

# Basics

## Functional Programming



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# Goals

- ▶ Function definitions
  - ▶ Local definitions
  - ▶ Guards and pattern matching
- ▶ Working with tuples and lists
- ▶ Layout and comments
- ▶ Notions about types
  - ▶ What is **polymorphism**?

Chapters 4 (up to 4.4) and 3 from Hutton's book



# Simple functions

From the previous lecture...

```
average ns = sum ns `div` length ns
```

- ▶ Function `average` and argument `ns` are in **lowercase**
- ▶ This line defines an **equation**
- ▶ Calling a function is done **without parentheses**
  - ▶ `div` is used as an **operator**



# Basic list functions

- ▶ `null` tells whether a list is empty
- ▶ `head` returns the first element in a list
- ▶ `tail` returns all but the first element

```
> null [1,2,3]
```

```
False
```

```
> head [1,2,3]
```

```
1
```

```
> tail [1,2,3]
```

```
[2,3]
```



# Basic list functions

- ▶ `null` tells whether a list is empty
- ▶ `head` returns the first element in a list
  - ▶ `head` fails if the list is empty
- ▶ `tail` returns all but the first element
  - ▶ `tail` fails if the list is empty

```
> null [1,2,3]
```

```
False
```

```
> head [1,2,3]
```

```
1
```

```
> head []
```

```
*** Exception: Prelude.head: empty list
```

```
> tail [1,2,3]
```

```
[2,3]
```



# List constructors

- ▶ `[]` is the empty list
- ▶ `x : xs` puts element `x` in front of the list `xs`
  - > `1 : []`  
`[1]`
  - > `1 : [2,3]`  
`[1,2,3]`
- ▶ In fact, `[1,2,3]` is **sugar** for `1 : (2 : (3 : []))`



# Types of the basic list functions

- ▶ What are the types of those functions?



# Types of the basic list functions

- What are the types of those functions?

Here is the first one: `null` checks if a list is empty

```
null :: [a] -> Bool
```

What about `head`, `tail`, `[]`, and `(:)`?





# Types of the basic list functions

- What are the types of those functions?

Here is the first one: `null` checks if a list is empty

```
null :: [a] -> Bool
```

What about `head`, `tail`, `[]`, and `(:)`?

```
head :: [a] -> a
```

```
tail :: [a] -> [a]
```

```
[] :: [a]
```

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



# Conditionals

if condition then expression else expression

```
abs n = if n < 0 then -n else n
```

```
firstordefault def list
```

```
=
```



# Conditionals

if condition then expression else expression

```
abs n = if n < 0 then -n else n
```

```
firstordefault def list
```

```
    = if null list then def else head list
```

- ▶ condition must be a Bool expression
- ▶ You always need **both** branches
  - ▶ What would you return if one is missing?
  - ▶ Remember, **everything is an expression**



# Layout rule

- ▶ Haskell does not have other delimiters but parentheses
  - ▶ Not completely true, but valid for human-produced code
  - ▶ The grouping is done by indentation
- ▶ The **layout rule** applies for indentation
  - ▶ Related elements must start on the same column
  - ▶ In the case of conditionals, no requirements

```
abs n = if n < 0
      then -n
      else n
```

```
abs n = if n < 0
      then -n
      else n
```



# Guards

Instead of conditionals, we use equations with **guards**

- ▶ Each guard defines a condition over the arguments
- ▶ These conditions are checked in order
  - ▶ The first satisfiable one is applied
- ▶ We typically use `otherwise` for the default case

```
abs n | n < 0      = -n  
      | otherwise = n
```



# Nested conditionals versus guards

```
sign n = if n < 0
         then -1
         else if n == 0
              then 0
              else 1
```

What does this function do?



# Nested conditionals versus guards

```
sign n = if n < 0
         then -1
         else if n == 0
              then 0
              else 1
```

What does this function do?

It reads much better with guards!

```
sign n | n < 0      = -1
      | n == 0     = 0
      | otherwise  = 1
-- Why not | n > 0 = 1 ?
```



# Nested conditionals versus guards

Good style

Prefer guards over conditionals





# Local definitions

```
distance px py qx qy =  
    sqrt ((px - qx)*(px - qx) + (py - qy)*(py - qy))
```

expression where name = expression

```
distance px py qx qy = sqrt (xDiff + yDiff)  
    where  
        xDiff      = square (px - qx)  
        yDiff      = square (py - qy)  
        square z   = z * z
```



# Local definitions

```
distance px py qx qy =  
    sqrt ((px - qx)*(px - qx) + (py - qy)*(py - qy))
```

**let name = expression in expression**

```
distance px py qx qy =  
    let xDiff    = square (px - qx)  
        yDiff    = square (py - qy)  
        square z = z * z  
    in sqrt (xDiff + yDiff)
```



# Local definitions

expression where name = expression

let name = expression in expression

- ▶ Local definitions assign a **name** to an expression
  - ▶ In the larger expression, this name is available
- ▶ Multiple benefits
  - ▶ Maintainability: reduce repetition of code
  - ▶ Performance: the expression is only computed once
  - ▶ Documentation: assign names to concepts



# Local definitions

- ▶ You can have more than one local definition
  - ▶ Definitions may refer to each other
- ▶ The **layout rule** kicks in
  - ▶ All definition must start in the same column
  - ▶ Aligning =’s is not mandated, but good style



# Let vs Where

- ▶ where when thinking top down
- ▶ let when thinking bottom up



# Let vs Where

- ▶ where when thinking top down
- ▶ let when thinking bottom up
- ▶ let is an expression; where is not.

```
foo x = show (let y = x*x in y*y) ++ " someString"
```

```
bar x | f x < 5    = undefined
      | f x == 5   = undefined
      | otherwise = undefined
```

where

```
  f y = undefined
```



# Tuples

- ▶ **Lists** are sequences of elements of the same type
  - ▶ Unknown length, uniform type  
`[True, False] :: [Bool]`
- ▶ **Tuples** are made of a number of components
  - ▶ Known length, different types  
`(True, 'a') :: (Bool, Char)`  
`(1, 'b', 3) :: (Int, Char, Int)`
  - ▶ Useful for returning several values



# Tuple Examples

Creating tuples:

```
trunc    :: Double -> (Int, Double)
trunc x = let i  = floor x
           in (i, x - fromIntegral i)
```





# Tuple Examples

Creating tuples:

```
trunc    :: Double -> (Int,Double)
trunc x = let i  = floor x
           in (i, x - fromIntegral i)
```

Extracting from tuples:

```
distance (px, py) (qx,qy) = sqrt (xDiff + yDiff)
  where
    tpl = squareBoth (px - qx, py - qy)
    squareBoth (xD,yD) = (xD*xD, yD*yD)

    xDiff = fst tpl
    yDiff = snd tpl
```



# Tuple Examples

Creating tuples:

```
remainder    :: Double -> (Int,Double)
remainder x = let i  = floor x
               in (i, x - fromIntegral i)
```

Extracting from tuples:

```
distance (px, py) (qx,qy) = sqrt (xDiff + yDiff)
  where
    tpl = squareBoth (px - qx, py - qy)
    squareBoth (xD,yD) = (xD*xD, yD*yD)

    (xDiff, yDiff) = tpl
```



# Comments

```
-- Euclidean distance between two points
distance (px, py) (qx, qy) =
    sqrt (xDiff + yDiff) -- some comment
where
    {- multi
       line comments are also
       possible -}
```

- ▶ -- comments skip until the end of the line
- ▶ {- comments skip until its matching -}
  - ▶ Warning! These comments nest



# Pattern matching, fac

From the previous lecture...

```
fac 0 = 1
```

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

- ▶ The first equation is chosen if the arguments is 0
- ▶ Otherwise, the second branch is executed
- ▶ This is an example of **pattern matching**



# Pattern matching, replicate

- ▶ For a call `replicate n x`,
  - ▶ If `n` is 0, we return an empty list
  - ▶ Otherwise, we attach a copy of `x` to the result of replicating the element `n-1` times



# Pattern matching, replicate

- ▶ For a call `replicate n x`,
  - ▶ If `n` is 0, we return an empty list
  - ▶ Otherwise, we attach a copy of `x` to the result of replicating the element `n-1` times

```
replicate      :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : replicate (n-1) x
```



# Pattern matching, replicate

- ▶ For a call `replicate n x`,
  - ▶ If `n` is 0, we return an empty list
  - ▶ Otherwise, we attach a copy of `x` to the result of replicating the element `n-1` times

```
replicate      :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

- ▶ **Good style:** use `_` if you don't care about a value



# Pattern matching for lists and tuples

- ▶ The syntax for construction can be used for matching
- ▶ Information is extracted by giving **names** to the parts
  - ▶ As usual, starting with lowercase

```
null [] = True  
null _  = False
```

```
length []      = 0  
length (_ : xs) = 1 + length xs
```

```
squareBoth (xD,yD) = (xD*xD, yD*yD)
```





# Pattern matching, conjunction

- For Booleans, we can list all the possible values

```
conj :: Bool -> Bool -> Bool
```

```
conj True  True  = True
```

```
conj True  False = False
```

```
conj False True  = False
```

```
conj False False = False
```



# Pattern matching, conjunction

- ▶ For Booleans, we can list all the possible values

```
conj :: Bool -> Bool -> Bool
```

```
conj True  True  = True
```

```
conj True  False = False
```

```
conj False True  = False
```

```
conj False False = False
```

- ▶ But this is very repetitive!

- ▶ All last three equations return False

```
conj True  True  = True
```

```
conj a      b      = False
```

- ▶ even better, use `_` instead of `a` and `b`



# Nested patterns

- ▶ Instead of just giving a name, you can further pattern match in a list or tuple
  - ▶ You can go as deep as you want

```
trimstart (' ' : xs) = trimstart xs
trimstart ('\t' : xs) = trimstart xs
trimstart xs          = xs
```

```
iszero (0, 0) = True
iszero _      = False
```

```
sumifthree (a : b : c : []) = a + b + c
sumifthree _                  = 0
```



# Pattern matching versus guards with ==

```
length xs | xs == []    = 0  
          | otherwise = 1 + length (tail xs)
```

Two problems with this definition:



# Pattern matching versus guards with ==

```
length xs | xs == [] = 0  
          | otherwise = 1 + length (tail xs)
```

Two problems with this definition:

- ▶ == is more expensive than matching
- ▶ You need to call `tail`

Good style for defining a function

- ▶ Pattern matching, maybe with guards
  - ▶ But **not** guards with ==



# Pattern matching versus guards with ==

```
length xs | xs == []    = 0  
          | otherwise = 1 + length (tail xs)
```

The correct way to write `length` is:

```
length []          = 0  
length (_ : xs) = 1 + length xs
```

- ▶ Substitute check of `[]` by pattern matching
- ▶ Access the tail of the list by matching `(_ : xs)`



## Exercise: define the `existsPositive` function

`existsPositive xs` should return `True` if and only if (at least) one of the elements in the list `xs` is positive, that is, greater than 0



## Exercise: define the existsPositive function

`existsPositive xs` should return `True` if and only if (at least) one of the elements in the list `xs` is positive, that is, greater than 0

```
existsPositive []                = False
existsPositive (x:xs) | x > 0    = True
                       | otherwise = existsPositive xs
```





## Exercise: define the existsPositive function

`existsPositive xs` should return `True` if and only if (at least) one of the elements in the list `xs` is positive, that is, greater than 0

```
existsPositive []      = False
existsPositive (x:xs) = x > 0 || existsPositive xs
```

Next lecture is devoted to functions over lists



# Operators

From the previous lecture...

- ▶ Operators are functions whose name is **exclusively** made out of **symbols**
- ▶ Operators are written **between** the arguments
  - ▶ Both for definition and call

```
True && True = True  
_    && _    = False
```

- ▶ Anywhere else, you need to use parentheses

```
(&&) :: Bool -> Bool -> Bool
```



# Associativity and precedence

How should we read the following expressions?

$$1 + 2 - 3$$

$$1 * 2 + 3 / 4$$

We make it explicit by introducing parentheses

$$1 + (2 - 3)$$

$$(1 * 2) + (3 / 4)$$

- ▶ We say that + **associates to the right**
  - ▶ So  $1 + 2 + 3$  means  $1 + (2 + 3)$
- ▶ We say that \* and / have **higher precedence** than +



# Declaring associativity and precedence

## `infixr/infixl/infix` precedence operator

- ▶ `infixr` and `infixl` declare associativity
- ▶ `infix` makes the operator **non**-associative
  - ▶ `==` and `/=` are examples of those
- ▶ Precedence ranges between 1 and 9
  - ▶ Function application has the highest number, 10

```
infixr 3 &&
```



# Types



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# Expressions have types

**Type** = collection of related values

- ▶ In Haskell, every **expression** has a **type**
- ▶ We write it as `expression :: type`

```
True      :: Bool
'a'       :: Char
[1, 2]    :: [Int]
(1, 'a')  :: (Int, Char)
not       :: Bool -> Bool
```

- ▶ This includes applied functions

```
1 + 2     :: Int
not True  :: Bool
```



# Static typing and type safety

- ▶ Haskell forbids executing code with type errors
  - ▶ This is known as **static** typing
  - ▶ Other languages are **dynamically** typed
    - ▶ E.g., Python, JavaScript, Ruby...
- ▶ As a result, no run-time error may arise from this
  - ▶ We say that Haskell programs are **type safe**
- ▶ Some “valid” expressions are rejected
  - ▶ Code execution is not taken into account

```
if True then 1 else False
```



# Type checking and inference

**General rule:** if  $f :: A \rightarrow B$  and  $e :: A$ , then  $f\ e :: B$

This rule can be used in two ways:

- ▶ To **check** whether an application is correct

```
not :: Bool -> Bool
'a'  :: Char
not 'a'
-- Couldn't match expected type 'Bool'
--           with actual type 'Char'
```

- ▶ To **infer** the result of an expression

```
f :: Bool -> String
f True :: String -- No further details needed!
```





# Basic types

- ▶ `Bool`: logical values, that is, either `True` or `False`
- ▶ `Char`: single characters like `'a'`
- ▶ Integral types:
  - ▶ `Int`: machine integers with a fixed range  
> `maxBound :: Int`  
`9223372036854775807`
  - ▶ `Integer`: integers with unlimited range
- ▶ Floating-point types:
  - ▶ Numbers with a decimal comma
  - ▶ `Float`: single-precision
  - ▶ `Double`: double-precision, take up more space



# Compound types

These types are parametrized by other types

- ▶ Lists  $[T]$ , uniform sequences of  $T$ s
- ▶ Tuples come in different **arities**
  - ▶ Pairs  $(T_1, T_2)$
  - ▶ Triples  $(T_1, T_2, T_3)$
  - ▶ ... up to 62 in GHC 8.0.1
- ▶ Functions  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow R$

Types can be nested as much as we want



# Some differences

`([1, 2], [True])`

`[(1, True), (2, False)]`



# Some differences

```
                                -- ↓ Tuple of lists
([1, 2], [True])               :: ([Int], [Bool])
                                -- ↓ List of tuples
[(1, True), (2, False)]       :: [(Int, Bool)]
```



# Some differences

```
-- ↓ Tuple of lists
([1, 2], [True])      :: ([Int], [Bool])
-- ↓ List of tuples
[(1, True), (2, False)] :: [(Int, Bool)]

f :: (Int, Int) -> Int

g :: Int -> Int -> Int
```



# Some differences

```
-- ↓ Tuple of lists
([1, 2], [True])      :: ([Int], [Bool])
-- ↓ List of tuples
[(1, True), (2, False)] :: [(Int, Bool)]

f :: (Int, Int) -> Int  -- Takes one argument
                        -- which is a pair
g :: Int -> Int -> Int  -- Takes two arguments

> f (1, 2)  -- OK
> g 1 2     -- OK
> g (1, 2)
-- Couldn't match expected type 'Int'
--           with actual type '(Int, Int)'
```



# Some differences

```
-- ↓ Tuple of lists
([1, 2], [True])      :: ([Int], [Bool])
-- ↓ List of tuples
[(1, True), (2, False)] :: [(Int, Bool)]

f :: (Int, Int) -> Int  -- Takes one argument
                        -- which is a pair
g :: Int -> Int -> Int  -- Takes two arguments

> f (1, 2)  -- OK
> g 1 2     -- OK
> g (1, 2)
-- Couldn't match expected type 'Int'
--           with actual type '(Int, Int)'
```



# Functions are first-class citizens

*-- Functions can be put in a list*

```
[(+), (*), (-)] :: [Int -> Int -> Int]
```

```
[(&&), (||)]    :: [Bool -> Bool -> Bool]
```

*-- Elements must agree in their type*

```
[(+), (&&)]      -- Type error!
```

*-- Functions can be arguments and results*

*-- 'flip' takes one function and swaps the order*

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





# length is polymorphic

```
length [1, 2, 3]      -- OK
length [True, False] -- OK
length "abcd"         -- OK
```

- ▶ `length` can be applied to any expression which is a list
  - ▶ In type terms, to any  $[T]$ , regardless of  $T$
  - ▶ We say that `length` is **polymorphic**
    - ▶ From Greek, Πολυμορφισμός “of many forms/shapes”
- ▶ How does this show up in the type?

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

- ▶ Types starting with lowercase are **variables**
- ▶ They can be substituted with whatever we need



# Other polymorphic list functions

```
null      :: [a] -> Bool
(++ )     :: [a] -> [a] -> [a]   -- Concatenation
reverse   :: [a] -> [a]
```

**Important!** A variable has to be substituted **uniformly** throughout the whole type

```
[1, 2] ++ [3, 4] :: [Int]
-- OK, 'a' is substituted by 'Int'
```

```
[1, 2] ++ [True, False]
-- Couldn't match expected type 'Int'
--           with actual type 'Bool'
```

This is the **#1 type error** in Haskell programming



# Build your own polymorphic function

```
id x = x
```

What is the type of `id`?



# Build your own polymorphic function

`id x = x`

What is the type of `id`?

1. It is a function with one argument
  - ▶  $\alpha \rightarrow \beta$  for yet unknown  $\alpha$  and  $\beta$
2. We return the same type we are given
  - ▶  $\alpha \rightarrow \alpha$  for a yet unknown type  $\alpha$
3. There are no further constraints for `x`
  - ▶ We reach the final type  $a \rightarrow a$
  - ▶ This function works for **any** type



# Inferring the type of `id id`

Expect these kind of problems in the exam

`id id :: ?`



# Inferring the type of `id id`

Expect these kind of problems in the exam

`id id :: ?`

1. Disambiguate the names of variables for each `id`
  - ▶ First `id ::  $\alpha \rightarrow \alpha$`
  - ▶ Second `id ::  $\beta \rightarrow \beta$`
2. If `f :: A -> B`, in `f e` we must have `e :: A`
  - ▶ In this case,  $\alpha$  must be  $\beta \rightarrow \beta$
  - ▶ Thus, first `id ::  $(\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)$`
3. The result type of `f e` is B
  - ▶ In this case, `id id ::  $\beta \rightarrow \beta$`
4. Finally, replace by variables types without constraints
  - ▶ `id id :: a -> a`



# Elements in a list have to match

```
> :t sin
sin :: Float -> Float
> :t [sin, id]
[sin,id] :: [Float -> Float]
```

1. We can choose any type for the `a` in `id`
2. All elements in a list must have the same type
3. The only solution is to make `a` be `Float`



# Elements in a list have to match

What about these?

```
> :t [length, head]
> :t [head, null]
> :t [tail, null]
```





# Elements in a list have to match

What about these?

```
> :t [length, head]
```

```
> :t [head, null]
```

```
> :t [tail, null]
```

```
> :t [length, head]
```

```
[length,head] :: [[Int] -> Int]
```

```
> :t [head, null]
```

```
[head,null] :: [[Bool] -> Bool]
```

```
> :t [tail, null]
```

```
Couldn't match type '[a]' with 'Bool'
```



# Overloaded addition

In Haskell, addition works for different types:

```
> 1 + 2  -- Integers
```

```
3
```

```
> 1.0 + 2.5  -- Floating-point
```

```
3.5
```

But not for any type!

```
> 'a' + 'b'
```

No instance for (Num Char)  
arising from a use of ‘+’



# Overloaded addition

Addition cannot be given the following type

$(+) :: a \rightarrow a \rightarrow a$

because it does not work for **any** type.



# Overloaded addition

Addition cannot be given the following type

```
(+) :: a -> a -> a
```

because it does not work for **any** type.

Let's ask GHC what is its real type:

```
> :t (+)
```

```
(+) :: Num a => a -> a -> a
```

- ▶ The `Num a` before the `=>` symbol is a **constraint**
- ▶ It restricts `(+)` to types which satisfy the constraint
  - ▶ In this case `a` must be “numeric”
- ▶ `Num` is called a **type class**
  - ▶ **Warning!** Not to be confused with C++/C#/Java classes



# Basic type classes

- ▶ `Num` for numeric types
  - ▶ Includes `(+)`, `(*)`, `abs`, among others
- ▶ For example, `Int`, `Integer`, `Float`, and `Double` have `Num` instances.
- ▶ `Char` or `[Int]` are not numeric



# Basic type classes

- ▶ Num for numeric types
- ▶ Eq for types which support equality checks

`(==) :: Eq a => a -> a -> Bool`    *-- Equals*

`(/=) :: Eq a => a -> a -> Bool`    *-- Not equals*

- ▶ Int, Char, Bool, ..., have Eq instances
- ▶ Also [T] if T is itself a member of Eq
  - ▶ Like [Int] or String
- ▶ But not function types

```
> sin == cos
```

```
No instance for (Eq (Float -> Float))
```



# Basic type classes

- ▶ `Num` for numeric types
- ▶ `Eq` for types which support equality checks
- ▶ `Ord` for types which in addition have an ordering

`(<)`, `(>)` `:: Ord a => a -> a -> Bool`

`(<=)`, `(>=)` `:: Ord a => a -> a -> Bool`

`min`, `max` `:: Ord a => a -> a -> a`

- ▶ `Int`, `Char`, `Bool`, .., have `Ord` instances
- ▶ Every type which is `Ord` is also `Eq`



# Basic type classes

- ▶ Num for numeric types
- ▶ Eq for types which support equality checks
- ▶ Ord for types which in addition have an ordering
- ▶ Show for turning things into strings

```
show :: Show a => a -> String
```

```
age :: Int -> String
```

```
age y = "You are " ++ show y ++ " years old"
```

- ▶ Almost everything is in Show, but not functions
- ▶ We need an explicit call to show to preserve type safety





# Basic type classes

- ▶ `Num` for numeric types
- ▶ `Eq` for types which support equality checks
- ▶ `Ord` for types which in addition have an ordering
- ▶ `Show` for turning things into strings
- ▶ And many more!

You can also define your own (later in the course)



# Parse errors are not type errors

```
> isZero x = x = 0  
<interactive>:1:14: error:  
    parse error on input '='
```

**Parse error** = code does not follow the **syntax**

- ▶ The structure of the code cannot be understood
  - ▶ In this case, where does the real definition start?
- ▶ Parsing happens before typing
- ▶ Check the shape and the upper/lowercase distinction



# Parse errors are not type errors

```
> isZero x = x = 0
<interactive>:1:14: error:
    parse error on input '='
```

**Parse error** = code does not follow the **syntax**

- ▶ The structure of the code cannot be understood
  - ▶ In this case, where does the real definition start?
- ▶ Parsing happens before typing
- ▶ Check the shape and the upper/lowercase distinction

```
> isZero x = x == 0
```



# Important concepts

- ▶ Every expression has a type
- ▶ Types are used in two different ways
  - ▶ **Checking** that types match
  - ▶ **Inferring** a type for an expression
- ▶ Two forms of **polymorphism**
  - ▶ Functions that work for any type, **parametric**
  - ▶ Functions that work for a subset of types, **ad-hoc**

Check exercises at the end of chapter 3 of Hutton's book

