Session 2

Haskell Types

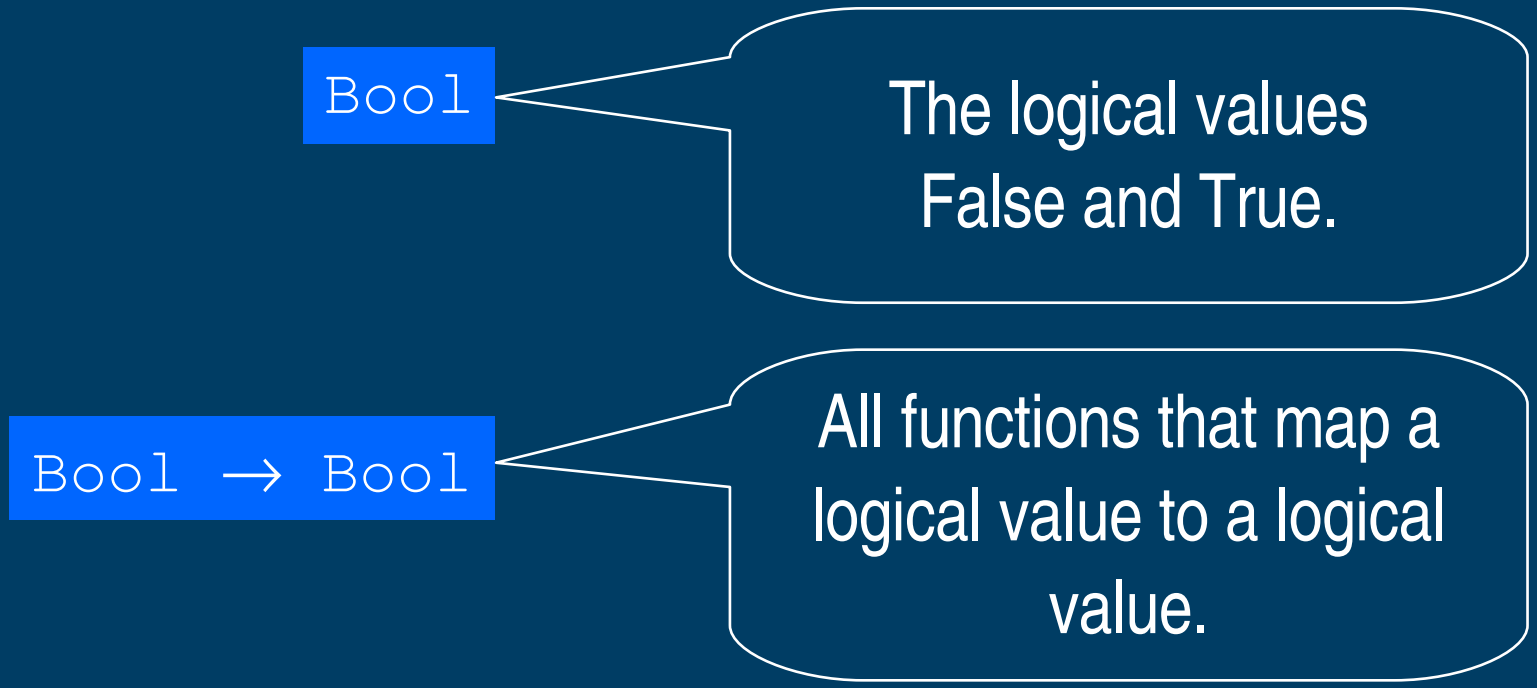
Defining Functions

Currying

Universiteit Antwerpen

What is a Type?

A type is a collection of related values.



Types in Haskell

We use the notation e :: T to mean that evaluating the expression e will produce a value of type T.

```
False            :: Bool

not              :: Bool → Bool

not False        :: Bool

False && True    :: Bool
```

Note:

- Every expression must have a valid type, which is calculated prior to evaluating the expression by a process called type inference;
- Haskell programs are type safe, because type errors can never occur during evaluation;
- Type inference detects a very large class of programming errors, and is one of the most powerful and useful features of Haskell.

Basic Types

Haskell has a number of basic types, including:

- `Bool` - Logical values
- `Char` - Single characters
- `String` - Strings of characters
- `Int` - Fixed-precision integers
- `Integer` - Arbitrary-precision integers

Universiteit Antwerpen

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

In general:

[T] is the type of lists with elements of type T.

Universiteit Antwerpen

Note:

- The type of a list says nothing about its length:

```
[False,True]          :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Universiteit Antwerpen

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)       :: (Bool,Bool)

(False,'a',True)   :: (Bool,Char,Bool)
```

In general:

(T1,T2,…,Tn) is the type of n-tuples whose ith components have type Ti for any i in 1…n.

Note:

- The type of a tuple encodes its arity:

```
(False,True)          :: (Bool,Bool)

(False,True,False)    :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b'))  :: (Char,(Bool,Char))

(True,['a','b'])   :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not      :: Bool → Bool

isDigit  :: Char → Bool
```

In general:

T1 → T2 is the type of functions that map arguments of type T1 to results of type T2.

Universiteit Antwerpen

Note:

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

Task: Implement without tuple

```
add          :: (Int,Int) → Int
add (x,y)     = x+y

zeroto       :: Int → [Int]
zeroto n      = [0..n]
```

Exercises

(1) What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

[isDigit,isLower,isUpper]
```

(2) Check your answers using Hugs.

Session 2

Haskell Types

Defining Functions

- Conditionals
- Guard Patterns
- Lambda Expressions

Currying

Universiteit Antwerpen

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Universiteit Antwerpen

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Universiteit Antwerpen

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

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

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Note:

- The catch all condition otherwise is defined in the prelude by otherwise = True.

Universiteit Antwerpen

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not          :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Universiteit Antwerpen

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)              :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

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

Universiteit Antwerpen

However, the following definition is more efficient, as it avoids evaluating the second argument if the first argument is False:

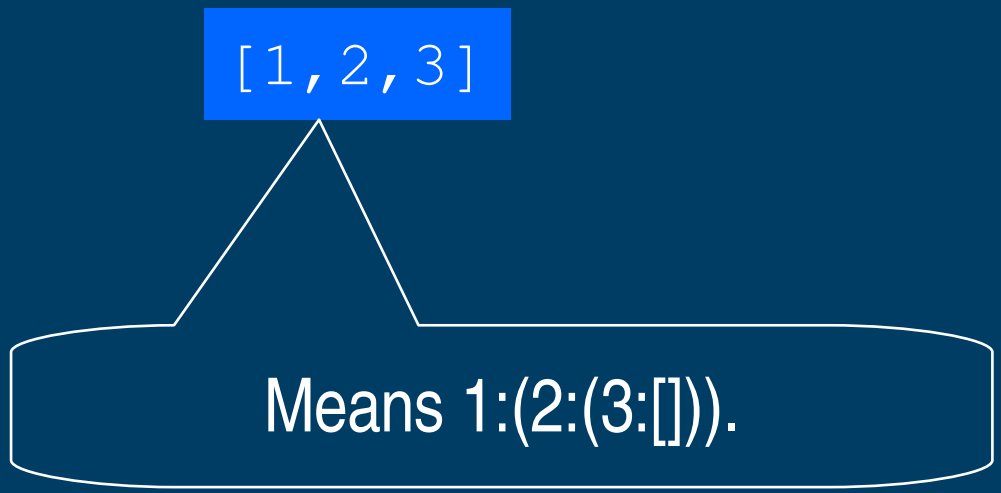
```
False &&  _ = False
True  &&  b = b
```

Note:

- The underscore symbol _ is the wildcard pattern that matches any argument value.

List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called “cons” that adds a new element to the start of a list.



The cons operator can also be used in patterns, in which case it destructs a non-empty list.

```
head        :: [a] → a
head (x:_)   = x

tail        :: [a] → [a]
tail (_:xs)  = xs
```

head and tail map any non-empty list to its first and remaining elements.

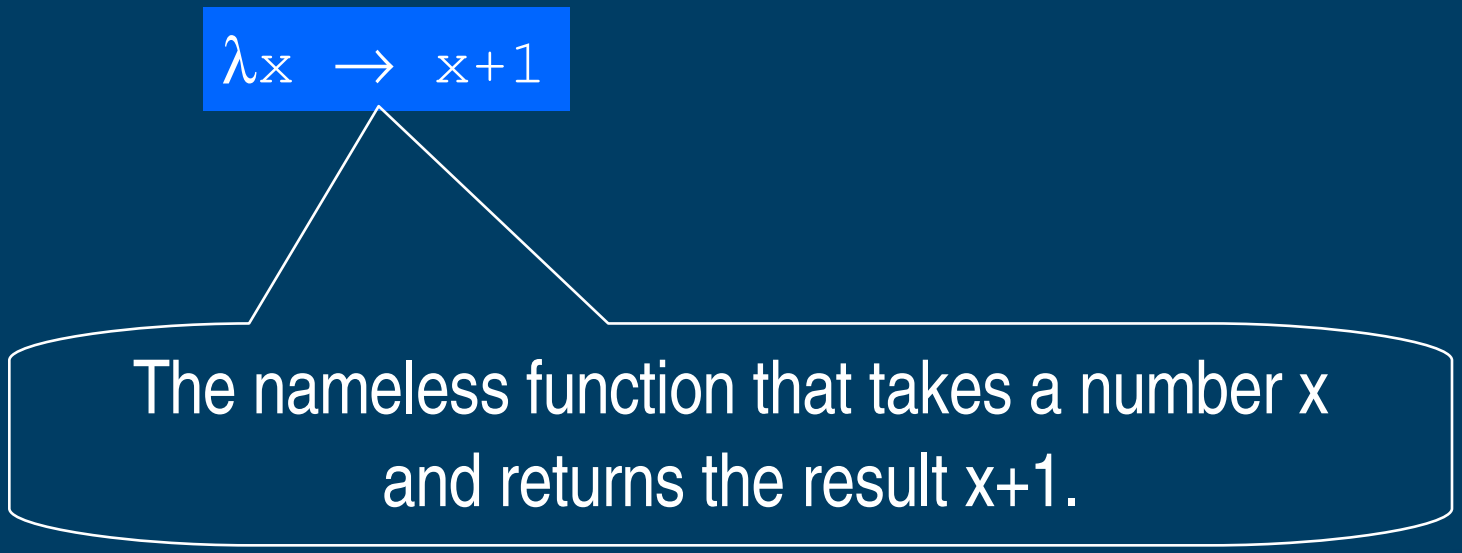
Summary: Pattern Expressiveness

- Constants
  - f::Integer->b
    f 1= …
- Lists
  - Empty
    - f::[a]->b
      f []= …
  - Non-Empty
    - f::[a]->b
      f (x:xs)= …
  - Exactly 1 Element
    - f::[a]->b
      f [x]= …

Universiteit Antwerpen

Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression.



Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x+y
```

means

```
add = λx → (λy → x+y)
```

After variable x is bound, it is used as a constant in the definition of the nested function...

Universiteit Antwerpen

Lambda expressions are also useful when defining functions that return functions as results.

For example,

```
compose f g x = f (g x)
```

is more naturally defined by

```
compose f g = λx → f (g x)
```

The lambda allows one to omit particular variables from the match pattern on the LHS.

Universiteit Antwerpen

Exercises

(1) Assuming that else branches were optional in conditional expressions, give an example of a nested conditional with ambiguous meaning.

(2) Give three possible definitions for the logical or operator || using pattern matching.

(3) Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(i) a conditional expression;
(ii) guarded equations;
(iii) pattern matching.

Hint:
The prelude function null :: [a] → Bool can be used to test if a list is empty.

Session 2

Haskell Types

Defining Functions

Currying

Universiteit Antwerpen

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

add' takes an integer x and returns a function. In turn, this function takes an integer y and returns the result x+y.

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

What function is returned by call to “add' 3”? Write it.

Universiteit Antwerpen

Note:

Wanneer een functie een functie van de vorm “add3” verwacht, kan je die aanmaken uit add' maar niet uit add.

- add and add’ produce the same final result, but add takes its two arguments at the same time, whereas add’ takes them one at a time:

1 parameter, onmiddellijk eindresultaat

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions.

Universiteit Antwerpen

- Functions with more than two arguments can be curried by returning nested functions:

```
mult       :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function, which in turn takes an integer y and returns a function, which finally takes an integer z and returns the result x*y*z.

Universiteit Antwerpen

Curry Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

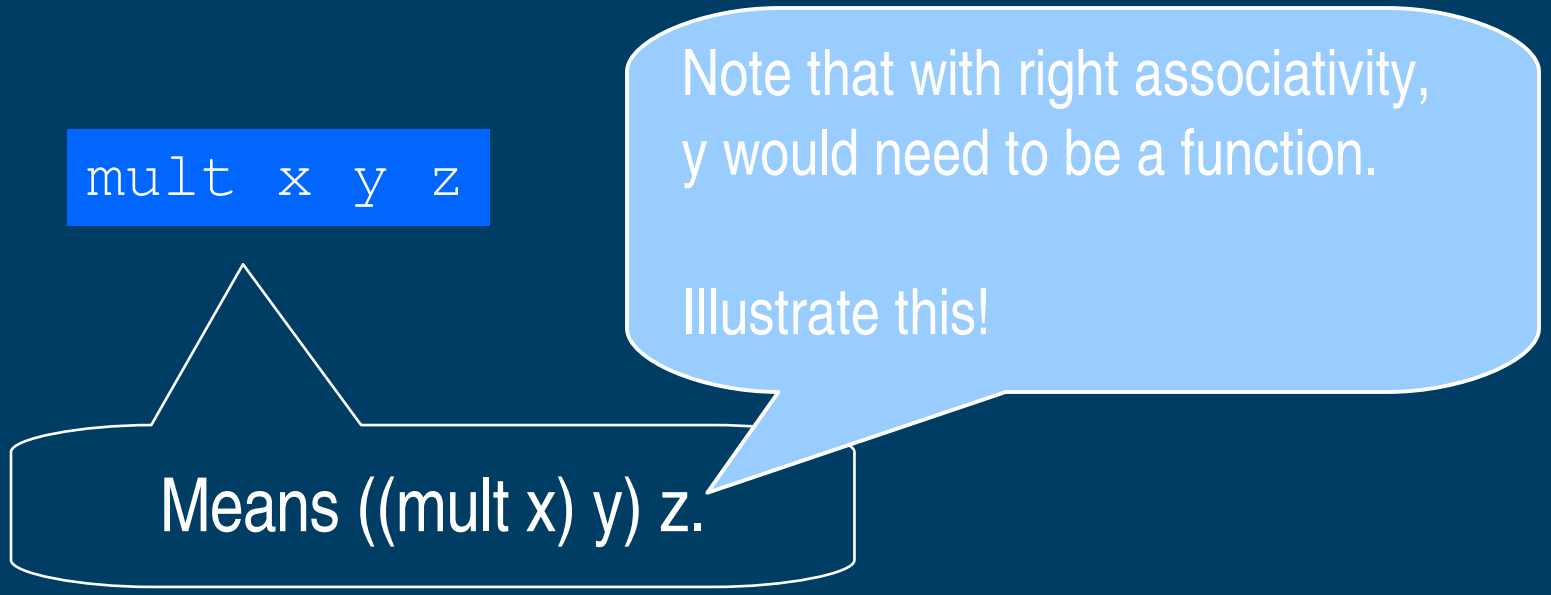
- The arrow $\rightarrow$ associates to the right.

```
Int → Int → Int → Int
```

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int)).

Universiteit Antwerpen

- As a consequence, it is then natural for *function application* to associate to the *left*.



Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

ApplyToAll

- Write a function that applies a unary function to all elements from a list

Why always Currying?

Motivation= Reuse.

Example: implement addition.

- Simple case: take two numbers, return one
- Special application: add a number to a each element of a list of numbers

Using ***tuples***: if you want (2) to reuse (1) then you need to make a function that converts functions on a tuple of a number and a list of those numbers to a repeated application of your basic function. This function is hence specific to converting couples.

Using ***currying***: you build every list application function in terms on unary functions. Since any function with more arguments can be transformed to unary functions by simple argument application, it stimulates reuse. Example:

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

The "map" function is written without knowing that you would want to write an add function for the two cases shown above (it is only written for unary functions and lists), still case (2) can be realized.

Les 4: H.O. Functions

Voorbeeld: ***add a->a->a*** kan je als volgt toepassen:

map (add 1) [2,3,4]
geeft: **[3,4,5]**

omdat **(add 1)** een functie teruggeeft waarbij de constante **1** wordt opgeteld bij variabele **y**.

Functie "**map**" past die unaire functie toe op lijst **[2,3,4]**.

Universiteit Antwerpen