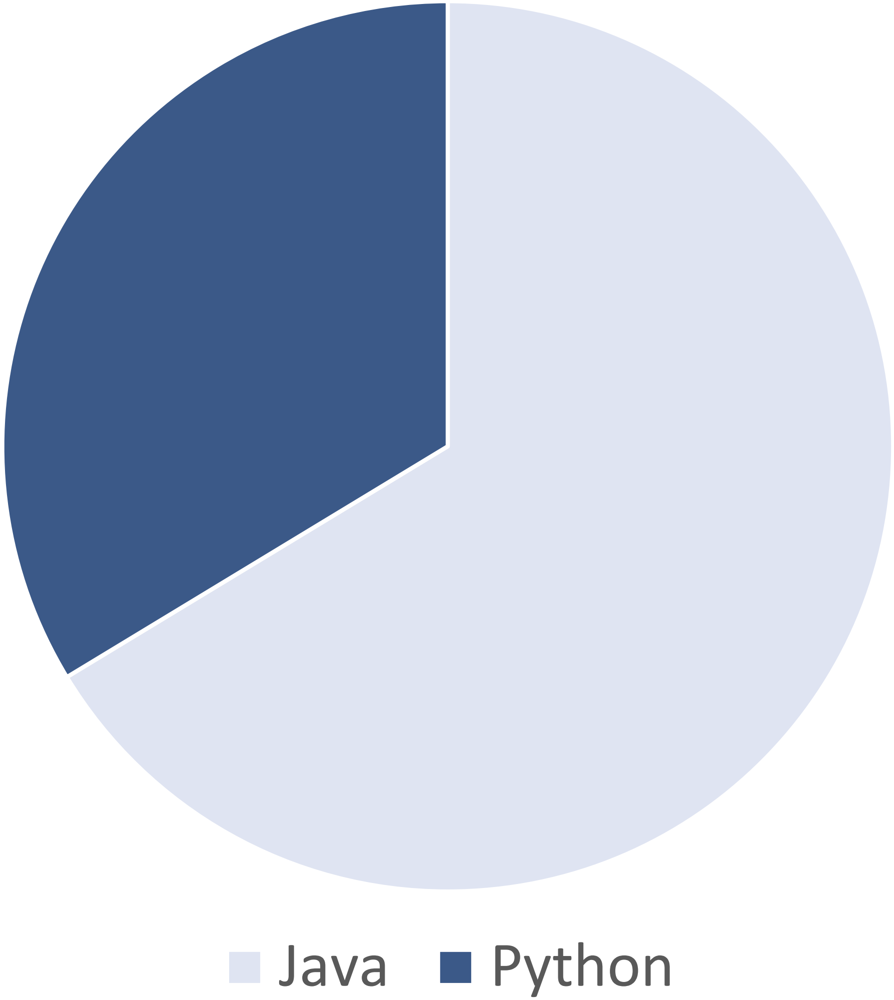
Programming Paradigms – Haskell part

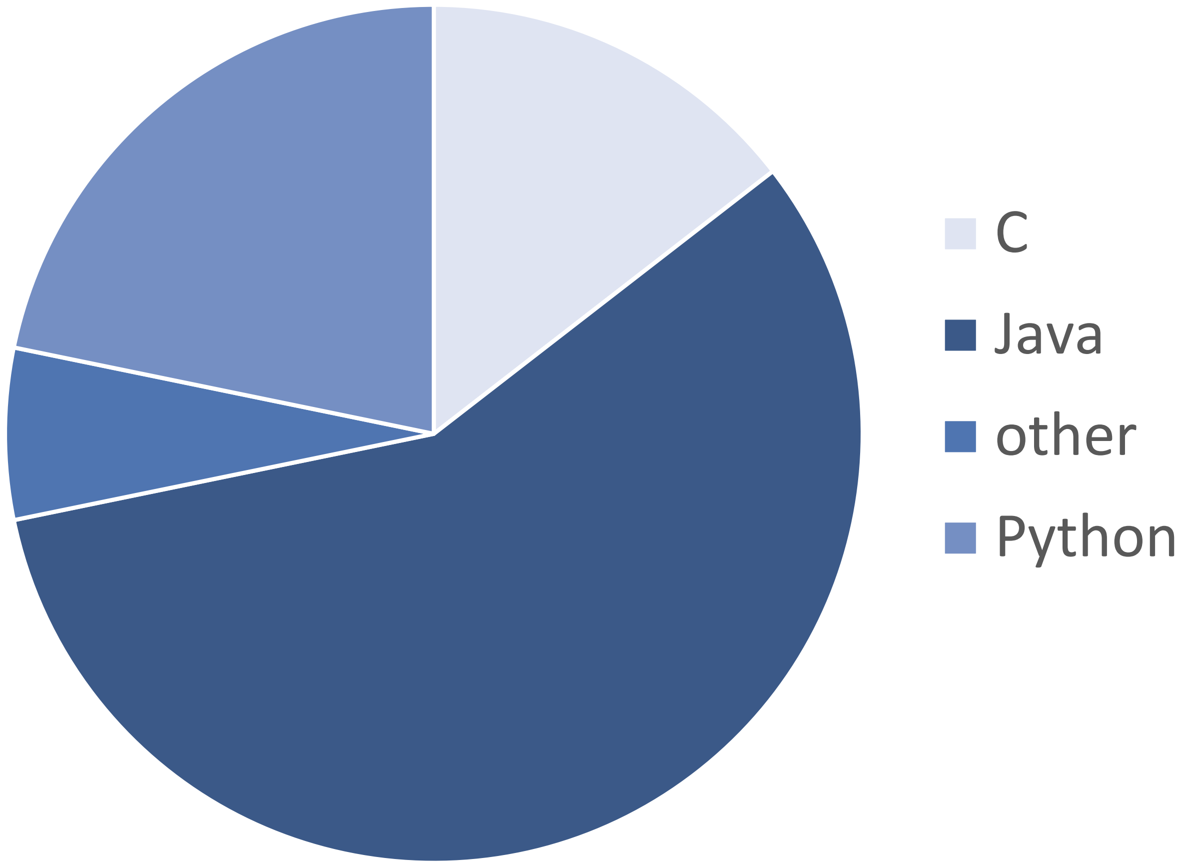
Summer Term

From a Moodle survey in 2019

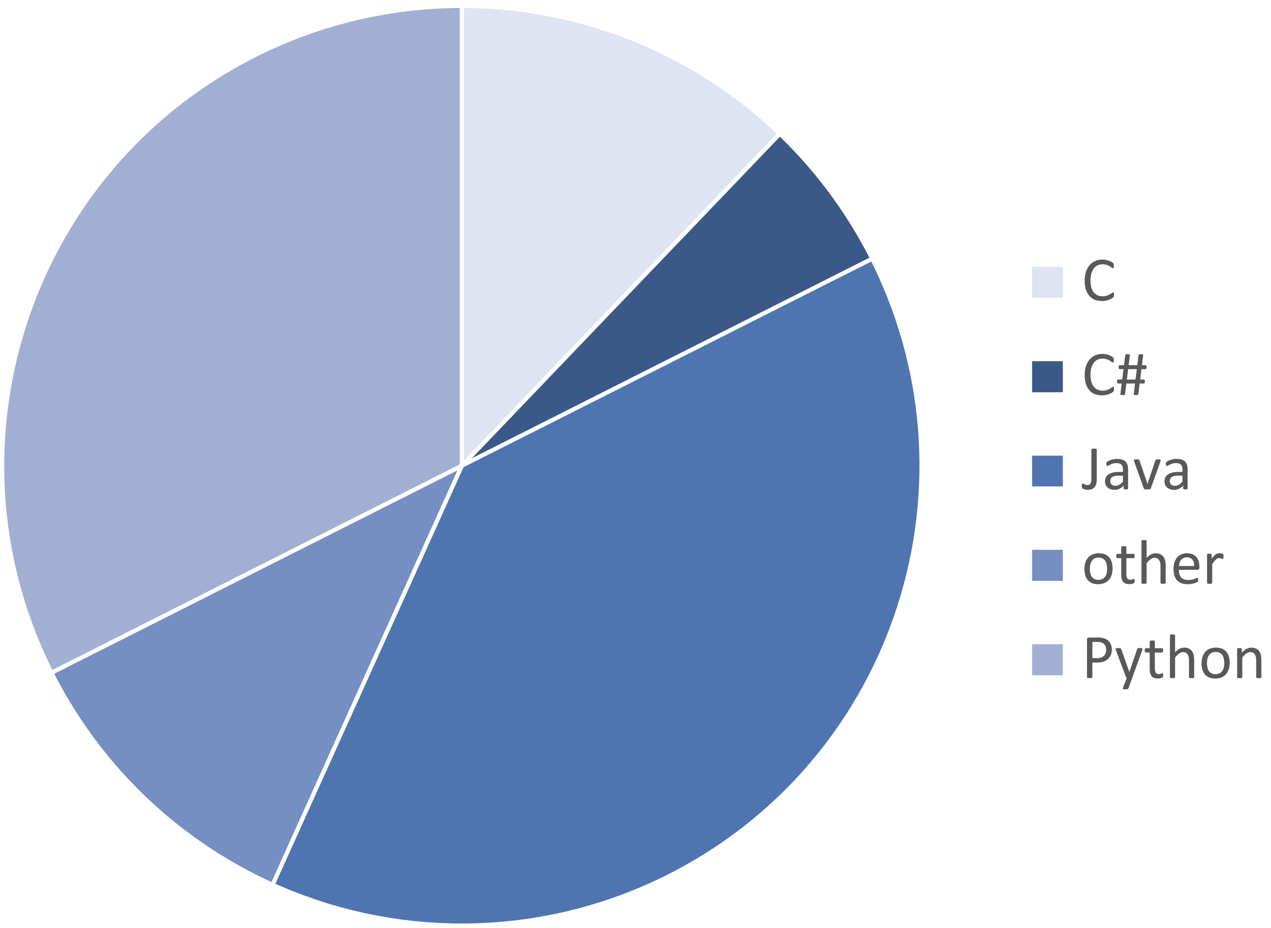
What language did you mainly use in GPT?



In what language are you most proficient?



What is your favourite programming language?



UNIVERSITÄT DUISBURG ESSEN

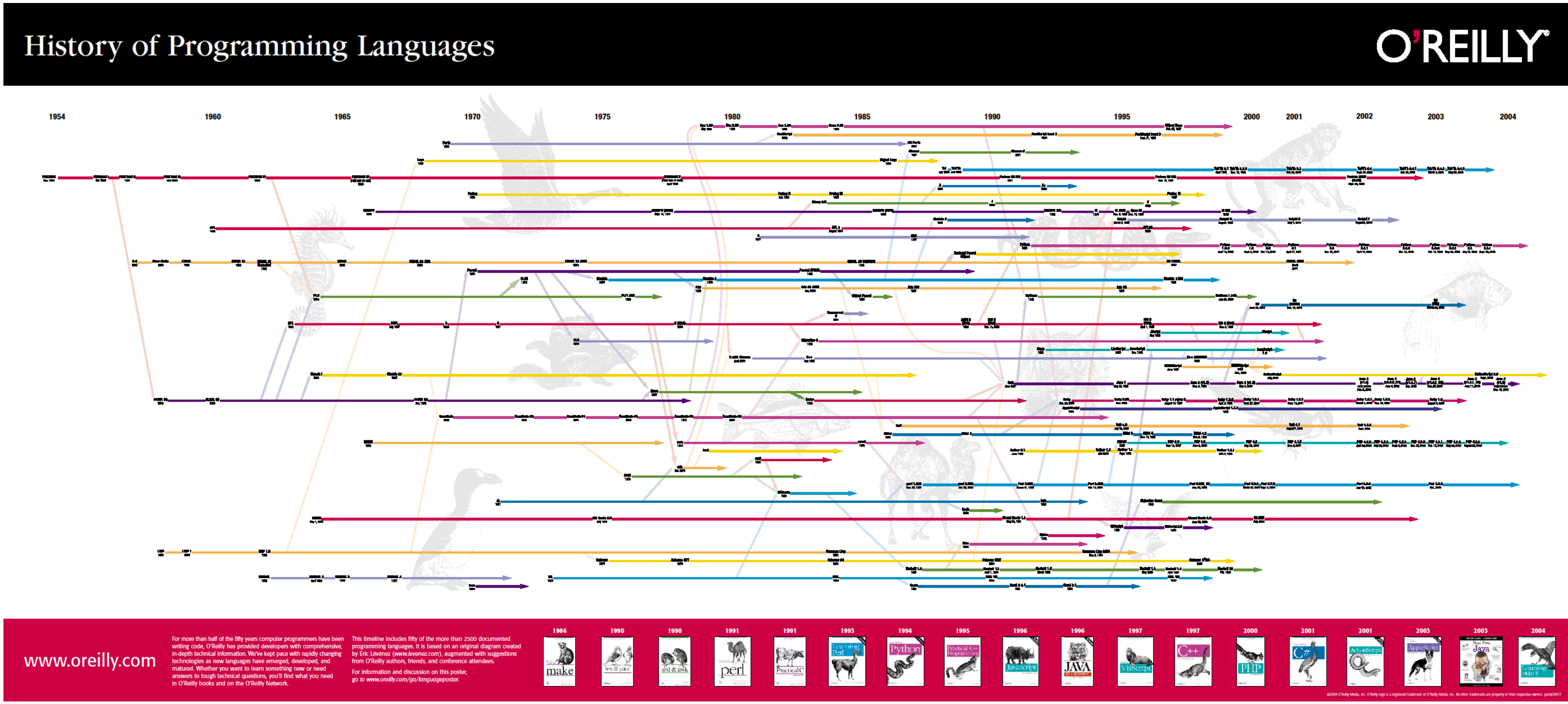
Open-Minded

Introduction / Motivation

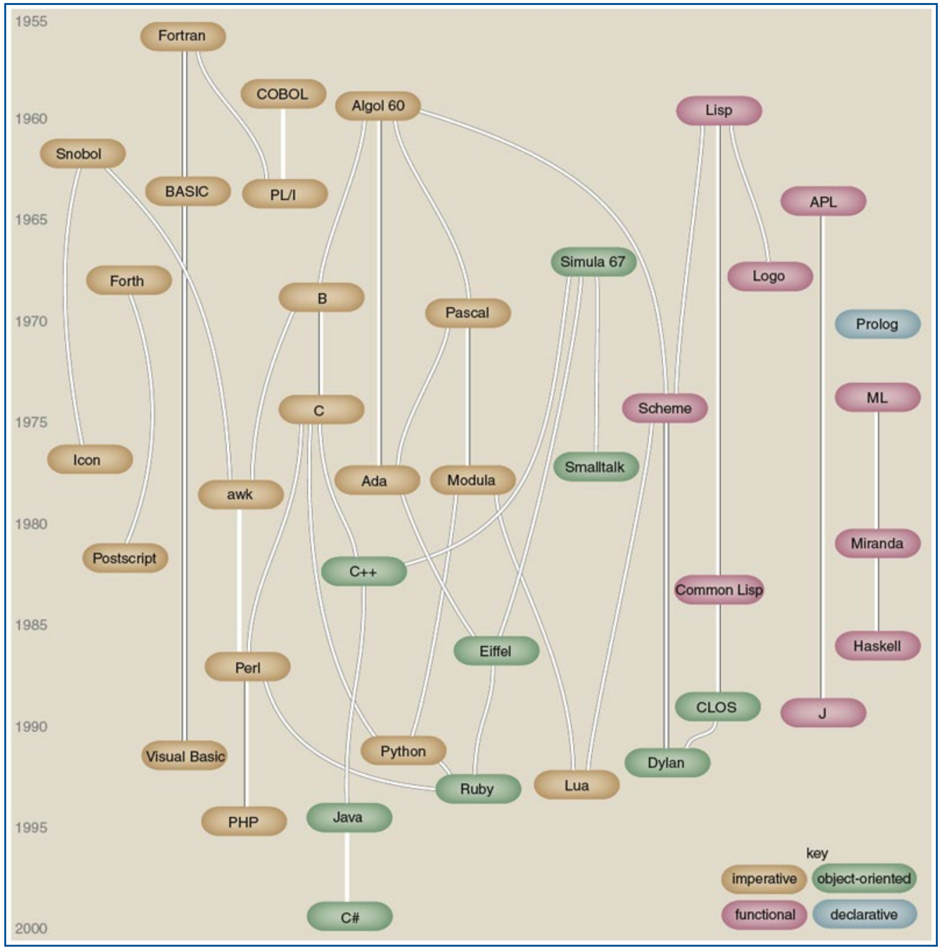
"To know another language is to have a second soul."

Charlemagne, 747/748 – 814

Many high-level programming languages in existence

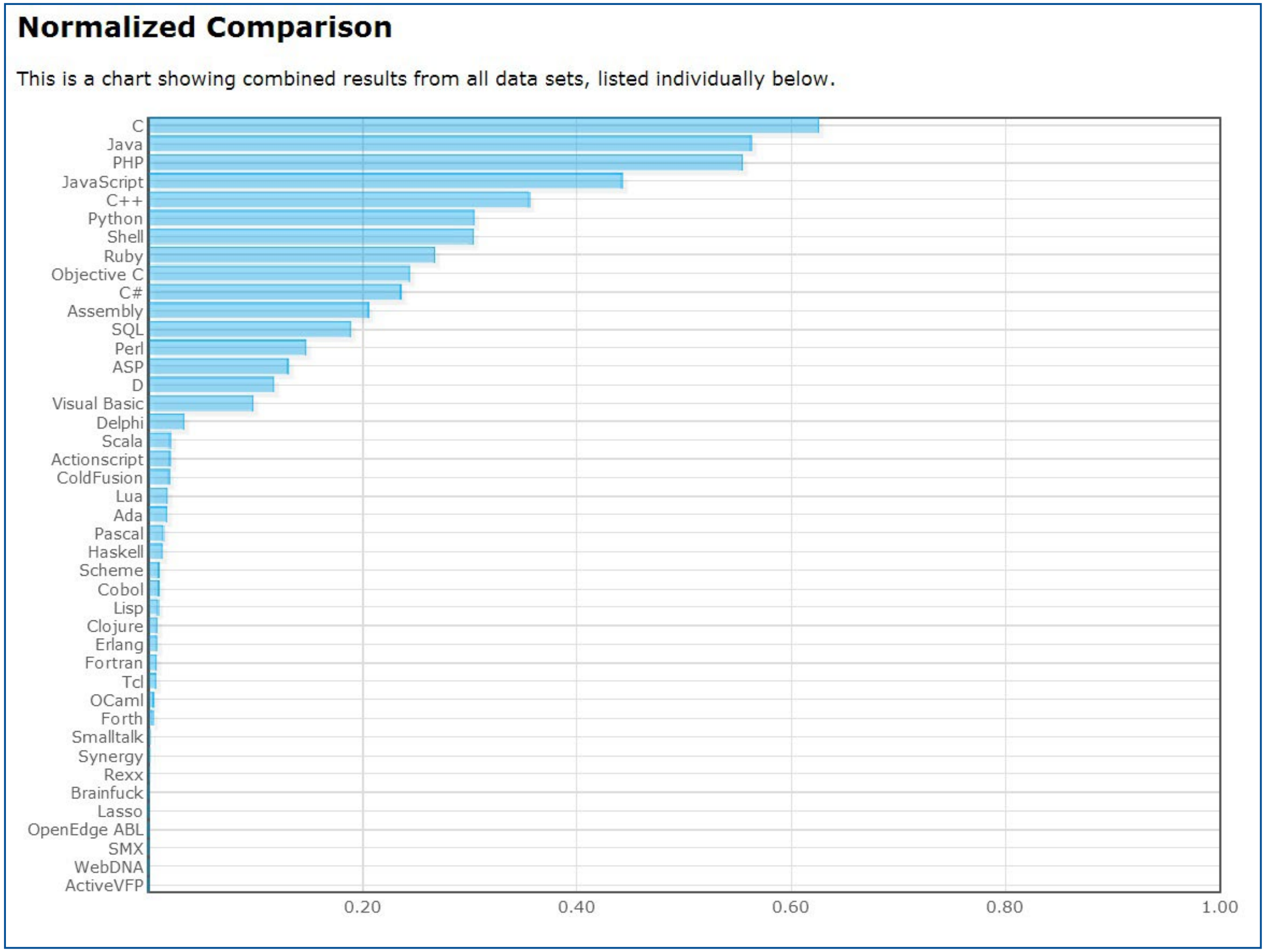


Another perspective



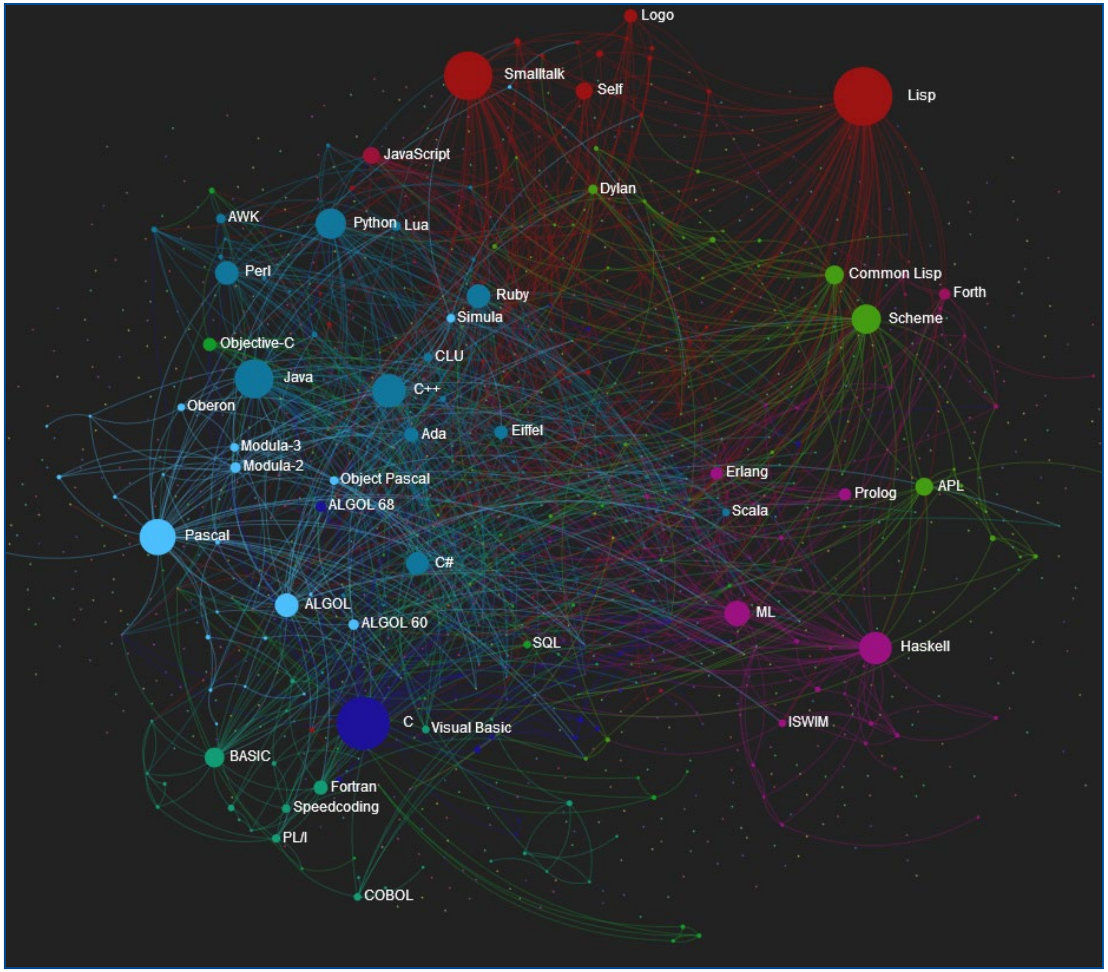
From "American Scientist": The Semicolon Wars, © 2006 Brian Hayes

Also, popularity contests, …



http://preview.tinyurl.com/popular-languages

And yet another visualization



http://preview.tinyurl.com/language-influences

So, why such diversity?

- Can one (or each) language do “more” than others?
- Are there problems that one cannot solve in certain languages?
- Is there a “best” language? At least for a certain purpose or application area?
- What does actually separate different programming languages from each other?

So, why such diversity?

Some relevant distinctions:

- syntactically rich vs. syntactically scarce (e.g., APL vs. Lisp)
- verbosity vs. succinctness (e.g., COBOL vs. Haskell)
- compiled vs. interpreted (e.g., C vs. Perl)
- domain-specific vs. general purpose (e.g., SQL vs. Java)
- sequential vs. concurrent/parallel (e.g., JavaScript vs. Erlang)
- typed vs. untyped (e.g., Haskell vs. Prolog)
- dynamic vs. static (e.g., Ruby vs. ML)
- declarative vs. imperative (e.g., Prolog vs. C)
- object-oriented vs. ???
- …

And, yet, there are common principles

Approaches to the specification of programming languages

- **… describing syntax,**
- **… describing semantics,**

as well as implementation strategies.

Language concepts:

- **variables and bindings**
- **type constructs**
- **control structures and abstraction features**

And, of course, paradigms that span a whole class of languages.

A rough plan of the lecture

- **We will focus on two paradigms: functional and logic programming.**
- **For each, we pick a specific language: Haskell, Prolog.**
- **We consider actual programming concepts, and also aspects related to semantics (evaluation, resolution).**
- **With Haskell, we explore typing concepts like inference, genericity, polymorphism.**
- **We discuss and compare concepts like variables, expressions vs. commands, etc., in different languages.**

Declarative programming

- Functional and logic programming are often called “declarative” or “descriptive” programming.
- The idea is that programmers can think more in terms of “What?” instead of “How?”, in other words, more in terms of specification than planning a certain computation process.
- Of course, there is still a need for algorithmic thinking etc., as there is no magic.
- But it is true that declarative programming has a more high-level, sometimes mathematical, feel to it.
- Also, the “What-instead-of-How” aspect will become concrete with observations like the roles of expressions vs. commands in different languages/paradigms.
- A side benefit in declarative languages is often reduced syntax.

Other reasons for studying “new” paradigms

- **Learning different languages now makes it easier to pick up new languages later on.**
- **Concepts from once “exotic” languages make their way into “mainstream” ones.**
- **In some application domains, there is an increased demand for very disciplined, conceptually expressive, mathematics-based languages.**
- **Generally, knowing more paradigms increases capacity to express ideas.**

Literature

Books on Haskell

- Programming in Haskell, 2nd edition; Graham Hutton
- Haskell – The Craft of Functional Programming, 3rd edition; Simon Thompson
- Thinking Functionally with Haskell; Richard Bird
- Haskell-Intensivkurs; Marco Block, Adrian Neumann
- Einführung in die Programmierung mit Haskell; Manuel Chakravarty, Gabriele Keller

Books on Prolog

- **Learn Prolog Now!; Patrick Blackburn, Johan Bos, Kristina Striegnitz**
- **Programmieren in Prolog; William Clocksin, Christopher Mellish**
- **Prolog – Verstehen und Anwenden; Armin Ertl**

A first glimpse of FP with CodeWorld

A first complete animation program

```
import CodeWorld

main = animationOf scene

scene t =
    circle 8
  & colored green (solidRectangle 4 4)
  & rotated (pi/2)
    (translated 8 0 (colored red (polygon [(0,0),(1,-0.5),(1,0.5)])))
  & pictures
    [ rotated ((a+t)*pi/20)
      (rectangle (4+a) (4+a)) | a <- [ 0, 0.5 .. 9 ] ]
```

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Expressions vs. commands

Expression-based programming

- **Proposition:**
  Functional programming is about expressions, whereas imperative programming is about commands.
- **Some kinds of expressions you (probably) know:**

$$\mathbf{2 + 3 \cdot (x + 1)^2}$$

$$\boldsymbol{p \land \neg(q \lor r)}$$

SUMIF(A1:A8,"<0")

- **Generally: terms in any algebra, built from constants and functions/operators, possibly containing variables**

Properties of (pure) expressions

Expressions

- … are compositional, built completely from subexpressions,
- … often have a meaningful type,
- … have a value, which does not depend on “hidden influences”, and does not change on re-evaluation or based on the order of evaluating subexpressions.

The compositionality is not just syntactical (expressions are built from subexpressions), but extends to typing and semantics/evaluation.

UNIVERSITÄT DUISBURG ESSEN Open-Minded

Example $\mathbf{2 + 3 \cdot (x + 1)^2}$:

The constants are $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$ of type $\mathbb{Z}$.

The operators are $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $\cdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $()^2 : \mathbb{Z} \to \mathbb{Z}$.

The value of $\mathbf{2 + 3 \cdot (x + 1)^2}$ depends only on the value of $\mathbf{2}$ and the value of $\mathbf{3 \cdot (x + 1)^2}$, the latter only depends on the value of $\mathbf{3}$ and the value of $\mathbf{(x + 1)^2}$, …

Properties of (pure) expressions

- Thanks to these properties, we can easily use notation known from mathematics, for example reformulating "$2 + 3 \cdot (x + 1)^2$" as follows:
  "$2 + 3 \cdot y^2$ where $y = x + 1$".
- Also, we can apply simplifications, for example replacing exponentiation by multiplication:
  "$2 + 3 \cdot y \cdot y$ where $y = x + 1$".
- And while this example was about arithmetic expressions, the concepts apply much more generally.
- But only if we have <u>pure</u> expressions!

The situation in imperative programming languages

- So what is different in imperative programming?
- Don’t we also have expressions there?
  For example in:

```
b = 100000;
if (z > 0) {
  z = 100 + z;
  j = 0;
  while (b < 200000) {
    b = b * z / 100;
    j = j + 1; }
} else j = -1;
```

- Yes, there are expressions, but they are not the dominating syntactical construct. Commands are!

The situation in imperative programming languages

- Why is this difference relevant? What properties do commands, as opposed to expressions, not have?
- Well, for example, they are not even syntactically compositional: Not every well-formed smaller part of a command is itself a command.

```
while (b < 200000) {
  b = b * z / 100;
  j = j + 1;
}
```

- Instead, expressions occur, also keywords, …
- Moreover, commands do not always have a meaningful type.
- Or even just a value. (Try giving a value for the above block.)

The situation in imperative programming languages

- As a consequence, we cannot name arbitrary well-formed smaller parts (as opposed to what we saw for expressions and their subexpressions).
- For example, we cannot simply write:

```
body = {
  b = b * z / 100;
  j = j + 1;
}
while (b < 200000) body;
```

- Even workarounds involving functions/procedures/methods are not as flexible and useful as the kind of mathematical notation for expressions: “$2 + 3 \cdot y^2$ where $y = x + 1$”.

The situation in imperative programming languages

- Okay, so what about the sublanguage of expressions in an imperative language? Can they, at least, be treated as we saw before?
- Not in general! For example, we saw that mathematically we should be able to rewrite something like "$exp_1 + exp_2 \cdot (exp_3)^2$" as any of:

$$exp_1 + exp_2 \cdot var^2 \quad \text{where } var = exp_3$$
$$exp_1 + exp_2 \cdot var \cdot var \quad \text{where } var = exp_3$$
$$exp_1 + exp_2 \cdot exp_3 \cdot exp_3$$

- But code snippets like "`result = exp₁ + exp₂ * (exp₃)^2;`" do not always take well to being replaced by:

```
var = exp₃; result = exp₁ + exp₂ * var^2;
```

- … or by code snippets corresponding to the other expression alternatives above.

The situation in imperative programming languages

- Indeed, consider these four code snippets:

```
result = exp₁ + exp₂ * (exp₃)^2;
var = exp₃; result = exp₁ + exp₂ * var^2;
var = exp₃; result = exp₁ + exp₂ * var * var;
result = exp₁ + exp₂ * exp₃ * exp₃;
```

- And imagine instantiations with `exp`$_3$ being the "expression" `i++` or some invocation `f()` for a procedure/method `f`.

- The problem is that expressions in an imperative language are typically not pure expressions. Instead, they have side-effects!
- (For same reason, re-evaluation of an expression can change the value. And order of evaluating subexpressions becomes relevant.)

So what?

- So, how “bad” is all that?
- Do these artificial examples “prove” anything?

- Well, I haven’t (yet?) really argued that the pure expression-based style is better in some sense.

- But what should have become clear is that it is <u>different</u>!

- In any case, let us (again) “do” something with CodeWorld. (… also in your first exercise tasks)

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Another look at FP with CodeWorld

Describing a picture via an expression

A rather simple example:

```
main :: IO ()
main = drawingOf scene

scene :: Picture
scene = circle 0.1 & translated 3 0 (colored red triangle)

triangle :: Picture
triangle = polygon [(0,0),(1,-0.5),(1,0.5)]
```

Let us discuss this from the “expression” perspective …

Brief recap from last week

- **Expressions: syntactic structures one could imagine after the "`=`" in an assignment "`var = …`" in C or Java.**
- **Values: results of evaluating expressions, obtained by combining values of subexpressions.**
- **Commands: syntactic structures that are characterized not so much by what (if anything at all) they evaluate to, but rather by what effect they have (change of storage cells, looping, etc.).**
- **In a pure setting without commands, any two expressions that have the same value can be replaced for each other, without changing the behaviour of the program.**

Describing a picture via an expression

Observations:

- Compositionality on level of syntax, types, and values.
- Pictures are expressions/values here, can be named etc.
- Functions/operators used:

```
circle     : ℝ → Picture
polygon    : [ℝ×ℝ] → Picture
colored    : Color × Picture → Picture
translated : ℝ × ℝ × Picture → Picture
&          : Picture × Picture → Picture
```

- Properties like:

```
  translated a b (colored c d)
≡ colored c (translated a b d)
```

Describing an animation via a function

A slight variation of example from last week:

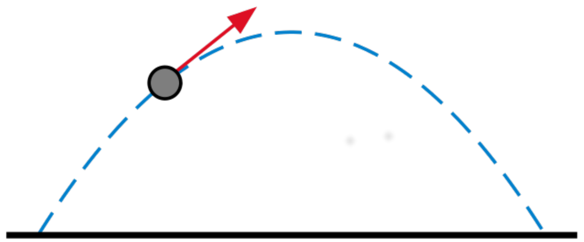
```
main :: IO ()
main = animationOf scene


scene :: Double -> Picture
scene t = translated t 0 (colored red triangle)
```

- Dependence on time expressed via parameter `t`.
- That parameter is never set by us ourselves for the animation.
- No `for`-loop or other explicit control.
- Instead, the `animationOf` construct takes care “somehow” (this involves evaluating `scene` for different `t`).

Another example

- Mathematically describing dynamic behaviour as a function of time should not be much of a surprise.
- A well-known physics example:

$$x(t) = v_{0x} \cdot t$$
$$y(t) = v_{0y} \cdot t - \frac{g}{2} \cdot t^2$$



- As a program:

```
scene :: Double -> Picture
scene t = cliff & translated x y (circle 0.1)
  where x = 3 * t
        y = 6 * t - 9.81 / 2 * t^2
        cliff = polyline [(-5,0),(0,0),(0,-2)]
```

Rich expressions

A desire for additional expressivity

- In the examples today, we have already expressed continuous distribution, throughout time, via functions.
- What if we also, or alternatively, want a discrete distribution, “throughout space”?
- So, instead of one triangle moving in time, we want several static triangles at different places.
- But we do not really want to replicate these “by hand”.
- Maybe now is the time for a `for`-loop?
- No, we don’t have that.
- But what do we have instead?

One kind of richer expressions: list comprehensions

Using a list comprehension:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])

scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- With `pictures :: [ Picture ] -> Picture`.
- And a list comprehension `[ scene d | d <- [0..5] ]`.
- This is not exactly like a `for`-loop, for several reasons.
- Instead, it is like a mathematical set comprehension $\{ 2 \cdot n \mid n \in \mathbb{N} \}$.

More mundane examples of list comprehensions

```
> [1,3..10]
[1,3,5,7,9]

> [ x^2 | x <- [1..10], even x ]
[4,16,36,64,100]

> [ y | x <- [1..10], let y = x^2, mod y 4 == 0 ]
[4,16,36,64,100]

> [ x * y | x <- [1,2,3], y <- [1,2,3] ]
[1,2,3,2,4,6,3,6,9]
```

More mundane examples of list comprehensions

```
> [ (x,y) | x <- [1,2,3], y <- [4,5] ]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [ (x,y) | y <- [4,5], x <- [1,2,3] ]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

> [ (x,y) | x <- [1,2,3], y <- [1..x] ]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

> [ x ++ y | (x,y) <- [("a","b"),("c","d")] ]
["ab","cd"]
```

So where are we, expressivity-wise?

Some takeaways from examples we have seen:

- **Non-constant behaviour expressed as functions, in the mathematical sense.** $\boldsymbol{f(x) = \cdots}$
- **Such a description defines the behaviour "as a whole", not in a "piecemeal" fashion.**
- **For example, there is no "first run this piece of animation, then that piece, and then something else".**
- **Actually, there is not even a concept of "this piece of animation stops at some point".**

Of course, we should be able to also express possibly non-continuous behaviours. But we are <u>not</u> resorting to sequential commands, with imperative keywords or semicolons etc.

List comprehensions are also not the answer, because they do not define functions, just (list) values. Instead, …

Case distinctions

- **Switching by conditional expressions:**

```
scene :: Double -> Picture
scene t = if t < 3
            then translated t t (circle 1)
            else blank
```

- **This is very much in line with case distinctions in mathematical functions:**

$$f(x) = \begin{cases} -x, & if\ x < 0 \\ x, & else \end{cases}$$

Comparison to the situation in imperative setting

- In C/Java we have two forms of `if` on commands:

```
if (...) { ... }
if (...) { ... } else { ... }
```

- In an expression language, the form without `else` does not make sense, so in Haskell we always have:

```
if ... then ... else ...
```

- This corresponds to C/Java's conditional operator:

```
... ? ... : ...
```

Some usage hints on case distinctions in Haskell

- Pragmatically, an `if`-`then`-`else` expression "without an `else`" would be realized by having some "neutral value" in the `else`-branch. Remember:

```
scene :: Double -> Picture
scene t = if t < 3
          then translated t t (circle 1)
          else blank
```

- Similarly, in a list context: `if condition then list else []`

- Also, do not hesitate to use `if`-`then`-`else` as subexpressions freely:

$$\begin{aligned} & \texttt{f x y (if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3\texttt{)} \\ \equiv\ & \texttt{if } exp_1 \texttt{ then f x y } exp_2 \texttt{ else f x y } exp_3 \end{aligned}$$

Some remarks on syntax and types

“Oddities” of syntax at the type level

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

Instead of:

```
circle     : ℝ → Picture
polygon    : [ ℝ × ℝ ] → Picture
colored    : Color × Picture → Picture
translated : ℝ × ℝ × Picture → Picture
&          : Picture × Picture → Picture
```

type signatures actually look like this:

```
circle     :: Double -> Picture
polygon    :: [ (Double, Double) ] -> Picture
colored    :: Color -> Picture -> Picture
translated :: Double -> Double -> Picture -> Picture
(&)        :: Picture -> Picture -> Picture
```

“Oddities” of syntax at the expression/function level

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

- Instead of `f(x)` and `g(x,y,z)`, we write `f x` and `g x y z`.
- As an example for nested function application, instead of `g(x,f(y),z)`, we write `g x (f y) z`.
- The same syntax is used at function definition sites, so something like

```
float f(int a, char b)
{ ... }
```

in C or Java would correspond to

```
f :: Int -> Char -> Float
f a b = ...
```

in Haskell.

Layout-sensitivity

In Haskell, this:

```
let y = a * b
    f x = (x + y) / y
in f c + f d
```

is equivalent to:

```
let { y = a * b; f x = (x + y) / y }
in f c + f d
```

But these are not accepted:

```
let y = a * b
      f x = (x + y) / y
in f c + f d
```

```
let y = a * b
   f x = (x + y) / y
in f c + f d
```

Other syntax remarks

- Haskell beginners tend to use unnecessarily many brackets. For example, no need to write `f (g (x))` or `(f x) + (g y)`, since `f (g x)` and `f x + g y` suffice.

- Further brackets can sometimes be saved by using the `$` operator, for example writing `f $ g x $ h y` instead of `f (g x (h y))`. I don't like it in beginners' code.

- We let Autotool give warnings about redundant brackets, as well as about overuse of `$`.
  Sometimes we <u>enforce</u> adherence to those warnings.

A specific observation based on exercise submissions

If you have repeated occurrences of a common subexpression, share them! For example, instead of something like this:

```
scene t =
    if 8 * sin t > 0
    then translated (8 * cos t) (8 * sin t) ...
    else ...
```

rather write this:

```
scene t =
    let x = 8 * cos t
        y = 8 * sin t
    in if y > 0 then translated x y ... else ...
```

Specifics about number types

- Haskell has various number types: `Int`, `Integer`, `Float`, `Double`, `Rational`, …
- Number literals can have a different concrete type depending on context, e.g., `3 :: Int`, `3 :: Float`, `3.5 :: Float`, `3.5 :: Double`
- For general expressions there are overloaded conversion functions, for example `fromIntegral` with, among others, any of the types `Int -> Integer`, `Integer -> Int`, `Int -> Rational`, …, and `truncate`, `round`, `ceiling`, `floor`, each with any of the types `Float -> Int`, `Double -> Integer`, …

… and arithmetic operators

- Operators are also overloaded, and often no conversion is necessary, for example in `3 + 4.5` or also in:

```
f x = 2 * x + 3.5
g y = f 4 / y
```

- In other cases, conversion is necessary, for example in this:

```
f :: Int -> Float
f x = 2 * fromIntegral x + 3.5
```

or:

```
f x = 2 * x + 3.5
g y = f (fromIntegral (length "abcd")) / y
```

… and arithmetic operators

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

- Some operators are available only at certain types, e.g., no division symbol "`/`" on integer types.

- Instead, the function `div :: Int -> Int -> Int` (also on `Integer`).

- Binary functions (not just arithmetic ones) can be <u>used</u> like operators, for example writing `` 17 `div` 3 `` instead of `div 17 3`.

- Useful mathematical constants and functions exist, e.g., `pi`, `sin`, `sqrt`, `min`, `max`, …

Some observations based on past years' exercises

- In case of doubt concerning number conversions, it usually does not hurt to add some `fromIntegral`-calls, which in the worst case will be no-ops (since, among others, `fromIntegral :: Int -> Int`).

- It is always a good idea to write down type signatures for (at least) top-level functions. Among other benefits, it saves you from having to deal with (errors involving) types like:

```
fun :: (Floating a, Ord a) => a -> a
```

UNIVERSITÄT DUISBURG ESSEN Open-Minded

Other pre-existing types:

- Type `Bool`, with values `True` and `False` and operators `&&`, `||`, and `not`.
- Type `Char`, with values `'a'`, `'b'`, …, `'\n'` etc., and functions `succ`, `pred`, as well as comparison operators.
- List types: `[Int]`, `[Bool]`, `[[Int]]`, …, with various pre-defined functions and operators.
- Character sequences: `type String = [Char]`, with special notation `"abc"` instead of `['a','b','c']`.
- Tuple types: `(Int,Int)`, `(Int,String,Bool)`, `((Int,Int),Bool,[Int])`, also `[(Bool,Int)]` etc.

Programming by case distinction (more ways of doing it)

Expressing conditional behaviour

Remember:

- Switching by conditional expressions:

```
scene :: Double -> Picture
scene t = if t < 3
             then translated t t (circle 1)
             else blank
```

- This is very much in line with case distinctions in mathematical functions:

$$f(x) = \begin{cases} -x, & if\ x < 0 \\ x, & else \end{cases}$$

- **Likely not yet seen, function definition using guards:**

```
scene t
  | t <= pi                = ...
  | pi < t && t <= 2 * pi = ...
  | 2 * pi < t             = ...
```

- **This is again similar to mathematical notation:**

$$f(x) = \begin{cases} 0, & if\ x \leq 0 \\ x, & if\ 0 < x \leq 1 \\ 1, & if\ x > 1 \end{cases}$$

Function definition using guards

- Let us discuss some details based on this example:

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | n > 0  = n * factorial (n - 1)
```

- First of all, what about the order of clauses?
- Well, in this example, the following variant is equivalent:

```
factorial :: Integer -> Integer
factorial n
  | n > 0  = n * factorial (n - 1)
  | n == 0 = 1
```

Function definition using guards

- **What if the guard conditions overlap?**
- **Then this is okay:**

```
factorial :: Integer -> Integer
factorial n
   | n == 0 = 1
   | n >= 0 = n * factorial (n - 1)
```

but this is problematic:

```
factorial :: Integer -> Integer
factorial n
   | n >= 0 = n * factorial (n - 1)
   | n == 0 = 1
```

- **Always the first matching clause is used!**

Function definition using guards

- Even with the “correct” order:

```
factorial :: Integer -> Integer
factorial n
   | n == 0 = 1
   | n >= 0 = n * factorial (n - 1)
```

  we can have problems with some inputs.

- If no clause matches, we get a runtime error!

Function definition using guards

- **In fact, if called with appropriate settings, the compiler warns us of a potential runtime error ahead of time.**

- **We can avoid both the warning and the actual non-exhaustiveness error at runtime by having a “catch-all” clause:**

```
factorial :: Integer -> Integer
factorial n
  | n == 0    = 1
  | otherwise = n * factorial (n - 1)
```

Function definition using guards

- **In this specific case, negative inputs would still be a problem.**

- **Which we could remedy as follows:**

```
factorial :: Integer -> Integer
factorial n
  | n <= 0    = 1
  | otherwise = n * factorial (n - 1)
```

- **Some lessons: order matters (and can be exploited), exhaustiveness matters. Also, some further aspects…**

Function definition using guards

- The compiler's checks ahead of time are nice, but necessarily not perfect.
- For example, it cannot in general detect infinite recursion ahead of time. (The Halting Problem!)
- Even the "simpler" static exhaustiveness checks are not as powerful as one might sometimes hope.
- For example, one might hope that something like this:

```
f x y
  | x == y = ...
  | x /= y = ...
```

is statically determined safe. But no (and for good reason). So it is usually better to use an explicit `otherwise` clause.

Function definition using guards

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- Also, the more desirable “fix” to the issue of possible negative inputs for

```
factorial :: Integer -> Integer
factorial n
  | n == 0    = 1
  | otherwise = n * factorial (n - 1)
```

(instead of switching to `n <= 0` in the first clause) would be to statically prevent negative inputs from occurring at all, via the type system.

- But that is a topic for another lecture.

Function definition using guards

- For now, let us apply our insights to this situation considered earlier:

```
scene t
  | t <= pi                   = ...
  | pi < t && t <= 2 * pi = ...
  | 2 * pi < t                = ...
```

- Here is how this should probably look instead:

```
scene t
  | t <= pi     = ...
  | t <= 2 * pi = ...
  | otherwise   = ...
```

Function definition using guards

Some further syntactic variations:

```
factorial :: Integer -> Integer
factorial n | n == 0    = 1
factorial n | otherwise = n * factorial (n - 1)
```

```
factorial :: Integer -> Integer
factorial n | n == 0 = 1
factorial n          = n * factorial (n - 1)
```

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Another example:

```
ackermann :: Integer -> Integer -> Integer
ackermann 0 n | n >= 0 = n + 1
ackermann m 0 | m > 0  = ackermann (m - 1) 1
ackermann m n | m > 0 && n > 0
  = ackermann (m - 1) (ackermann m (n - 1))
```

This one gives some interesting non-exhaustiveness warnings.

Function definitions generally

General rules for function definitions:

- One or more equations, with or without guards.
- One or more arguments; so far, only variable names (can be anonymous) or constants.
- Uniqueness of variable names within one equation.
- Never expressions, in argument position at definition sites, that would require computation or “solving”.

Function definitions generally

A few more examples:

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

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

(&&) :: Bool -> Bool -> Bool
b && True = b
_ && _    = False
```

Specific observations from exercises

If the Autotool/`hlint` feedback mentions "eta reduction", here is what it means:

- **Instead of something like:**

```
ball :: Double -> Picture
ball t = solidCircle t
```

one might just as well write:

```
ball :: Double -> Picture
ball = solidCircle
```

- **Also consider:**

```
opening :: Double -> Picture
opening = rectangle 10
```

Almost every time one sees a use of access-by-index in Haskell code, it was not the best choice of expression.

A typical case is if something corresponding to this:

```
whatever [ computeFrom argument
         | argument <- list ]
```

was instead written like this:

```
whatever [ computeFrom (list !! index)
         | index <- [0..(length list - 1)] ]
```

Generally working with lists

A few words about lists up front

- We will consider a lot of examples in the lecture and exercises that deal with lists.
- But that is mostly for didactical reasons. In the “real world”, there are often more appropriate data structures (and we will eventually see how to define them ourselves).
- In part due to historical precedent (Lisp), Haskell has a very rich library of list processing functions.
- It also has specific syntactical support for lists (e.g., list comprehensions).
- As already mentioned, Haskell lists are homogeneous.

Examples of existing (first-order) functions on lists

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
take 3 [1..10]            ==      [1,2,3]
drop 3 [1..10]            ==      [4,5,6,7,8,9,10]
null []                   ==      True
null "abcde"              ==      False
length "abcde"            ==      5
head "abcde"              ==      'a'
last "abcde"              ==      'e'
tail "abcde"              ==      "bcde"
init "abcde"              ==      "abcd"
splitAt 3 "abcde"         ==      ("abc","de")
"abcde" !! 3              ==      'd'
reverse "abcde"           ==      "edcba"
"abc" ++ "def"            ==      "abcdef"
zip "abc" "def"           ==      [('a','d'),('b','e'),('c','f')]
concat [[1,2],[],[3]]     ==      [1,2,3]
```

Different ways of working with lists

We now have certain choices, such as whether to work with recursion or by just combining existing functions (and possibly list comprehensions).

For example:

```
isPalindrome :: String -> Bool
isPalindrome s | length s < 2 = True
isPalindrome s = head s == last s &&
                 isPalindrome (init (tail s))
```

vs.:

```
isPalindrome :: String -> Bool
isPalindrome s = reverse s == s
```

Infinite lists

- In Haskell there are even expressions and values for infinite lists, for example:

```
[1,3..]                    ≡ [1,3,5,7,9,...]
[ n^2 | n <- [1..] ] ≡ [1,4,9,16,...]
```

- And while we of course cannot print complete such lists, we can still work normally with them, as long as the ultimate output is finite:

```
take 3 [ n^2 | n <- [1..] ] == [1,4,9]
zip [0..] "ab" == [(0,'a'),(1,'b')]
```

But there is no mathematical magic at work, so for example this:

```
[ m | m <- [ n^2 | n <- [1..] ], m < 100 ]
```

will “hang” after producing a finite prefix.

Why is that, actually?

Discussion: involves referential transparency!

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Essentially Quicksort:

```
sort :: [Integer] -> [Integer]
sort []    = []
sort list =
 let
  pivot = head list
  smaller = [ x | x <- tail list, x < pivot ]
  greater = [ x | x <- tail list, x >= pivot ]
 in sort smaller ++ [ pivot ] ++ sort greater
```

“Wholemeal” programming on lists

Wholemeal programming

- **"Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; …"**

  Ralf Hinze

- **"Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction."**

  Richard Bird

Wholemeal programming on lists

We earlier had this example:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])

scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- This is already a wholemeal approach, since we express the application of `scene` to the elements of `[0..5]` “in one go”.
- Specifically, we do not conceptually consider “one after another”. Instead, the resulting values are completely independent, no individual instance influences any other.
- Just like in the mathematical notation $\{\, f(n) \mid n \in \mathbb{N} \,\}$.

Wholemeal programming on lists

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

We earlier had this example:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])


scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- Of course, the individual evaluations will, on a sequential machine, happen in some order. And the resulting list is really a sequence, not a set. But the individual values will be independent of all that.
- Indeed, one can show that for any `f` and `n`, in Haskell:

```
          [ f a | a <- [0..n] ]
  ≡  reverse [ f a | a <- reverse [0..n] ]
```

Contrast to for-loops in Java, C, etc.

- In contrast, it is not remotely true that in an imperative language we can always replace a piece of code written like this:

```
for (a = 0; a <= n; a++)
   result[a] = f(a);
```

  by this:

```
for (a = n; a >= 0; a--)
   result[a] = f(a);
```

- And even for the cases where commands as above are equivalent, a formulation given that way is less useful than the Haskell equation we saw, or indeed its more general version:

```
  reverse [ f a | a <- list ]
≡ [ f a | a <- reverse list ]
```

Wholemeal programming on lists

- Another example: Assume we want to multiply each element of an array or list by its position in that data structure, and sum up over all the resulting values.
- It seems fair to say that this is a typical solution in C:

```
int array[n];
int result = 0;

for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

- And that is about okay, but it does suffer from indexitis.

Wholemeal programming on lists

- The same example, in a wholemeal fashion, in Haskell:

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

- Nice, short, declarative.
- Of course, one could consider this cheating, because it is using a conveniently predefined function `sum`.
- But actually, that is besides the point. Even without that convenience function, it would not have taken more than a dozen keystrokes to express the summation.
- And using a convenient array sum function would not exactly have made the C version any nicer than it is.

Wholemeal programming on lists

- So let us discuss the actual issues, expressivity and susceptibility to change and refactoring.
- Say, what if we decided that the counting of positions should start at 1 instead of 0?
- In the C version, that could mean we would switch from this:

```
for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

  to this:

```
for (int i = 1; i <= n; i++)
  result = result + i * array[i-1];
```

- Indexitis!

Wholemeal programming on lists

- In the Haskell version, we simply switch from this:

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

  to this:

```
sum [ i * v | (i, v) <- zip [1..] list ]
```

- To be fair again, in C we could have made a different edit:

```
for (int i = 0; i < n; i++)
  result = result + (i+1) * array[i];
```

- But actually, that is just indexitis in a different form.

Wholemeal programming on lists

- The fundamental issue in the C version is a lack of conceptual separation of values to enumerate/process on the one hand, and loop control on the other hand.
- Whereas the Haskell version has that separation in the `zip [k..] ...` expression.
- Basically, the Haskell version needs no explicit loop control, it does not access data structure elements by index (remember what I said about avoiding use of the `!!` operator whenever possible), and it does not need to increment a loop counter or talk about the “loop end” condition (because: infinite lists).

Wholemeal programming on lists

- Okay, but are we fooling ourselves, efficiency-wise?
- Certainly, code like

```
for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

is more efficient than

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

because it does not need to use extra memory, and does not need several data structure traversals?

Wholemeal programming on lists

- **Well, no. Actually, a compiler can translate the declarative code into a tight C-like loop, not using an intermediate data structure, just fine.**
- **A compiler can even spot parallelization opportunities, thanks to the "independent values" aspect we already discussed when comparing list comprehensions against `for`-loops.**
- **That all has to do also with the "lawful program construction" aspect from the Richard Bird quote.**
- **We could also talk more about refactoring…**
- **But is what we saw for the somewhat artificial example now representative of real situations? Claim: Yes!**

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Polymorphic types

Polymorphic functions on lists

- Remember that each Haskell list is homogeneous, i.e., cannot contain elements of different types.

```
"abc"   :: [Char]
[1,2,3] :: [Integer]
['a',2] -- ill-typed
```

- At the same time, functions and operators on lists can be used quite flexibly:

```
reverse "abc"   == "cba"
reverse [1,2,3] == [3,2,1]
"abc" ++ "def"  == "abcdef"
[1,2] ++ [3,4]  == [1,2,3,4]
```

- We have already depended on this flexibility a lot!

Polymorphic functions on lists

- So there should be a way to reconcile the rigidity of types with flexible use of functions.
- We want to be able to write

  `"abc" ++ "def"` and `[1,2] ++ [3,4]`,

  as well as

  `elem 2 [1,2]` and `elem 'c' "ab"`,

  but at the same time prevent calls like

  `"ab" ++ [3,4]` and `elem 'a' [1,2,3]`.

Polymorphic functions on lists

- So what are the types of functions like those seen?
- We do not have, and clearly do not want, different functions like `reverseChar :: [Char] -> [Char]` and `reverseInteger :: [Integer] -> [Integer]`.
- Instead, we use type variables, as in:

```
reverse :: [a] -> [a]
```

- That is not, at all, like being untyped. For example, the type `(++) :: [a] -> [a] -> [a]` does not mean that “anything goes”.
  (Still not possible to write this: `"ab" ++ [3,4]`.)

Polymorphic functions on lists

- We have already seen a lot of functions that fit this pattern:

```
head   :: [a] -> a
tail   :: [a] -> [a]
last   :: [a] -> a
init   :: [a] -> [a]
length :: [a] -> Int
null   :: [a] -> Bool
concat :: [[a]] -> [a]
```

- In concrete applications, the type variable gets instantiated appropriately: `head "abc" :: Char`.

Polymorphic functions on lists

- Of course, a polymorphic function does not need to be polymorphic in <u>all</u> its arguments.

- For example:

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

- And what about `zip`?

Polymorphic functions on lists

- The function `zip` also takes homogeneous lists as arguments.
- But unlike the case of `(++)`, where we want to allow `"ab" ++ "cd"` and `[1,2] ++ [3,4]`, but to disallow `"ab" ++ [3,4]`, for `zip` we want to allow all of the following:

```
zip "ab" "cd"
zip [1,2] [3,4]
zip "ab" [3,4]
```

- So the type cannot be like that for `(++)`:

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

Polymorphic functions on lists

- **Instead:**

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

- **Different type variables can be, but do not have to be, instantiated by different types.**

- **Hence, all of these make sense:**

```
zip "ab" "cd"       -- a = Char, b = Char
zip [1,2] [3,4]     -- a = Int, b = Int
zip "ab" [3,4]      -- a = Char, b = Int
```

- **Whereas a mixed call for `(++)` does not:**

```
"ab" ++ [3,4]       -- a = Char or Int?
```

Polymorphic functions in other languages

- **Have you seen something like those types in another language before?**

- **Example in Java with Generics:**

```
<T> List<T> reverse(List<T> list)
{ ... }
```

corresponding to:

```
reverse :: [a] -> [a]
reverse list = ...
```

Inference of polymorphic types

- **One aspect (among several) that distinguishes polymorphism in Haskell and its FP predecessors from those other languages is type inference.**
- **We need not declare polymorphism, since the compiler will always infer the most general type automatically.**

- For example, for `f (x,y) = x` the compiler infers `f :: (a,b) -> a`.

- And for `g (x,y) = if pi > 3 then x else y`, `g :: (a,a) -> a`.

Consequences of polymorphic types

- Polymorphism has really interesting semantic consequences.
- For example, earlier in the lecture, I mentioned that always:

```
  reverse [ f a | a <- list ]
≡ [ f a | a <- reverse list ]
```

- What if I told you that this holds, for arbitrary `f` and `list`, not only for `reverse`, but for any function with type `[a] -> [a]`, no matter how it is defined?
- Can you give some such functions (and check the above claim on an intuitive level)?

Consequences of polymorphic types

- Recall that the `reverse`-claim earlier in the lecture occurred in the context of comparing, in the imperative world, this:

```
for (a = 0; a <= n; a++)
  result[a] = f(a);
```

  vs. this:

```
for (a = n; a >= 0; a--)
  result[a] = f(a);
```

- Not only are these two loops not necessarily equivalent, but even when imposing conditions under which they are, we do not get an as general and readily applicable law as just seen in the declarative world.

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Higher-order functions

Higher-order functions

- So far, we have mainly dealt with first-order functions, that is, functions that take “normal data” as input arguments and ultimately return some value.
- But we have also already seen functions to which we passed other functions as arguments. For example, `quickCheck` and `animationOf`.
- Indeed, let us take a look at the type of the latter:

```
animationOf :: (Double -> Picture) -> IO ()
```

- Note: Every function is a (mathematical) value, but not every value is a function.

The types of higher-order functions

- The type

```
animationOf :: (Double -> Picture) -> IO ()
```

means something completely different than the type

```
animationOf :: Double -> Picture -> IO ()
```

- Indeed, parentheses in such places are very significant.

- Let us discuss this based on a simpler example type.

The types of higher-order functions

- What are some functions of the following type?

```
f :: Int -> Int -> Int
```

- And what about the following type?

```
f :: (Int -> Int) -> Int
```

- What kinds of inputs does either of these take?
- And what can they do with their inputs?

The types of higher-order functions

```
f :: Int -> Int -> Int
f x y = x + y
f x y = x - y
f _ y = y
f _ _ = 12
```

or

```
f :: (Int -> Int) -> Int
f h = h 7
f _ = 17
f h = h (h 13)
f h = h 4 + h 7
```

or

↑ pure extensional uses, not looking at the syntax of h

Functions to pass to higher-order functions

- Where do we get functions from that we can pass as arguments to higher-order functions?
- Well, in Haskell functions are almost everywhere, right? So we should not have any shortage of supply.
- Of course, there are many predefined functions already.
- We could also use functions we have explicitly defined in our program (such as passing your own `scene` function to `animationOf`).
- Or partial applications of any of those. For example, `(+) :: Int -> Int -> Int`, and as a consequence, `(+) 5 :: Int -> Int`.

Functions to pass to higher-order functions

$$f\ h = h\ 7$$

$$f\ \underbrace{((+)\ 5)}_{h} = h\ 7 = \underbrace{(+)\ 5}_{h}\ 7 = 12$$

$$f\ \underbrace{(5\ +)}_{h} = h\ 7 = \underbrace{5\ +}_{h}\ 7 = 12$$

again, h used purely extensionally!

Some syntactic specialties

- Indeed, the type `Int -> Int -> Int` could be read as `Int -> (Int -> Int)`.
- But <u>those</u> parentheses can be omitted.
- Two viewpoints here: a function that takes two `Int` values and returns one `Int` value, or a function that takes one `Int` value and returns a function that takes one `Int` value and returns one `Int` value.
- Both viewpoints are valid! No difference in usage (thanks to Haskell's function application syntax).
- Another syntactic specialty: so-called "sections". For example, "`(+) 5`" can be written as "`(5 +)`".

Some syntactic specialties

prefix: (+) , mod

infix: 1 + 2 , `mod`

(+) 5 ≡ (5 +)

also (+ 5) , semantically the same

(5 -) and (- 5) do different things

also useful for predicates:

(< 5) :: Int -> Bool

(5 -) is a function, waiting for x, computing 5 - x

Lambda-abstractions

- We can also syntactically create new functions “on the fly”, instead of predefined or own, explicitly defined and named, functions already in the program.
- Such anonymous functions use the so-called lambda-abstraction syntax (which we have already seen in the context of QuickCheck tests): `\x -> x + x`
- So, some options of functions we could pass to a function `f :: (Int -> Int) -> Int` are:
  `id, succ, (gregorianMonthLength 2019), (- 5), (\x -> x + x), (\n -> length [1..n])`

Lambda-abstractions

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

- The lambda-abstraction syntax also allows us to get a clearer view on Haskell's function definition syntax (and its choice to be different from standard mathematical function definition syntax).
- Namely, the following four definitions are equivalent (each of type `add :: Int -> Int -> Int`):

```
add x y = x + y
add x = \y -> x + y
add = \x -> \y -> x + y
add = \x y -> x + y
```

- With standard mathematical notation, `add(x,y) = …`, such variations would not have been so fluent.

Usefulness of higher-order functions

- But is any of that really useful to us?
- The examples so far look somewhat esoteric and artificial, except maybe for the `animationOf` and `quickCheck` “drivers”, which we do not know how to write ourselves yet though, anyway (due in part to the involvement of `IO`).
- Well, there are many immediately useful higher-order functions on lists as well…

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Higher-order functions on lists

Higher-order functions on lists

- For example, the function

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

  puts a (left-associative) function/operator between all elements of a non-empty list.

- So to compute the sum of such a list:

```
foldl1 (+) [1,2,3,4]
```

  which will expand to:

```
1 + 2 + 3 + 4
```

UNIVERSITÄT DUISBURG ESSEN Open-Minded

- Another useful function:

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

which applies a function to all elements of a list.

- For example:

```
map even [1..10]

map (dilated 5) [ pic1, pic2, pic3 ]
```

- And another one:

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

which selects list elements that satisfy a certain predicate.

- For example,

```
filter isPalindrome completeDictionary

filter (> 0.5) bonusPercentageList
```

Relationship to list comprehensions

- While the following are not the actual definitions of `map` and `filter`, we can think of them as such:

```
map :: (a -> b) -> [a] -> [b]
map f list = [ f a | a <- list ]

filter :: (a -> Bool) -> [a] -> [a]
filter p list = [ a | a <- list, p a ]
```

- Conversely, <u>every</u> list comprehension expression, no matter how complicated with several generators, guards, etc., can be implemented via `map`, `filter`, and `concat`.

Relationship to list comprehensions

- **Is programming with `map` and `filter` (and `foldl1` and the like) still "wholemeal programming", which is what we have mostly used list comprehensions for so far?**
- **Yes, absolutely. In a sense even more so, since higher-order functions provide a further step in the direction of more abstraction.**

- **For example, if we want to square some numbers from a given list, subject to the condition that we are specifically interested in numbers divisible by four, but still have to work out whether we want to check this divisibility before or after squaring, then …**

Relationship to list comprehensions

… with list comprehensions we would consider, and maybe experiment with,

```
[ x^2 | x <- list, x `mod` 4 == 0 ]
```

vs.

```
[ y | x <- list, let y = x^2, y `mod` 4 == 0 ]
```

While with `map` and `filter` we would simply decide between

```
map (^2) . filter (\x -> x `mod` 4 == 0)
```

and

```
filter (\x -> x `mod` 4 == 0) . map (^2)
```

Expressing laws

- Also, a law like (mentioned earlier):

```
    reverse [ f a | a <- list ]
  ≡ [ f a | a <- reverse list ]
```

  can nicely be expressed as:

```
reverse . map f ≡ map f . reverse
```

- Then we can also ask under which conditions this holds:

```
filter p . map f ≡ map f . filter q
```

- Generally, higher-order functions are a boon for “lawful program construction” (see the Richard Bird quote).

Algebraic data types

Types in Haskell

- We have so far seen various types on which functions can operate, such as number types (`Integer`, `Float`, …), other base types like `Bool` and `Char`, as well as list and tuple constructions to make compound types, arbitrarily nested (`[…]`, `(…,…)`).
- We have also seen that libraries can apparently define their own, domain specific types, such as `Picture`.
- To do the same ourselves: algebraic data types.
- These are a more general and more stringent version of what is usually known as enumeration or union types. They are also the inspiration for features like Swift’s (recursive) `enum` types.

Simple enumeration types

- Let us start simple. Assume we want to be able to talk about days of the week, and compute things like “this is a workday, yes/no”.
- We could fix some encoding of Monday, Tuesday etc. as numbers (e.g., Monday = 1, Tuesday = 2, …) and define functions like:

```haskell
workday :: Integer -> Bool
workday d = d < 6
```

- In a sense, we were lucky here that the intended property corresponds to number ranges 1–5 and 6–7.

Simple enumeration types

- So let us try to instead express on which days of the week there would have been an exercise session in the ProPa course.
- The answer this time is not a simple arithmetic comparison like `d < 6`, but we can for example implement:

```
exerciseDay :: Integer -> Bool
exerciseDay 3 = False
exerciseDay 6 = False
exerciseDay 7 = False
exerciseDay _ = True
```

- In either case, what if we call `workday` or `exerciseDay` with an input like `12`?

Simple enumeration types

- **Alternative approach, explicit new values:**

```
data Day
  = Monday | Tuesday | Wednesday | Thursday
  | Friday | Saturday | Sunday
```

- **Now:**

```
exerciseDay :: Day -> Bool
exerciseDay Wednesday = False
exerciseDay Saturday  = False
exerciseDay Sunday    = False
exerciseDay _         = True
```

… and it is impossible to pass illegal inputs (like 12th day).

- **Terminology: type constructors and data constructors.**

Simple enumeration types

- **In addition to excluding absurd inputs, we get more useful exhaustiveness (and also redundancy) checking.**
- **For example, remember the game level example:**

```
level :: (Integer, Integer) -> Integer

aTile :: Integer -> Picture
aTile 1 = block
aTile 2 = water
aTile 3 = pearl
aTile 4 = air
aTile _ = blank
```

- **Imagine that we introduce a new kind of tile, produce its new “number code” inside the `level`-function, but forget to also handle it in the `aTile`-function. No compiler warning!**

Simple enumeration types

If we had instead introduced a new type:

```
data Tile = Blank | Block | Pearl | Water | Air
```

and used `level :: (Integer, Integer) -> Tile`

and:

```
aTile :: Tile -> Picture
aTile Blank = blank
aTile Block = block
aTile Pearl = pearl
aTile Water = water
aTile Air   = air
```

then adding another value to `data Tile` could not go unnoticed in `aTile`.

The compiler would actually warn us if we forgot to handle the new value there!

General algebraic data types

- Going beyond simple enumeration types, algebraic data types can encapsulate additional values in the alternatives.
- That is, the data constructors can take arguments.
- For example:

```
data Date = Day Integer Integer Integer
data Time = Hour Integer
data Connection = Train Date Time Time
                | Flight String Date Time Time
```

- A possible value of type `Connection`:

```
Train (Day 20 04 2011) (Hour 11) (Hour 14)
```

General algebraic data types

- Computation on such types is via pattern-matching:

```
travelTime :: Connection -> Integer

travelTime (Train _ (Hour d) (Hour a))
  = a - d + 1
travelTime (Flight _ _ (Hour d) (Hour a))
  = a - d + 2
```

- At the same time, the data constructors are also normal functions, for example:

```
Day :: Integer -> Integer -> Integer -> Date

Train :: Date -> Time -> Time -> Connection
```

Recursive types

- Algebraic data types can be recursive. For example:

```
data Nat = Zero | Succ Nat
```

- Values of this type:

```
Zero, Succ Zero, Succ (Succ Zero), ...
```

- Computation by recursive function definitions:

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

Recursive types

- With several recursive occurrences, tree structures:

```
data Tree = Leaf | Node Tree Integer Tree
```

- Values: `Leaf`, `Node Leaf 2 Leaf`, …

- Computation:

```
height :: Tree -> Integer
height Leaf
  = 0
height (Node left _ right)
  = 1 + max (height left) (height right)
```

Just like functions, algebraic data types can be polymorphic:

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

height :: Tree a -> Integer
height Leaf
  = 0
height (Node left _ right)
  = 1 + max (height left) (height right)
```

Polymorphism in algebraic data types

- Another example, from the standard library:

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

- Popular for functions that would otherwise be partial.
- Such as also in a re-design of the game level example:

```
data Tile = Block | Pearl | Water | Air

level :: (Integer, Integer) -> Maybe Tile

aTile :: Tile -> Picture
aTile Block = block
aTile Pearl = pearl
aTile Water = water
aTile Air   = air
```

Persistency of data structures

- Note that, just as any other data in Haskell, values of algebraic data types are immutable.
- For example, we do not <u>change</u> any tree in a function like this:

```
insert :: Integer -> Tree Integer
                  -> Tree Integer
insert n Leaf = Node Leaf n Leaf
insert n tree@(Node left m right)
  | n < m = Node (insert n left) m right
  | n > m = Node left m (insert n right)
  | otherwise = tree
```

- Discuss what this means …

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Lists as algebraic data type

Another example data structure

- If Haskell did not yet have a list type, we could implement one ourselves:

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

- Example value: `Cons 1 (Cons 2 Nil) :: List Int`

- Computation:

```
length :: List a -> Int
length Nil              = 0
length (Cons _ rest) = 1 + length rest
```

Lists as just another algebraic data type

- In fact, modulo special syntax, that is exactly what Haskell lists are:

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

- So, for example, `[1,2]` is simply `1:(2:[])`, which thanks to right-associativity of "`:`" can also be written as `1:2:[]`.

- Functions on lists can then, of course, also be defined using pattern-matching.

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Some example functions:

```
length :: [a] -> Int
length []         = 0
length (_:rest) = 1 + length rest

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

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

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

Pattern-matching on lists

- Note how clever arrangement of cases/equations can make function definitions more succinct.
- For example, we might on first attempt have defined `zip` as follows:

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

- But the version from the previous slide is equivalent.
- Both versions also work with infinite lists, btw.

Higher-order examples

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

Also, as another example of a function we have used:

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

And indeed related:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap _ Leaf = Leaf
treeMap f (Node left x right)
  = Node (treeMap f left)
         (f x)
         (treeMap f right)
```

Higher-order examples

- **Also remember the function**

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

which puts a (left-associative) function/operator between all elements of a non-empty list.

- **It is a member of a whole family of related functions, the most prominent of which is `foldr`, defined thus:**

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

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Notes on pattern-matching

Evaluation by pattern-matching

- Ultimately, pattern-matching is what drives (lazy) evaluation in Haskell.

- For example, let us consider how the expression

```
head (tail (f [3, 3 + 1]))
```

  is evaluated, given the following function definitions (and the known `head` and `tail` functions):

```
f :: [Int] -> [Int]
f []     = []
f (x:xs) = g x : f xs
```

```
g :: Int -> Int
g 3 = g 4
g n = n + 1
```

Explicit case-expressions

- Pattern-matching is not restricted to the left-hand sides of function definitions, it can also occur inside expressions, using the `case`-keyword.

- For example, instead of something like this:

```
if maybeThing == Nothing
then ... something ...
else ... something else, using fromJust maybeThing ...
```

we can (and would usually prefer to) write this:

```
case maybeThing of
  Nothing    -> ... something ...
  Just thing -> ... something else, directly using thing ...
```

Binding of variables

- Pattern-matching always binds variable names that occur in patterns, possibly shadowing existing things of same name.

- That sometimes leads to confusion for beginners, such as why it does not work to write a function like the following one (given the existence of `red :: Color` etc., imported from `CodeWorld`):

```
primaryColor :: Color -> Bool
primaryColor red   = True
primaryColor green = True
primaryColor blue  = True
primaryColor _     = False
```

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Input / Output

“In short, Haskell is the world’s finest imperative programming language.”

Simon Peyton Jones

Input / Output in Haskell, general approach

- Even in declarative languages, there should be some (disciplined) way to embed “imperative” commands like “print something to the screen”.
- In pure functions, no such interaction with the operating system / user / … is possible.
- And clearly it should not be, since it would defy referential transparency.
- But there is a special `do`-notation in Haskell that enables interaction, and from which one can call “normal” functions.
- All the features and abstraction concepts (higher-order, polymorphism, …) of Haskell remain available even in and with `do`-code.

Input / Output in Haskell, very simple example

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- Getting two numbers from the user and then printing some value computed from them to the screen:

```
main :: IO ()
main = do n <- readLn
          m <- readLn
          print (prod [n..m])

prod :: [Integer] -> Integer
prod []       = 1
prod (x:xs)   = x * prod xs
```

- Note the (apparent) type inference on `n` and `m`.

Input / Output in Haskell, the principles

- There is a predefined type constructor `IO`, such that for every type like `Int`, `Tree Bool`, `[(Int,Bool)]` etc., the type `IO Int`, `IO (Tree Bool)`, … can be built.
- The interpretation of a type `IO a` is that elements of that type are not themselves concrete values, but instead are (potentially arbitrarily complex) sequences of input and output operations, and computations depending on values read in, by which ultimately a value of type `a` is created.
- An (independently executable) Haskell program overall always has an "`IO` type", usually `main :: IO ()`.

Input / Output in Haskell, the principles

- To actually create "`IO` values", there are certain predefined primitives (and one can recognize their `IO`-related character based on their types).
- For example, there are `getChar :: IO Char` and `putChar :: Char -> IO ()`.
- Also, for multiple characters, `getLine :: IO String` and `putStr, putStrLn :: String -> IO ()`.
- More abstractly, for any type for which Haskell knows (or was instructed) how to convert from or to strings, `readLn :: Read a => IO a` for input as well as `print :: Show a => a -> IO ()` for output.

To combine `IO`-computations (i.e., to build more complex action sequences based on the `IO` primitives), we can use the `do`-notation.

Its general form is:

```
do cmd₁
   x₂ <- cmd₂
   x₃ <- cmd₃
   cmd₄
   x₅ <- cmd₅
   ...
```

where each cmd_i has an `IO` type and to each x_i (if present) a value of the type encapsulated in the cmd_i will be bound (for use in the rest of the `do`-block), namely exactly the result of executing cmd_i.

Input / Output in Haskell, the principles

- The `do`-block as a whole has the type of the last $\texttt{cmd}_n$.
- For that last command, generally no $\texttt{x}_n$ is present.
- Often also useful (for example, at the end of a `do`-block): a predefined function `return :: a -> IO a` that simply yields its argument, without any actual `IO` action.
- What is never ever, at all, possible or allowed is to directly extract (beyond the explicit sequentialisation and binding structure within `do`-blocks) the encapsulated value from an `IO` computation, i.e., to simply turn an `IO a` value into an `a` value.

User defined “control structures”

- As mentioned, also in the context of `IO`-computations, all abstraction concepts of Haskell are available, particularly polymorphism and definition of higher-order functions.
- This can be employed for defining things like:

```
while :: a -> (a -> Bool) -> (a -> IO a)
           -> IO a
while a p body = loop a
  where loop x = if p x then do x' <- body x
                                loop x'
                        else return x
```

- Which can then be used thus:

```
 while 0
       (< 10)
       (\n -> do {print n; return (n+1)})
```