# 1. Introduction to Haskell

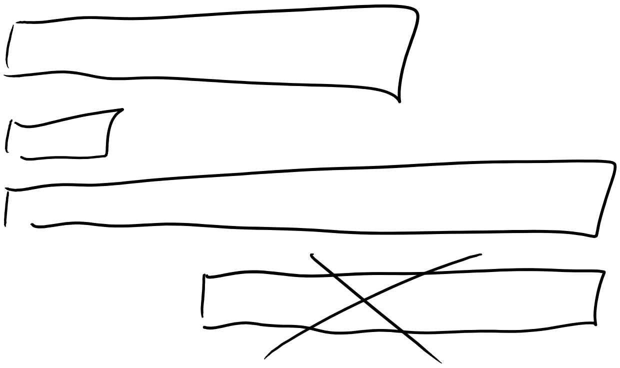
1.1. Main Constructs of Haskell
1.2.
1.3
1.4 } Prog. techniques in functional languages

## 1.1. Main Constructs of Haskell

4 Kinds of main constructs: declarations, expressions, patterns, types

### 1.1.1. Declarations

Haskell prog = sequence of declarations that have to start at the same column



Declarations are either <u>type declarations</u> or <u>function declarations</u>

We define Haskell's syntax by EBNF-rules where non-terminal symbols are underlined

## Comments in Haskell:

```
{-
  ⋮
-}
```

or `-- ...` until end of line

## Type declarations (Slide 4)

$$\text{square} :: \underbrace{\text{Int} \to \text{Int}}$$

Variables are used as names of functions

Type of Functions from Int to Int

Haskell has pre-defined types like Int, Bool, ... .

One can create new types by "->" or by "[ ... ]":

```
Int -> Bool
[Int]
[Int] -> Int
```

Type declarations do not have to be specified by the programmer (then Haskell computes them automatically).

But it is good programming style to provide them.

## Function declarations

$$\underbrace{\text{square}\ x} = \underbrace{x * x}$$

Defining equations specifying a function funlhs

variable, stands for name of a function

pattern

stands for the expected argument. Variable stands for an arbitrary argument

expression

Haskell has certain built-in operations:

arithmetic operations: +, -, *, /, ...

comparison operations: ==, >, <, >=, <=, ...

Boolean operations: not, &&, ||, ...

↗

they evaluate to True or False

## Evaluation of functional programs (Slide 5)

Program tries to simplify a given expression.

Ex: Square (12-1)

Evaluation works by term rewriting:

1. Computer searches for a sub-expression such that the left-hand side of a defining equation matches this sub-expression (i.e., the lhs is equal to the sub-expression if one instantiates its variables appropriately).
2. Replace the instantiated lhs of the equation

by the instantiated right-hand side.

```
square :: Int -> Int
square x = x * x
```

2 main evaluation strategies:

- leftmost innermost (call-by-value)
  disadvantage: one evaluates all subexpressions, even if they are not needed for the result ⇝ might lead to (unnecessary non-termination)
- leftmost outermost (call-by-name)
  disadvantage: duplicated arguments have to be evaluated several times

Haskell: Lazy Evaluation
leftmost outermost, but Haskell keeps track of duplicated arguments and only evaluates them once in parallel.

```
three :: Int -> Int          non_term :: Int -> Int
three x = 3                  non_term x = non_term (x+1)
```

`non_term 0` does not terminate

`three (non_term 0)` terminates in 1 evaluation step, result is 3

## Haskell implementation

We recommend the GHC (Glasgow Haskell Compiler), in particular the GHCi interpreter.
↑
"interactive"

## Conditional equations (Slide 6)

```
maxi :: (Int, Int) -> Int
maxi (x,y) | x >= y    = x
           | otherwise = y
```

(Int, Int) : type of tuples of 2 Components, both have type Int.
(2,3) has type (Int, Int)

- rhs of an equation can be restricted by a Boolean condition
- Conditions are checked from top to bottom, don't have to be exhaustive
- "otherwise" always evaluates to True

## Currying

named after logician Haskell B. Curry

Idea: don't provide several arguments as a tuple, but provide them individually one after another

```
plus :: Int -> (Int -> Int)
plus x y = x+y
```

$\underbrace{\text{plus 2}}$ is a function of type Int -> Int

(plus 2) 3 = 5

Here:

Int → Int → Int stands for Int → (Int → Int) → associates to the right

plus x y stands for (plus x) y

function application associates to the left

Advantages of Currying:

- less brackets
- "partial applications" are possible: one may apply a function to fewer arguments

plus 2 is an expression of type Int → Int.

It stands for the function that takes an argument and increases it by 2.

⇒ $\underline{var}$ can now be applied to several arguments $\underline{pat_1}, \ldots, \underline{pat_n}$ (Slide 7).

## Function definition by pattern matching

```
und :: Bool -> Bool -> Bool
und True  y = y
und False y = False
```

2 fct. declarations for the same fct. symbol ("und").

Patterns (True, False, y) are used to describe the expected arguments.

Data type Bool has 2 data constructors: True, False

Such functions are used to construct data objects, can't be evaluated further.

Pattern matching: perform a case analysis depending on the data constructors that the arguments are built with.

To evaluate und $\underline{exp_1}$ $\underline{exp_2}$
one first has to check, whether the lhs of the first equation matches und $\underline{exp_1}$ $\underline{exp_2}$.

If yes: use 1st equation for evaluation

If no: check the next equation

↝ equations are checked from top to bottom

↝ one could simplify "und" to:

```
und :: Bool -> Bool -> Bool
und True y = y
und x    y = False
```

```
unclear :: Int -> Bool
unclear x = not (unclear x)
```

```
und False (unclear 0) = False
und (unclear 0) False
```

does not terminate:

Haskell needs to find out whether the pattern True matches (unclear 0). To this end, it evaluates (unclear 0) step by step and checks whether True matches the result ...

Pattern matching for the data type Bool:

$\underline{\text{Bool}} \rightarrow \text{True} \mid \text{False}$

Pattern matching for data type of lists ([a]):

$$[a] \to \underbrace{[\;]}_{\text{Data constructor for empty list}} \;\Big|\; a \underbrace{:}_{\text{Data constructor for non-empty lists}} [a]$$

```
len :: [a] -> Int
len [ ] = 0
len (x:xs) = 1 + len xs
```

Pattern ≙ expression built from variables and data constructors (True, False, [ ], :, ...)

Evaluation of $\text{len}\ \underbrace{[15,70,36]}_{15:(70:(36:[\,]))\ \text{or}\ 15:70:36:[\,]}$

← : associates to the right

= 1 + len [70,36]

= ... = 3

```
second :: [Int] -> Int
second [ ] = 0
second (x:[ ]) = 0          ← second [x] = 0
second (x:y:xs) = y
```

Defining equations don't have to be exhaustive. (Slide 7)

## Pattern declarations (Slide 8)

Can be used to declare constants (functions of arity 0)

```
plus x y = x+y
```

suc 5 = plus 1 5 = 1+5 = 6

A pattern declaration assigns a certain expression to a pattern.

## Local declarations (Slide 9)

Local declarations are only valid within some other declaration.

Ex: Given $a, b, c \in \mathbb{Q}$, find $x \in \mathbb{Q}$ such that

$$ax^2 + bx + c = 0 \iff$$

$$x = \frac{-b \pm \underbrace{\sqrt{b^2 - 4ac}}_{d}}{\underbrace{2a}_{e}}$$

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - d)/e, (-b + d)/e)
        where  d = sqrt (b*b - 4*a*c)
               e = 2 * a
```

↖ Local declarations, only valid in this rhs

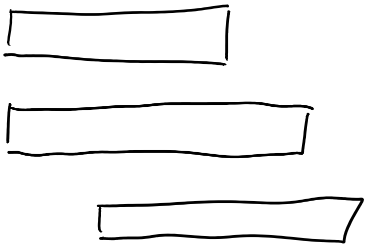
- more readable
- more efficient, because d and e are only evaluated once

## Offside-rule for layout in Haskell

1. The first symbol in a block decls of declarations determines the leftmost column of this block.

the leftmost column of this block.

2. A new line starting in this column is a new declaration of the same block.



3. A new line starting further to the right continues the declaration in the line above.

e.g.
```
d = sqrt (b*b -
           4 * a * c)
```

4. A new line starting further to the left means that the current declaration block has ended. The new line does not belong to this decl. block anymore.

```
roots a b c = ....
      where d = ....
            e = ....

square x = x*x
```

## Operators and infix declarations

up to now: functions in prefix notation: plus 2 3

Sometimes one wants to use infix notation instead:

2 + 3, 2 < 3, 2 == 3, True || False, ...

Infix functions are also called operators.

Every infix operator can be turned into a prefix function by using (...):

```
(+) 2 3      (is 2+3)
(:) 2 [3]    (is 2:[3])
```

Every prefix fct. can be turned into an infix operator by using \`...\`:

```
2 `plus` 3   (is plus 2 3)
```

Moreover, one can also define infix operators directly.

2 properties have to be determined for infix operators:

## 1. Association

```
divide :: Float -> Float -> Float
divide x y = x / y
```

```
36 `divide` 6 `divide` 2
```

If \`divide\` associates to the left, the result is

$(36/6)/2 = 3$

If \`divide\` associates to the right, the result is

$36/(6/2) = 12$

Haskell allows *infix-declarations*:

```
default → infixl `divide`     ← this means that `divide`
          infixr `divide`       associates to the left
                                resp. to the right

          infix `divide`      ← this means that `divide` does not
                                associate, i.e.,
```

``36 `d` 6 `d` 2`` is forbidden

E.g.: → associates to the right ( Int → Int → Int stands for
Int → (Int → Int) )

function applic. associates to the left
( plus 2 3 stands for (plus 2) 3 ).

square square 2 stands for
(square square) 2 ↯ type error

## 2. Binding priority

| | |
|---|---|
| (%%) :: Int → Int → Int | (@@) :: Int → Int → Int |
| x %% y = x + y | x @@ y = x * y |

What is the result of $1 \overset{+}{\widetilde{\%\%}} 2 \overset{*}{\widetilde{@@}} 3$

One can specify binding priority by a number between 0,…,9:

infixl 9 %% ← default

infixl 8 @@

Then

$1 \%\% 2 @@ 3 =$

$(1 \underbrace{\%\%}_{+} 2) \underbrace{@@}_{*} 3 = 9$

Slide 10

prefix functions: strings that start with lower-case symbol
(e.g. square)

prefix data constructors: strings that start with upper-case symbol (e.g. True)

infix functions: strings of special symbols not starting with :
(e.g., +, ==, >=, @@, ...)

infix data constructors: strings of special symbols starting with : (e.g, : )

Currying is also possible for operators:

6 `divide` has type Float -> Float.
It takes an argument y and divides 6 by y.

`divide` 6 has type Float -> Float.
It takes a number x and divides x by 6.