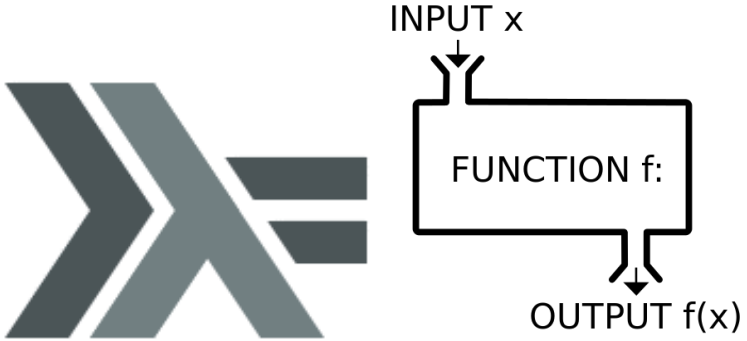
# Functional Programming




## Syntax in Functions

**Christoph Denzler / Daniel Kröni**
**University of Applied Sciences Northwestern Switzerland**
**Institute for Mobile and Distributed Systems**

# Learning Targets

You can program your own functions in Haskell

- you know basic language constructs
- you know about scoping
- you know the importance of indentation

# Content

- Function Definitions
- Pattern Matching
- Function Application
- Case Expressions
- Guards
- Conditional Expressions
- Where Bindings
- Let Expressions

# Remark: Rules for Function Application

- **Given the two functions**


  - Function application binds strongest

```
f a + b == (f a) + b
```

```
not: f (a + b)
```

  - Function application associates to the left

```
g a b == ((g a) b)
```

```
not: (g (a b))
```

  - Note, in Haskell:
    - f (a, b) is a function with **one** argument (a tuple)
    - g a b is a function with **two** arguments

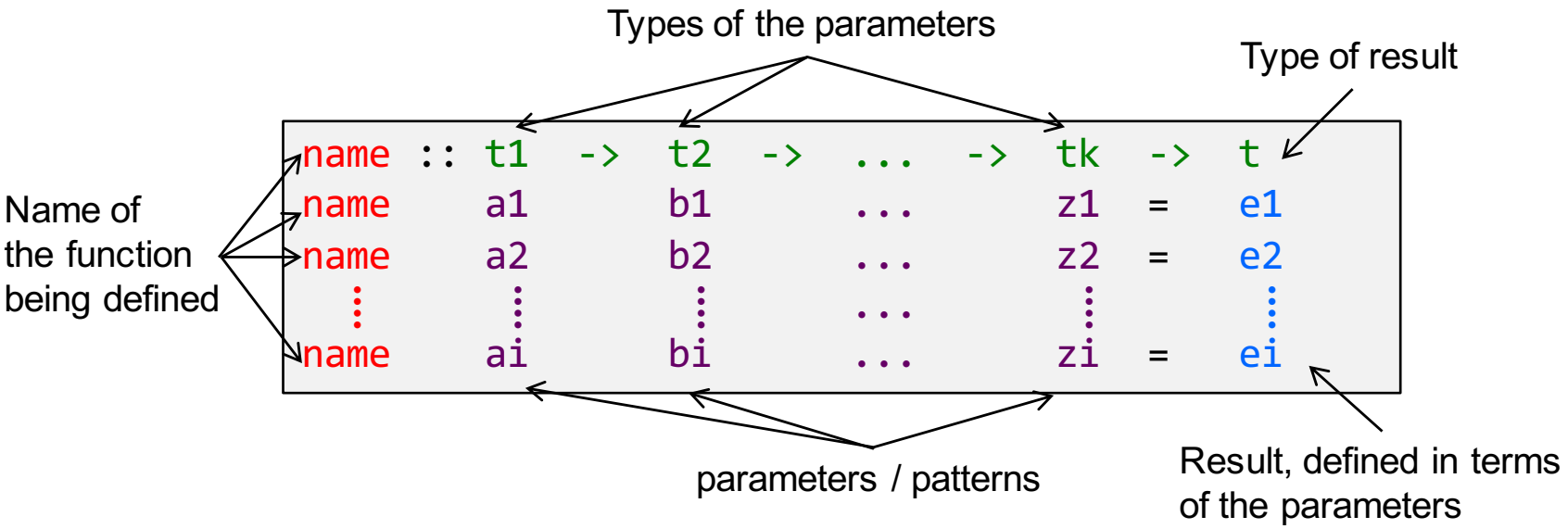
# Definitions

- **A definition associates a name with a value of a particular type**

```
name :: type
name =  expression
```

- **The name is used for identification.**
  - The `name` is needed to associate the `expression` in the second line with the `type` in the first line.
  - But **more important**: later on you can refer to the function by using its name
- **The type describes how to interact with a function**
  - The `type` determines what inputs a function expects, and what it will return as a result
- **The expression defines what the function does**
  - Note that there may be more than one `expression` for the same `name`!
  - All expressions with the same name make up the function definition!

# Function Definition even more General




## Example

```
sayNumber :: Int -> String -> String
sayNumber 0 s = "No " ++ s
sayNumber 1 s = "One " ++ s
sayNumber 2 s = "Another " ++ s
sayNumber n s = "Many" ++ s ++ "s"
```

# Pattern Matching

- All expressions together define the computation rule of the function
- Left hand side of "=" is used to match the actual parameters when calling a function and to determine which part of the functions computation rule shall be applied
- Patterns are tried from top to bottom
  - If first equation does not match, then try second equation, etc.
  - *Therefore: Place most specialized patterns first and then define more general patterns*
- Patterns may include
  - constants like 0 or []
  - names like n
  - wildcard '_' (matches always but binds no name to the matched value)
  - structures like lists (x:xs) or tuples (a,b)
  - more things you'll see as we proceed

# Worksheet: Pattern Matching

## Key learnings:

# Expressions

**Now let's have a look at the syntax we can use to define a function**

- **Case Expression**
- **Guards**
- **Conditional Expression (if then else)**
- **Let Expressions (let in)**
- **Where Bindings**

```
name :: type
name =  expression
```

# Case expressions

- **The general form for pattern matching is the case expression**

```
case expression of
  pattern -> result
  pattern -> result
  ...
```

- **Case expressions let us use pattern matching anywhere:**

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of
                                      []  -> "empty."
                                      [x] -> "a singleton list."
                                      xs  -> "a longer list."
```

- **Actually these two pieces of code do the same thing:**

```
head :: [a] -> a
head [] = error "No head"
head (x:_) = x
```

```
head :: [a] -> a
head xs = case xs of
            []    -> error "No head"
            (x:_) -> x
```

# Guards

- Pattern Matching cannot be used for complex conditions
  - We cannot write:

```
abs :: (Num a, Ord a) => a -> a
abs n < 0  = -n
abs n >= 0 = n
```

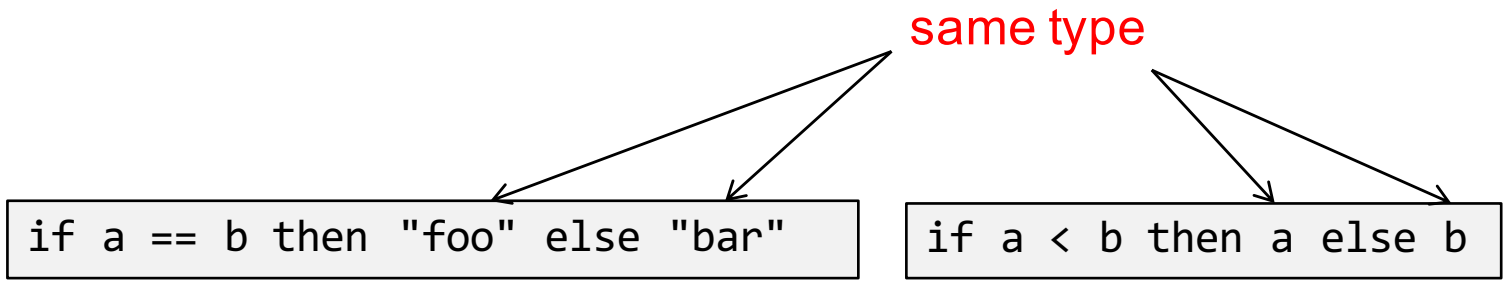
- We need **guards** to express this:

```
abs :: (Num a, Ord a) => a -> a
abs n
  | n < 0     = -n
  | otherwise = n
```

```
abs :: (Num a, Ord a) => a -> a
abs n = case n of
          m | m < 0     -> -m
            | otherwise -> m
```

- Note: the alternative conditions are preferably **indented** and **aligned** below each other!

# Conditional Expressions

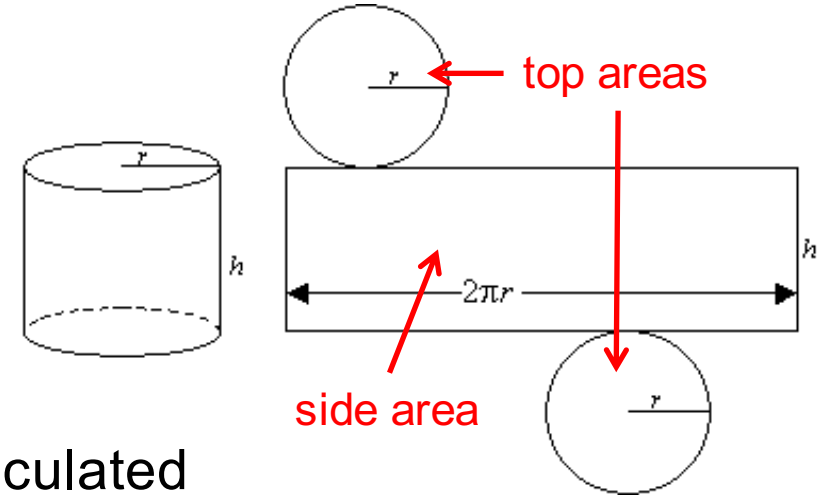



- Conditional expressions **must** evaluate to a value
- The else branch must be present, it cannot be omitted
- Both branches of conditional expression must be of the same type!
- Use indentation to increase readability

```
if a == b
    then "Eq"
    else "Not Eq"
```

# Worksheet: Conditional Branching

# Increase Readability




- Names can increase readability.
  Eg. The surface area of a cylinder is calculated by adding the two top areas to the side area.
- Given the radius r and the height h of the cylinder, its surface area can be computed with the following Haskell expression:

```
cylinder :: Float -> Float -> Float
cylinder r h = 2 * pi * r * h + 2 * pi * r ^ 2
```

- Using bindings increases readability:

```
cylinder :: Float -> Float -> Float
cylinder r h = 2 * topArea + sideArea
    where sideArea = 2 * pi * r * h
          topArea  = pi * r ^ 2
```

bindings

This form reflects the description above better than the one-liner!

# Where bindings

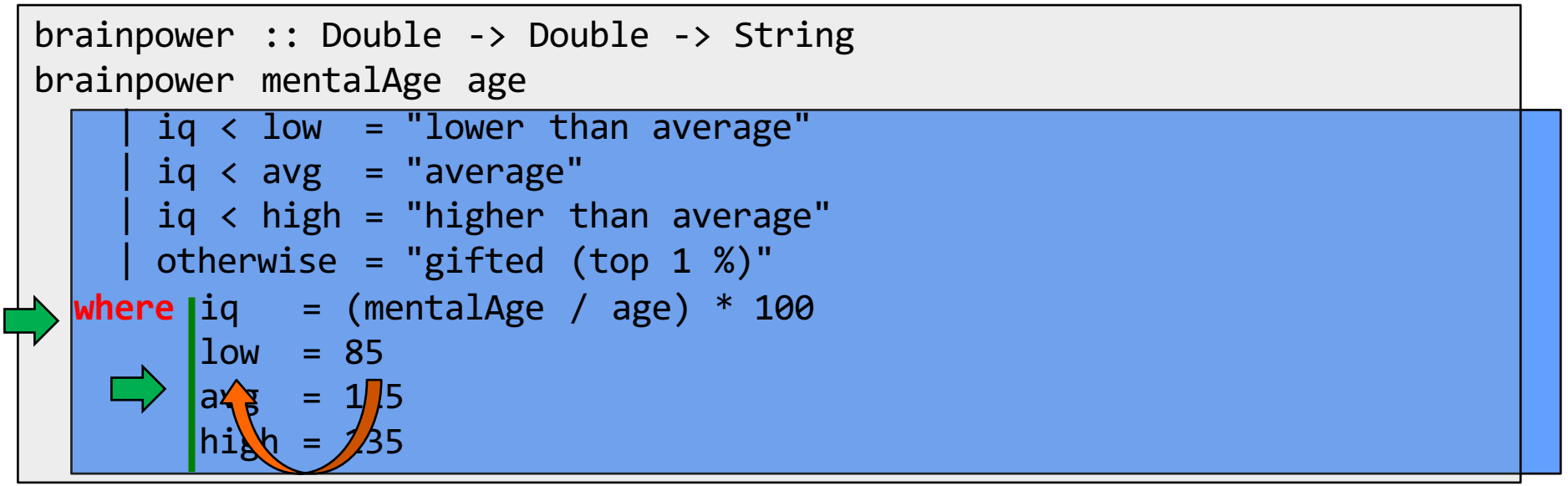
**Guard expressions tend to be unreadable as parts of the condition need to be repeated in each guard.**

```
brainpower :: Double -> Double -> String
brainpower mentalAge age
    | (mentalAge / age) * 100 < 85  = "lower than average"
    | (mentalAge / age) * 100 < 115 = "average"
    | (mentalAge / age) * 100 < 135 = "higher than average"
    | otherwise                     = "gifted (top 1 %)"
```

**This can be improved with a where binding:**

```
brainpower :: Double -> Double -> String
brainpower mentalAge age
    | iq < 85   = "lower than average"
    | iq < 115  = "average"
    | iq < 135  = "higher than average"
    | otherwise = "gifted (top 1 %)"
  where iq = (mentalAge / age) * 100
```

# Where bindings




- A binding calculates a value and binds it to a name that can be used elsewhere
- The bindings in a where clause are visible in the same function clause as the where clause is placed
- Use indentation to make clear to where the bindings belong

# Where bindings

- Where clauses can contain local function definitions

```
shoutName f l = shout f ++ " " ++ shout l
  where shout s = map toUpper s
```

- Pattern matching can be applied in where clauses!

```
...
  where iq = (mentalAge / age) * 100
        (low, avg, high) = (85, 115, 135)
```

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
  where (f:_) = firstname
        (l:_) = lastname
```

- The left hand side of a binding can be a pattern.
  - pattern = expression

# Let Expressions

- Where bindings are visible within the whole function clause. If the visibility (the scope) of a binding should be narrower, use let

```
cylinder :: Float -> Float -> Float
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r ^2
    in  2 * topArea + sideArea
```

- The bindings are only visible in the **in** block of the expression.
- 

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

  - This example does not make sense as a one-liner. But when the in-block grows to several lines it helps to clarify the semantics of the code.

# Let Expressions vs Where Bindings

- Where bindings are visible within the whole function clause. If the visibility (the scope) of a binding should be narrower, use let

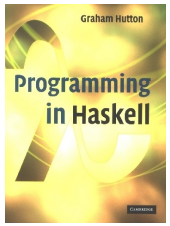
```
doSomething :: [Int] -> [Int] -> Int
doSomething xs ys =  if x < y then
                        let (_:sx:_) = xs; (_:sy:_) = ys
                        in  sx + sy
                     else
                        x + y
                     where
                        (x:_) = xs; (y:_) = ys
```

- Let bindings are expressions themselves, whereas where bindings can only be used as part of function declarations.
- Let bindings can be nested, where bindings cannot.

```
letNesting a = let b = 5 in
                   let b = 6 in
                      a + b
```

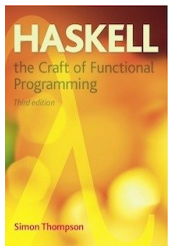
# Worksheet: Bindings

# Further Reading




Chapter 4.1 – 4.4

Page 20




Chapter 3.4, 6.3

Pages 97 – 99, 109 – 111, 123 – 128


Chapter 4