imperative programming

Do-Notation

- there is special syntax for combinations of binds, lambdas and lets

```
do x <- act          =      act >>= \ x -> do block  
    block
```

```
do act               =      act >> do block  
    block
```

```
do let x = e         =      let x = e in do block  
    block
```

- putStrLn "Hi. What's your name?" >>

```
getLine >>= \ name ->  
let answer = "Hello " ++ name in  
putStrLn answer
```

can be written as

```
do putStrLn "Hi. What's your name?"  
    name <- getLine  
    let answer = "Hello " ++ name           -- no "in"!  
    putStrLn answer
```

- as in let-syntax, do-blocks can also written via do { .. ; .. ; .. }

Further Notes

- inside do-block, order is important; I/O actions are executed in order of appearance; result of block is result of **last** action
- **x <- a** is not available outside I/O actions,
in particular there is no function of type **IO a -> a** which extracts the results of an action (of type **IO a**) without being an action itself (result type **a**)
 - once we are inside an IO action, we cannot escape
 - **strict separation between purely functional code and I/O**
 - when **IO a** does not appear inside type signature, we can be absolutely sure that no I/O (“side-effect”) is performed
- **main :: IO ()** is the I/O action that is executed when running a compiled file via
`ghc --make prog.hs` and then `./prog`
(`prog.hs` must contain a module **Main** that exports **main**)

Using Purely Functional Code Inside I/O Actions

```
-- reply is purely functional: no IO in type
reply :: String -> String
reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ n ++ " characters."
    where n = show $ length name
-- pure code can be invoked from I/O-part
```

```
main :: IO ()
main = do
    putStrLn "Greetings again. What's your name?"
    name <- getLine
    let niceReply = reply name
    putStrLn niceReply
```

- invoking purely functional code inside I/O is easy
- the other direction is not possible

Some Predefined I/O Functions

- `return :: a -> IO a` – turn anything into an I/O action which does nothing
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (no newline-character in result)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

Recursive I/O Actions

- branching and recursion is also possible with I/O actions
- example: implement `getLine` via `getChar`

```
import Prelude hiding (getLine)

getLine = do
    c <- getChar
    if c == '\n'                      -- branching
        then return ""
    else do
        l <- getLine                  -- recursion
        return $ c : l
```

Examples – Imitating Some GNU Commands

- cat.hs – print file contents

```
import System.Environment (getArgs)
main = do
    [file] <- getArgs      -- assume there is exactly one file
    s <- readFile file
    putStrLn s
```

- wc.hs – count number of lines/words/characters in input

```
count s = nl ++ " " ++ nw ++ " " ++ nc ++ "\n"
where nl = show $ length $ lines s
      nw = show $ length $ words s
      nc = show $ length s
main = interact count
```

- sort.hs – sort input lines

```
import Data.List (sort)
main = interact (unlines . sort . lines)
```

Laziness and I/O Actions

- consider a simple copying program

```
main = do           -- imports omitted
    [src, dest] <- getArgs
    s <- readFile src
    writeFile dest s
```

- `readFile` and `writeFile` are **lazy**, e.g., `readFile` only reads characters on demand
- positive effect: large files can be copied without fully loading them into memory

- laziness might lead to problems

```
main = do           -- imports omitted
    [file] <- getArgs
    s <- readFile file
    writeFile file (map toUpper s)
```

- since `readFile` is lazy, when executing `s <- readFile file` nothing is read immediately
- but then the **same** file should be opened for writing; conflict, which will result in error
- solution: more fine-grained control via file-**handles** which explicitly open and close files, see lecture Operating Systems

Higher-Order on I/O Actions

- `foreach :: [a] -> (a -> IO b) -> IO ()`
`foreach [] io = return ()`
`foreach (a:as) io = do { io a; foreach as io }`
- better `cat.hs`

```
main = do
  files <- getArgs
  if null files then interact id else do
    foreach files readAndPrint
    where readAndPrint file = do
      s <- readFile file
      putStrLn s
```

Monads

- bind and do-notation are **not** fixed to I/O
- there exists a more general concept of **monads**
- example: also the **Maybe**-type is a monad

```
data Expr = Const Double | Div Expr Expr
eval :: Expr -> Maybe Double
eval (Const c) = return c
eval (Div expr1 expr2) = do
    x1 <- eval expr1
    x2 <- eval expr2
    if x2 == 0
        then Nothing
        else return (x1 / x2)
```

- monads won't be covered here, but they are the reason why the Haskell literature speaks about the I/O-**monad**

Example Application: Connect Four

Connect Four

- aim: implement **Connect Four**, MB Spiele



- with textual user interface

0123456

.....

.XO.X..

.XOOOXO

XOXOXOX

OXXOXOO

XXOXOOX

Player X to go

Choose one of [0,1,2,3,4,5,6]

Connect Four: Implementation

- clear separation between
 - user interface (I/O)
 - ask for a move
 - print the current state
 - ...
 - game logic (purely functional code)
 - type to represent a state (board + next player)
 - perform a move
 - check for a winner
 - display a state as string
 - ...
- both parts would be written as two separate modules
 - Logic contains the game logic
 - Main contains the user interface and the main function

Game Logic: Interface

- types: State, Move and Player
- constant initState :: State
- function showPlayer :: Player -> String
- function showState :: State -> String
- function winningPlayer :: State -> Maybe Player
- function validMoves :: State -> [Move]
- function dropTile :: Move -> State -> State
- in total

```
module Logic(State, Move, Player,  
    initState, showPlayer, showState,  
    winningPlayer, validMoves, dropTile) where  
... -- details, which the user interface doesn't have to know
```

The Read-Class

- class `Read` provides methods to convert `Strings` into other types
 - `read :: Read a => String -> a`
 - `readMaybe :: Read a => String -> Maybe a`
import of module `Text.Read` required
 - when using `read`, often the type `a` has to be chosen explicitly
 - examples
 - `(read "(41, True)" :: (Integer,Bool)) = (41, True)`
 - `(read "(41, True)" :: (Integer,Integer)) = error ...`
 - `(readMaybe "1" :: Maybe Integer) = Just 1`
 - `(readMaybe "one" :: Maybe Integer) = Nothing`
- for the `Logic` module, we assume that the type `Move` is an instance of `Show` and `Read`

User Interface

```
module Main(main) where      -- module name must be "Main" for compilation
import Logic

main = do
    putStrLn "Welcome to Connect Four"
    game initState

game state = do
    putStrLn $ showState state
    case winningPlayer state of
        Just player -> putStrLn $ showPlayer player ++ " wins!"
        Nothing -> let moves = validMoves state in
            if null moves then putStrLn "Game ends in draw."
            else do
                putStrLn $ "Choose one of " ++ show moves ++ ": "
                moveStr <- getLine
                let move = (read moveStr :: Move)
                game (dropTile move state)
```

Game Logic: Encoding a State and Initial State

```
type Tile    = Int    -- 0, 1, or 2
type Player = Int    -- 1 and 2
type Move    = Int    -- column number
data State = State Player [[Tile]]  -- list of rows

empty :: Tile
empty = 0

numRows, numCols :: Int
numRows = 6
numCols = 7

startPlayer :: Player
startPlayer = 1

initState :: State
initState = State startPlayer
  (replicate numRows (replicate numCols empty))
```

Game Logic: Valid Moves and Displaying a State

```
validMoves :: State -> [Move]
validMoves (State _ rows) =
    map fst . filter ((== empty) . snd) . zip [0 .. numCols - 1] $ head rows

showPlayer :: Player -> String
showPlayer 1 = "X"
showPlayer 2 = "O"

showTile :: Tile -> Char
showTile t = if t == empty then '.' else head $ showPlayer t

showState :: State -> String
showState (State player rows) = unlines $
    map (head . show) [0 .. numCols - 1] :
    map (map showTile) rows
    ++ ["\nPlayer " ++ showPlayer player ++ " to go"]
```

Game Logic: Making a Move

```
otherPlayer :: Player -> Player
```

```
otherPlayer = (3 -)
```

```
dropTile :: Move -> State -> State
```

```
dropTile col (State player rows) = State
```

```
(otherPlayer player)
```

```
(reverse $ dropAux $ reverse rows)
```

```
where
```

```
dropAux (row : rows) =
```

```
case splitAt col row of
```

```
(first, t : last) ->
```

```
if t == empty
```

```
then (first ++ player : last) : rows
```

```
else row : dropAux rows
```

Game Logic: Winning Player

```
winningRow :: Player -> [Tile] -> Bool
winningRow player [] = False
winningRow player row = take 4 row == replicate 4 player
    || winningRow player (tail row)

transpose ([] : _) = []
transpose xs = map head xs : transpose (map tail xs)

winningPlayer :: State -> Maybe Player
winningPlayer (State player rows) =
    let prevPlayer = otherPlayer player
        longRows = rows ++ transpose rows           -- ++ diags rows
    in if any (winningRow prevPlayer) longRows
        then Just prevPlayer
        else Nothing
```

Connect Four: Final Remarks

- implementation is quite basic
 - diagonal winning-condition missing
 - crashes when invalid moves are entered
 - no iterated matches
- exercise: improve implementation

Summary

Summary

- in Haskell I/O is possible, `IO a` is type of I/O-actions with result of type `a`
- clear separation between purely functional and I/O-code
- multiple actions can be connected via `(>>=)` or `do`-blocks
- several predefined functions to access I/O
- more information on I/O in Haskell:
<http://book.realworldhaskell.org/read/io.html>
- `Read` class provides method `read :: String -> a`, opposite to `Show`
- connect four: separate implementation of game logic (pure) and user interface (I/O)



Functional Programming

Week 11 – Lazy Evaluation, Infinite Lists

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- `IO a` is type of I/O-actions with resulting type `a`
- `do`-blocks are used for sequential composition of I/O-actions
- clear separation between purely functional and I/O-code:
 - embed functional code into I/O: `return :: a -> IO a`
 - the other direction is not available: no function of type `IO a -> a`
- `ghc` compiles programs that provide `main :: IO ()` function in module `Main`
- example application: connect four
 - user-interface: I/O-code
 - game logic: purely functional

Evaluation Strategies

Pure Functions

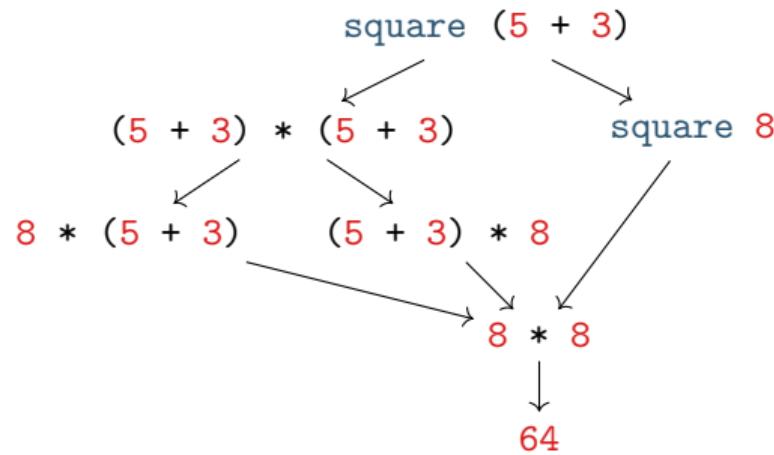
- a function is **pure** if it always returns same result on same input
- pure functions are similar to mathematical functions
- examples of pure functions
 - addition
 - sort a list
 - ...
- examples of non-pure functions
 - roll a dice
 - current time
 - position of cursor
 - ...
- pure languages permit to define only pure functions
- **Haskell is a pure language**

Pure Functions and I/O

- even I/O is pure in Haskell
- consider `main = getLine >>= putStrLn . ("Hello " ++)`
- it seems that the result depends on user input, so is not pure
- however `main :: IO ()`, so the functional value of `main` is not what is entered and printed during execution, but the value is of type `IO ()`, i.e., a sequence of actions that are executed when running the program; and indeed this sequence is always the same:
first read some input i and then print the string "Hello i "
- alternative argumentation: interpret type `IO a` a state transformer on the outside world, e.g., as a function of type `RealWorld -> (RealWorld, a)`
- remark: in the remainder of this lecture we will only consider purely functional programs without I/O

Evaluation Order

- there are several ways to evaluate expressions, consider square $x = x * x$



- in pure languages, the evaluation order has no impact on resulting normal form
- normal form: an expression that cannot be evaluated further, a result

Theorem

Whenever there are two (different) ways to evaluate a Haskell expression to normal form, then the resulting normal forms are identical.

Standard Evaluation Strategies

- each functional language fixes the evaluation order via some evaluation strategy
- three prominent evaluation strategies (expressions represented as **trees** and **dags**)
 - **call-by-value / strict / innermost**: first evaluate arguments

$$\begin{array}{ccccccc} \text{square} & & \text{square} & & * & & \\ | & & | & & / \backslash & & \\ \text{square } (5+3) = & + & 8 & = & 8 & 8 & = 64 \\ / \backslash & & & & / \backslash & & \\ 5 & 3 & & & 8 & 8 & \\ & & & & & & \end{array}$$

- **call-by-name / non-strict / outermost**: directly replace function application by rhs

$$\begin{array}{ccccccc} \text{square} & & * & & * & & * \\ | & & / \backslash & & / \backslash & & / \backslash \\ \text{square } (5+3) = & + & + & + & + & 8 & = 8 8 = 64 \\ / \backslash & 8 & \\ 5 & 3 & 5 & 3 & 5 & 3 & 8 \\ & & & & & & \end{array}$$

- **call-by-need / lazy evaluation**: like call-by-name + sharing (dags = directed acyclic graphs)

$$\begin{array}{ccccccc} \text{square} & & * & & * & & \\ | & & () & & () & & \\ \text{square } (5+3) = & + & + & = & 8 & = 64 \\ / \backslash & / \backslash & / \backslash & & 8 & \\ 5 & 3 & 5 & 3 & 8 & \\ & & & & & \end{array}$$

Evaluation Strategy of Haskell

- Haskell uses lazy evaluation with left-to-right argument order
- sharing is applied whenever a variable occurs multiple times
- example: consider definition $f\ x = g\ x + g\ (3 + 5) + x$
 - when evaluating $f\ (1 + 2) = g\ (1 + 2) + g\ (3 + 5) + (1 + 2)$ the two occurrences of $1 + 2$ are shared: they use the same variable x
 - when evaluating $f\ (3 + 5) = g\ (3 + 5) + g\ (3 + 5) + (3 + 5)$ the two occurrences of $g\ (3 + 5)$ are not shared: it was a coincidence that x was substituted by $3 + 5$ and this equality is not detected at runtime
- there might be further sharing (depending on the compiler), e.g. sharing common subexpressions such as the expression $g\ x$ in a function definition $f\ x = g\ x + h\ (g\ x)$
- argument evaluation within function invocation $f\ \text{expr1} \dots \text{exprN}$ is mainly triggered by pattern matching, i.e., the process of finding the suitable defining equation $f\ \text{pat1} \dots \text{patN} = \text{expr}$, cf. slides 12 and 14 of week 3
- many builtin arithmetic functions will trigger evaluation of all arguments, e.g., $(0 :: \text{Integer}) * \text{undefined}$ will result in error, and not in 0

Evaluation Strategy and Termination

- consider the following Haskell script

```
three :: Integer -> Integer
```

```
three x = 3
```

```
inf :: Integer
```

```
inf = 1 + inf
```

- strict evaluation does not terminate, i.e., it will evaluate forever

```
three inf = three (1 + inf) = three (1 + (1 + inf)) = ...
```

- non-strict and lazy evaluation are immediately done

```
three inf = 3
```

Theorem

- if the evaluation of an expression terminates for some evaluation strategy, then it terminates using non-strict or lazy evaluation
- if the evaluation of an expression terminates using strict evaluation, then it terminates for every evaluation strategy

Comparison of Evaluation Strategies

- call-by-value
 - easy to understand
 - easy to implement
 - overhead in evaluating non-required expressions
 - used in many functional programming languages
- lazy evaluation
 - harder to understand
 - single evaluation step is more complicated to implement:
pass arguments that are unevaluated expressions (thunks) instead of just values
 - overhead in computing with thunks
 - allows programmers to naturally define and work with infinite data
 - used in Haskell

Tail Recursion and Strict Evaluation

Different Kinds of Recursion

- a function calling itself is **recursive**
- functions that mutually call each other are **mutually recursive**

```
even n | n == 0      = True
        | otherwise = odd (n - 1)
odd n  | n == 0      = False
        | otherwise = even (n - 1)
```

- **nested recursion**: recursive calls inside recursive calls

```
ack n m | n == 0 = m + 1
        | m == 0 = ack (n - 1) 1
        | otherwise = ack (n - 1) (ack n (m - 1))
```

- **linear recursion**: at most one recursive call (per if-then-else branch)

- `fib n | n >= 2 = fib (n - 1) + fib (n - 2)` X
 - `length (x : xs) = 1 + length xs` ✓
 - `f x = if even x then f (x `div` 2) else f (3 * x + 1)` ✓
- **tail recursion** and **guarded recursion** will be discussed in more detail

Tail Recursion

- tail recursion is special form of linear recursion
- additional requirement
 - recursive function calls happen at the outermost level
 - however, they can be within an if-then-else
- examples
 - `length (x : xs) = 1 + length xs`
 - `f x = if even x then f (x `div` 2) else f (3 * x + 1)`
- advantage of tail recursion
 - no dangling function calls
 - can be evaluated as loop
 - space efficient

✗

✓

Example: Advantage of Tail Recursion

- linear but not tail recursive variant

```
sumRec 0 = 0
```

```
sumRec n = n + sumRec (n - 1)
```

```
sumRec 5 = 5 + sumRec (5 - 1)
```

```
= 5 + sumRec 4 = 5 + (4 + sumRec (4 - 1))
```

```
= 5 + (4 + sumRec 3) = 5 + (4 + (3 + sumRec (3 - 1))) = ...
```

```
= 5 + (4 + (3 + (2 + (1 + 0)))) = ... = 15 -- linear space
```

- tail recursive variant using **accumulator** to store intermediate results

```
sumTr n = aux 0 n where
```

```
aux acc 0 = acc
```

```
aux acc n = aux (acc + n) (n - 1)
```

```
sumTr 5
```

```
= aux 0 5 = aux (0 + 5) (5 - 1)
```

```
= aux 5 4 = aux (5 + 4) (4 - 1)
```

```
= aux 9 3 = ... = 15
```

-- constant space, implement as loop with two variables: acc and n

Problem of Tail Recursion using Lazy Evaluation

```
sumTr n = aux 0 n where  
  aux acc 0 = acc  
  aux acc n = aux (acc + n) (n - 1)
```

- example evaluation of `sumTr` on previous slide used call-by-value
- in lazy evaluation `acc` and `n` are only evaluated on demand
- causes linear memory consumption in `sumTr`

```
sumTr 5           -- with lazy evaluation  
= aux 0 5  
= aux (0 + 5) (5 - 1)  
= aux (0 + 5) 4  
= aux ((0 + 5) + 4) (4 - 1)  
= ...  
= aux (((((0 + 5) + 4) + 3) + 2) + 1) 0  
= (((0 + 5) + 4) + 3) + 2) + 1 = ... = 15
```

Enforcing Evaluation

- Haskell function to enforce evaluation: `seq :: a -> b -> b`
- evaluation of `seq x y` first evaluates `x` to WHNF and then returns `y`
- **WHNF**: weak head normal form
- expression `e` is in WHNF iff it has one of the following three shapes
 - `e = C expr1 ... exprN` for some constructor `C` (constructor application)
 - `e = f expr1 ... exprN` if the defining equations of `f` have `M > N` arguments, i.e., they are of the form `f pat1 ... patM = expr` (too few arguments)
 - `e = \ pat1 ... patN -> expr` (λ -abstraction)
- examples
 - in WHNF: `True`, `7.1`, `(5+1) : [1] ++ [2]`, `(:)`, `undefined : undefined`, `(++)`,
`(++ undefined)`, `\ x -> undefined`
 - not in WHNF: `[1] ++ [2]`, `(\ x -> x + 1) (1 + 2)`, `undefined ++ undefined`
 - evaluation:
`let x = 1 + 2 in seq x (f x)`
`= seq (1 + 2) (f (1 + 2))` -- with `1 + 2` shared
`= seq 3 (f 3)` -- seq enforced evaluation of argument
`= f 3 = ...` -- evaluation of `f 3` continues

Example Application using seq

- solve memory problem in tail recursion by enforcing evaluation of accumulator

```
sumTrSeq n = aux 0 n where
    aux acc 0 = acc
    aux acc n = let accN = acc + n in seq accN (aux accN (n - 1))

    sumTrSeq 5
= aux 0 5
= let accN = 0 + 5 in seq accN (aux accN (5 - 1))                                -- 0 + 5 is shared
= seq (0 + 5) (aux (0 + 5) (5 - 1))                                              -- and evaluated
= seq 5 (aux 5 (5 - 1))
= aux 5 (5 - 1)
= aux 5 4                                -- pattern matching triggers evaluation
= let accN = 5 + 4 in seq accN (aux accN (4 - 1))                                -- 5 + 4 is shared
= seq (5 + 4) (aux (5 + 4) (4 - 1))                                              -- and evaluated
= seq 9 (aux 9 (4 - 1))
= aux 9 (4 - 1)                           -- same structure as above
= ... = 15                                -- constant space
```

Enforcing Strict Evaluation . . . Continued

- besides `seq`, there are other options to enforce strict evaluation
- **strict library functions** like a strict version of `foldl`:

```
Data.List.foldl' :: (b -> a -> b) -> b -> [a] -> b
```

```
import Data.List
length = foldl' (\ x _ -> x + 1) 0
```

- pattern matching with **bang patterns** to enforce evaluation, e.g.,
`aux acc n = let !accN = acc + n in aux accN (n - 1)`
- **strict datatypes**
- see https://downloads.haskell.org/~ghc/8.10.7/docs/html/users_guide/glasgow_exts.html#bang-patterns-and-strict-haskell for further details

Lazy Evaluation and Infinite Lists

Guarded Recursion

- every recursive call is inside (“guarded by”) a constructor
- also known as “tail recursion modulo cons”
- more important than tail recursion in Haskell
- allows the result to be consumed lazily – tail recursion provides the result only at the end
- examples

- `map f [] = []`

- `map f (x:xs) = f x : map f xs`

- `reverse xs = revAux xs [] where`

- `revAux [] ys = ys`

- `revAux (x : xs) ys = revAux xs (x : ys)`

- `enumFrom x = x : enumFrom (x + 1)`

- remarks on `enumFrom`

- above definition is simplified, actual definition works for members of type class `Enum`, e.g., `Int`, `Char`, `Integer`, `Double`, ... and prevents overflows

- syntactic sugar: `[x..] = enumFrom x`



Infinite Lists

- infinite lists \sim **sequences** of elements (also known as streams)
- programming with infinite lists: producing and consuming elements of sequences one after another (e.g., with guarded recursion)
- example: `[x..] = x : [x + 1 ..]` generates infinite list
- in combination with lazy evaluation, infinite lists do not always cause non-termination
- examples

```
take 2 [7..]
= take 2 (7 : [8..])
= 7 : take 1 [8..]
= 7 : 8 : take 0 [9..]
= [7, 8]
```

```
takeWhile (< 95) $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81]

filter (< 100)    $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81] -- interrupted
```

Laziness and Infinite Data Structures Facilitate Modularity

- separation of concerns
 - write small functions with specific tasks
 - use potentially infinite data structures
- example: find index of first list element satisfying predicate
 - function `firstIndex :: (a -> Bool) -> [a] -> Int`
 - in Haskell

```
firstIndex p = fst . head . filter (p . snd) . zip [0..]
```
 - (lazy) evaluation (without showing expansion of (.) and (\$))

```
firstIndex (== 1) [1..9]
= fst . head . filter ((== 1) . snd) $ zip [0..] [1..9]
= fst . head . filter ((== 1) . snd) $ (0,1) : zip [1..] [2..9]
= fst . head $ (0,1) : filter ((== 1) . snd) $ zip [1..] [2..9]
= fst (0,1)
= 0
```
- without laziness several **complete** list traversals are required when using library functions (e.g., computation of length and addition of indices)
- remark: `firstIndex` works for arbitrary lists as input: finite and infinite

Sieve of Eratosthenes

- goal: generate list of **all** prime numbers
- algorithm
 1. start with list of all natural numbers (from 2 on)
 2. mark first element x as prime
 3. remove all multiples of x
 4. go to Step 2

- in Haskell

```
primes :: [Integer]
primes = sieve [2..] where
    sieve (x : xs) = x : sieve (filter (\ y -> y `mod` x /= 0) xs)

> take 1000 primes           -- the first 1000 primes
> takeWhile (< 1000) primes -- all primes below 1000
```

Summary

- in pure functional languages such as Haskell the result does not depend on the evaluation strategy
- different kinds of recursion
- **tail recursion** is usually efficient as it can be implemented as loop
- **seq** can be used to **enforce strict evaluation** (in particular of accumulators)
- **lazy evaluation** allows modeling of infinite lists
- **guarded recursion** is important for algorithms on infinite lists
- **infinite lists** permit to naturally formulate several algorithms
(without having to take care about boundary conditions)



Functional Programming

Week 12 – Cyclic Data Structures, Abstract Data Types

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture – Evaluation Strategies

- evaluation strategies determine order of evaluation
- three kinds: innermost, outermost, and lazy evaluation (outermost + sharing)
- in pure functional languages the result does not depend on the evaluation strategy
 - consider non-pure language with function `uNum :: Int` that asks the user for a number and returns it
 - what is result of evaluating
`f uNum where f x = x - x`
if the user will enter the two numbers 5 and 3?
 - outermost (left-to-right): $f \text{ uNum} = \text{uNum} - \text{uNum} = 5 - \text{uNum} = 5 - 3 = 2$
 - outermost (right-to-left): $f \text{ uNum} = \text{uNum} - \text{uNum} = \text{uNum} - 5 = 3 - 5 = -2$
 - innermost: $f \text{ uNum} = f 5 = 5 - 5 = 0$
- tail recursion in combination with innermost strategy can be implemented as loop
- `seq a b` enforces evaluation of `a` to WHNF and then results in `b`
 - pitfall: in the following Haskell program, `seq` does not have the required effect

```
sumAux acc 0 = acc
sumAux acc n = let accN = acc + n in sumAux (seq accN accN) (n - 1)
-- correct: = let accN = acc + n in seq accN (sumAux accN (n - 1))
```

Last Lecture – Lazy Evaluation and Infinite Data Structures

- it is possible to define infinite lists, trees, etc., e.g.,
`enumFrom x = x : enumFrom (x + 1)`
- finite parts of infinite lists can be accessed, e.g., via `take`, `takeWhile`, etc., and lazy evaluation will not enforce computation of whole infinite list
- benefit: natural definition of several algorithms without having to worry about bounds, lengths, etc.
- main algorithmic structure: guarded recursion so that new constructors are produced in each recursive evaluation step

Cyclic Data Structures

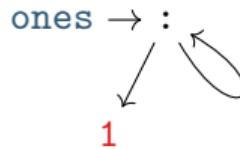
Cyclic Lists

- aim: direct definition of infinite lists which are implicitly computed on demand via lazy evaluation
- methodology: provide start of cyclic list and remaining cyclic list
- a first example: the infinite list of ones
 - starting element is 1
 - remaining list is the list of ones itself
 - Haskell definition

```
ones :: [Integer]
```

```
ones = 1 : ones
```

- created cyclic data structure



Combination of Lists

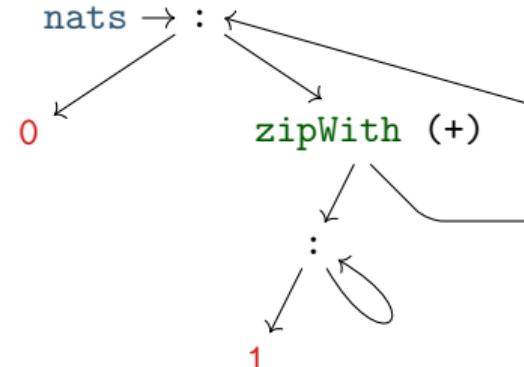
- cyclic definitions may involve auxiliary functions such as `map`, `filter`, and `zipWith`
- example: the list of natural numbers: `nats`
 - start is 0
 - remainder is addition of the list of ones with natural numbers itself

$$\begin{array}{r} 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ \dots \\ + \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \dots \\ = \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ \dots \quad (= \text{tail nats}) \end{array}$$

- in Haskell

```
nats :: [Integer]  
nats = 0 : zipWith (+) ones nats
```

- created cyclic data structure:



Computing Fibonacci Numbers

- definition:
$$fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n - 1) + fib(n - 2), & \text{otherwise} \end{cases}$$
- efficient computation of Fibonacci numbers via cyclic lists
- two starting elements: 0 and 1
- remainder is $\text{tail}(\text{tail fibs}) = \text{fibs} + \text{tail fibs}$

```
0 1 1 2 3 5 8 ...    -- fibs
+ 1 1 2 3 5 8 13 ...  -- tail fibs
= 1 2 3 5 8 13 21 ... -- tail (tail fibs)
```

- in Haskell

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

- remark: two starting elements, since otherwise `tail fibs` in rhs cannot be evaluated

Fibonacci Numbers in Haskell

- implementation was given in first lecture (slide 19 of week 1)

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

- cyclic definition of list, evaluation:

The diagram illustrates the evaluation of the infinite list comprehension `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`. It shows the list being built up step by step, with each step enclosed in a horizontal bracket. The steps are as follows:

- Initial state: `0 : 1 : zipWith (+) (tail ...)`
- Step 1: `= 0 : 1 : zipWith (+) ...` (The first `zipWith` application is shown).
- Step 2: `= 0 : 1 : zipWith (+) (0 : ...)` (The result of the first `zipWith` application is shown).
- Step 3: `= 0 : 1 : 1 : zipWith (+) ...` (The second `zipWith` application is shown).
- Step 4: `= 0 : 1 : 1 : 1 : zipWith (+) ...` (The third `zipWith` application is shown).
- Step 5: `= 0 : 1 : 1 : 2 : zipWith (+) ...` (The fourth `zipWith` application is shown).
- Step 6: `= 0 : 1 : 1 : 2 : 3 : zipWith (+) ...` (The fifth `zipWith` application is shown).

Each step shows the list growing to the right, with the `zipWith` and `tail` operations being applied to the current list to produce the next element.

Infinite Data Structures Beyond Lists

- lists are not the only infinite data structure, e.g., there are also infinite trees (vertically and/or horizontally), cf. exercise sheet 11
- also cyclic trees can be defined, e.g., consider a tree that represents all (finite and infinite) paths in the graph starting from node 1



- in Haskell we use a mutual recursive definition of four trees ([Paths](#))

```
data Paths = Root Integer [Paths]
```

```
paths1 = Root 1 [paths2]
paths2 = Root 2 [paths1, paths3]
paths3 = Root 3 [paths2, paths4]
paths4 = Root 4 []
```

Abstract Data Types

Concrete and Abstract Datatypes

- **concrete** datatypes
 - defined via `data` which defines **values** of that type
 - user defines own operations on this type via pattern matching
 - no need for primitive operations on that type
 - examples: `Rat`, `Person`, `Expr`, `Bool`, `[a]`, ...
- **abstract** datatypes
 - defined via their primitive **operations**
 - usually no access to internal structure of representation of values
 - pattern matching only via equality: `f 5 = ...` is equivalent to `f x = if x == 5 ...`
 - **abstraction barrier**: internal structure can be easily changed
 - meaning of operations usually specified
 - examples: `Char`, `Integer`, `Double`, ... which provide basic arithmetic operations and conversion to strings

Example Abstract Datatype: Queues

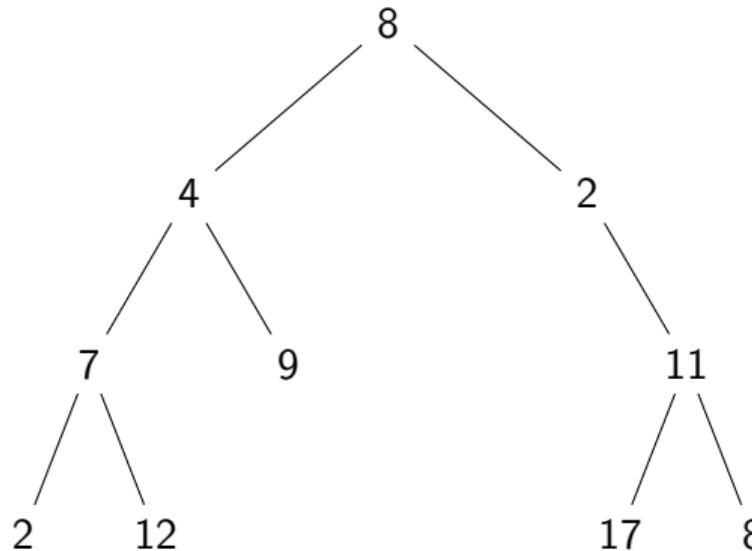
- queues are useful in computer science: printer (jobs), web-server (requests), ...
- queue provides the following operations
 - `empty :: Queue a` – the empty queue for elements of type `a`
 - `isEmpty :: Queue a -> Bool` – check whether queue is empty
 - `dequeue :: Queue a -> (a, Queue a)` – remove head of queue
 - `enqueue :: a -> Queue a -> Queue a` – add new element to end of queue

these operations in combination with their types are the **signature** of the abstract datatype `Queue a`

- signature only gives idea about operations; more information can be specified via **axiomatic specification** in the form of equations or formulas
 - `isEmpty empty`
 - `not $ isEmpty $ enqueue x q`
 - `dequeue (enqueue x empty) = (x, empty)`
 - `not $ isEmpty q -> dequeue q = (y, q') ->`
`dequeue (enqueue x q) = (y, enqueue x q')`

Example Application for Queues: Tree-Traversals

- consider binary tree



- tree-traversal: visit all nodes, e.g., to search for node, or convert nodes to list

- in-order
- depth-first search, pre-order
- breadth-first search

[2,7,12,4,9,8,2,17,11,8]

[8,4,7,2,12,9,2,11,17,8]

[8,4,2,7,9,11,2,12,17,8]

Tree Traversals in Haskell

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

inOrder :: Tree a -> [a]
inOrder Empty = []
inOrder (Node l n r) = inOrder l ++ [n] ++ inOrder r
-- preOrder is similar to inOrder

bfs :: Tree a -> [a]
bfs t = bfsMain (enqueue t empty) where
  bfsMain :: Queue (Tree a) -> [a]
  bfsMain q
    | isEmpty q = []
    | otherwise = let (t', q') = dequeue q in
      case t' of
        Empty -> bfsMain q'
        Node l n r -> n : (bfsMain $ enqueue r $ enqueue l $ q')
```

Implementing an Abstract Datatype

- implementation has to provide the desired operations and must satisfy the specification (informal text or axiomatic)
 - `empty :: Queue a`
 - `isEmpty :: Queue a -> Bool`
 - `dequeue :: Queue a -> (a, Queue a)`
 - `enqueue :: a -> Queue a -> Queue a`
 - `isEmpty empty`
 - `not $ isEmpty $ enqueue x q`
 - `dequeue (enqueue x empty) = (x, empty)`
 - `not $ isEmpty q -> dequeue q = (y, q') ->`
`dequeue (enqueue x q) = (y, enqueue x q')`
- any implementation can be used, e.g., a basic one in the beginning, which might be replaced by more efficient one later on
- if corner cases are not specified, implementation can choose freely, e.g., how `dequeue` should behave on empty queues
- modules can be used to hide internals

A Basic Implementation of Queues

```
module BasicQueue(Queue, empty, isEmpty, dequeue, enqueue) where
data Queue a = Empty | Enqueue a (Queue a)
empty = Empty
enqueue = Enqueue
isEmpty Empty = True
isEmpty (Enqueue x q) = False
dequeue (Enqueue x Empty) = (x, Empty)
dequeue (Enqueue x q) = (y, Enqueue x q') where
  (y, q') = dequeue q
dequeue Empty = error "dequeue on empty queue"
```

- implementation is rather direct translation of specification
- `empty` and `enqueue` are implemented as constructors of queues, and exported; still the constructors itself are not exported and so internal structure is not revealed, e.g., externally no pattern matching on queues is possible

Notes on the Basic Implementation of Queues

...

```
data Queue a = Empty | Enqueue a (Queue a)

isEmpty Empty = True
isEmpty (Enqueue x q) = False

dequeue (Enqueue x Empty) = (x, Empty)
dequeue (Enqueue x q) = (y, Enqueue x q') where
  (y, q') = dequeue q
dequeue Empty = error "dequeue on empty queue"
```

- we did not **prove** that implementation meets the specification; will be covered in
 - program verification (bsc), or
 - interactive theorem proving (msc)
- implementation is inefficient, since first enqueueing n elements and then dequeuing n elements requires $\sim \frac{1}{2}n^2$ evaluation steps

Towards a More Efficient Implementation of Queues

- previous queue-type is essentially a list where the list head represents the end of the queue (queue = reversed list)
- assume customers 1, 2, 3 and 4 enqueue in that order, then the representation is [4, 3, 2, 1]
- enqueueing is efficient since it just adds element in front of list
- dequeuing is expensive since it traverses and rebuilds whole list
- new version: store queue as pair of two lists: (front, rear)
 - front part of queue (head of queue is head of list)
 - rear part of queue in reverse order (tail of queue is head of list)
 - invariant: whenever front part of queue is empty then whole queue is empty
- example queue with customers 1, 2, 3, 4 has multiple representations
 - ([1,2,3,4], [])
 - ([1,2,3], [4])
 - ([1], [4,3,2])
 - ([] , [4,3,2,1])
- advantage: often constant time access to both ends of queue



More Efficient Implementation of Queues

```
module BetterQueue(Queue, empty, isEmpty, dequeue, enqueue) where
type Queue a = ([a], [a])
empty :: Queue a
empty = ([], [])
isEmpty :: Queue a -> Bool
isEmpty (front, _) = null front
enqueue :: a -> Queue a -> Queue a
enqueue x (front, rear) = maybeMtf (front, x : rear)
dequeue :: Queue a -> (a, Queue a)
dequeue ([] , _) = error "dequeue on empty queue"
dequeue (x : front, rear) = (x, maybeMtf (front, rear))
maybeMtf ([] , rear) = (reverse rear, [])
maybeMtf q = q
```

Efficiency of More Efficient Implementation

```
dequeue ([] , _) = error "dequeue on empty queue"  
dequeue (x : front, rear) = (x, maybeMtf (front, rear))
```

```
maybeMtf ([] , rear) = (reverse rear, [])  
maybeMtf q = q
```

- move-to-front operation required when `front` is empty (obey invariant)
- single move-to-front operation may be expensive, but these operations are rare
- efficiency: n queue operations require at most $2n$ evaluation steps
- proving technique: **amortized cost analysis**, will be covered in course algorithms and data-structures

Abstraction Barrier of More Efficient Implementation

```
module BetterQueue(Queue, empty, isEmpty, dequeue, enqueue) where  
  
type Queue a = ([a], [a])  
...  
empty :: Queue a  
...
```

- since `type` is just an abbreviation:

```
empty :: ([a], [a])
```

- since pairs and lists are visible, external users can completely inspect internal structure and create queues which are not permitted, e.g., `isEmpty ([] , [4,3,2,1])` evaluates to `True`

- since `type` is just an abbreviation, in particular `Queue`'s are instances of `Eq`, `Show`, and `Ord`, which might not be intended

- simple solution: hide representation in new datatype

```
data Queue a = Queue ([a], [a])
```

Implementation with Separate Datatype

```
module DataQueue(Queue, empty, isEmpty, dequeue, enqueue) where

data Queue a = Queue ([a], [a])                                -- new datatype

empty :: Queue a
empty = Queue ([] , [])                                     -- wrap Queue constructor around

isEmpty :: Queue a -> Bool
isEmpty (Queue (f, _)) = null f                            -- unwrap Queue constructor

queue = Queue . maybeMtf

enqueue :: a -> Queue a -> Queue a
enqueue x (Queue (f, r)) = queue (f, x : r)

dequeue :: Queue a -> (a, Queue a)
dequeue (Queue ([] , _)) = error "dequeue on empty queue"
dequeue (Queue (x : f, r)) = (x, queue (f, r))

maybeMtf ([] , r) = (reverse r, [])
maybeMtf q = q
```

Newtype

```
data Queue a = Queue ([a], [a])  
  
queue = Queue . maybeMtf  
  
enqueue :: a -> Queue a -> Queue a  
enqueue x (Queue (f, r)) = queue (f, x : r)  
...
```

- always wrapping and unwrapping the `Queue` constructor has some efficiency penalty
- more efficient version to hide an implementation type: `newtype`
- syntax: `newtype TName tvars = CName typ`
 - only `one` constructor (`CName`) allowed
 - this constructor must have exactly one argument type
 - nearly equivalent to `data TName tvars = CName typ`,
one difference: newtype is faster (`CName` won't be created at runtime)
- minimal change in implementation of queues
 - `newtype Queue a = Queue ([a], [a])` instead of
`data Queue a = Queue ([a], [a])`

Summary

- **cyclic lists**
 - implicit definition of infinite lists
 - can be used to elegantly and efficiently implement some functions (Fibonacci)
 - another example: see exercise sheet 12
- **abstract datatypes**: specify operations with their properties;
introduces **abstraction barriers** that permit change of implementations
- example: different implementations of **queues**
- **newtype** is efficient variant of **data** in case there is only one constructor with one argument
- example abstract datatypes
 - known: `Queue`, `Double`, `Char`, `Integer`, ...
 - further examples: sets (`Data.Set`), stacks (`Data.Stack`), dictionaries (`Data.Map`), ...



Functional Programming

Week 13 – Lambda Calculus, Summary

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Last Lecture

- cyclic definitions, e.g., `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`
- abstract data types
 - specify type of operations and behavior
 - hide implementation details (via suitable module export-lists)
 - example: queues
 - used to implement breadth-first-search in trees
 - basic implementation was simple, n operations require $\sim \frac{1}{2}n^2$ evaluation steps
 - improved implementation represents queues as two lists, n operations require $\sim 2n$ eval. steps

λ -Calculus

A Glimpse of λ -Calculus

- λ -calculus works on **λ -terms**, which is either a λ -abstraction, a variable, or an application
- no types, no data type definitions, no function definitions, no built-in arithmetic, . . .
- only one evaluation mechanism: **β -reduction**

replace $(\lambda x \rightarrow s) t$ by $s[x/t]$

where $s[x/t]$ is the term s where the variable x is substituted by t

- sufficiently strong to encode functional programs

Booleans in λ -Calculus

- encode Booleans as λ -terms, i.e., implement `Bool` as abstract data type
 - internal construction of provided operations
 - `Bool`: $a \rightarrow a \rightarrow a$
 - `True`: $\lambda x y \rightarrow x$
 - `False`: $\lambda x y \rightarrow y$
 - `if-then-else`: $\lambda c t e \rightarrow c t e$
 - satisfied axioms
 - $(\text{if } \text{True} \text{ then } t \text{ else } e) = t:$
$$\begin{aligned} & (\lambda c t e \rightarrow c t e) (\lambda x y \rightarrow x) t e \\ &= (\lambda t e \rightarrow (\lambda x y \rightarrow x) t e) t e \\ &= (\lambda e \rightarrow (\lambda x y \rightarrow x) t e) e \\ &= (\lambda x y \rightarrow x) t e \\ &= (\lambda y \rightarrow t) e \\ &= t \end{aligned}$$
 - $(\text{if } \text{False} \text{ then } t \text{ else } e) = e$: similar

Booleans in λ -Calculus, continued

- so far, we have λ -terms that encode `True`, `False`, and `if-then-else`
- other Boolean functions can easily be encoded
 - `b && c` = `if b then c else False`
 - `b || c` = `if b then True else c`
 - `not b` = `if b then False else True`

- example: computation of `False && True`:

```
False && True           -- unfold encoding of &&
= if False then True else False -- unfold encoding of ite, False, True
= (\ c t e -> c t e) (\ x y -> y) (\ x y -> x) (\ x y -> y)
   -- the line above is the lambda-term that is evaluated
= (\ t e -> (\ x y -> y) t e) (\ x y -> x) (\ x y -> y)
= (\ e -> (\ x y -> y) (\ x y -> x) e) (\ x y -> y)
= (\ x y -> y) (\ x y -> x) (\ x y -> y)
= (\ y -> y) (\ x y -> y)
= \ x y -> y           -- representation of False
```

Pairs in λ -Calculus

- pairs can be encoded similarly to Booleans
- we need three operations: (x, y) , fst , snd

- encoding of pairs is not typable in Haskell
- encoding of (x, y) : $\lambda c \rightarrow \text{if } c \text{ then } x \text{ else } y$
- encoding of fst : $\lambda p \rightarrow p \text{ True}$
- encoding of snd : $\lambda p \rightarrow p \text{ False}$

- soundness, e.g., $\text{snd } (x, y) = y$

```
   $\text{snd } (x, y)$                                      -- expand snd and  $(x, y)$ 
=  $(\lambda p \rightarrow p \text{ False}) (\lambda c \rightarrow \text{if } c \text{ then } x \text{ else } y)$            -- beta
=  $(\lambda c \rightarrow \text{if } c \text{ then } x \text{ else } y) \text{ False}$                          -- beta
=  $\text{if False then } x \text{ else } y$                                          -- soundness of ite
=  $y$ 
```

- using pairs, we can model tuples and lists

Church Numerals

- also natural numbers can be represented in λ -calculus
- Church numerals: n is encoded as $\lambda f x \rightarrow f(f \dots (f x) \dots)$ with n applications of f
- encoding type of natural numbers: $(a \rightarrow a) \rightarrow a \rightarrow a$
- examples
 - zero: $\lambda f x \rightarrow x$
 - one: $\lambda f x \rightarrow f x$
 - two: $\lambda f x \rightarrow f(f x)$
 - test on zero: $\lambda n \rightarrow n (\lambda b \rightarrow \text{False}) \text{ True}$
 - successor: $\lambda n f x \rightarrow f(n f x)$
 - addition: $\lambda n m f x \rightarrow n f(m f x)$
 - multiplication: $\lambda n m f x \rightarrow n(m f) x$
 - predecessor: possible, but more difficult

Recursion

- for defining general recursion, one can use the Y -combinator:

$$\text{Y} = \lambda f \rightarrow (\lambda x \rightarrow f(x x)) (\lambda x \rightarrow f(x x))$$

- important property: $\text{Y } g$ reduces to $g(\text{Y } g)$, i.e., $\text{Y } g$ is a fixpoint of g : $g(\text{Y } g) = \text{Y } g$

- recursive functions can be written as **fixpoints** of non-recursive functions

$$\text{add } x \ y = \text{if } x == 0 \text{ then } y \text{ else add } (x+1) \ (y-1)$$

-- add is fixpoint of the non-recursive function addNR

-- equality: addNR add = add

$$\text{addNR } a \ x \ y = \text{if } x == 0 \text{ then } y \text{ else } a(x+1) \ (y-1)$$

- encoding of above addition function in λ -calculus

- encode non-recursive function addNR as λ -term t similarly to previous slides
- encode add as fixpoint: $\text{add} = \text{fixpoint of addNR} = \text{Y } t$

Summary of Course

What You Should Have Learned

- definition of types and functions
 - type definitions via `type`, `newtype`, and `data`
 - specify functions in various forms: pattern matching, recursion, combination of predefined (higher-order) function, list comprehensions, ...
- understanding of types
 - parametric polymorphism and type classes
 - ability to infer most general types for simple definitions
- I/O in Haskell, do-notation, compilation with `ghc`
- definition and advantages of modules and abstract data types
- evaluation strategies, in particular Haskell's lazy evaluation
- basic knowledge of predefined types and functions within Prelude
 - types `Int`, `Integer`, `Double`, `[a]`, `Maybe a`, `Either a b`, `String`, `Char`, `Bool`, tuple
 - type classes for numbers, `Show`, `Read`, `Eq`, `Ord`
 - arithmetic and Boolean functions and operators
 - functions involving lists and strings
 - I/O: primitives for reading and writing (also into files)

What You Did Not Learn in This Course

- type inference algorithms
- compilation of functional programs
- static analysis and optimization of functional programs
- debugging and verification of functional programs
- concurrency
- more functional programming techniques (monads, functors, continuations, . . .)