Functional programming

technology that matters

What is it?

technology that matters

Functions?

Functions?

- Mapping
- Takes multiple arguments
- Produces one result
- Can be *named*
- Have a *body*

Function example

- Let's define a function that doubles its argument

$$\textit{double}\ x = x + x$$

- Named *double*
- Argument x
- Body $x + x$

Function application

double 3

= { apply double }

3 + 3

= { apply + }

6

Nested application

double (double 2)

= { apply inner double }

double (2 + 2)

= { apply + }

double 4

= { apply double }

4 + 4

= { apply + }

8

Does order matter?

Yes!

What is functional programming?

- A *style* of programming
- The basic *method of computation* is the *application of functions to arguments*
- Functions are *first-class citizens*

Contrast with OO

- Object-oriented programming is also a *style* of programming
- *Combine data and operations* on data in *objects*
- Data contained in object is called *state*
- The state of methods is *mutated* by calling methods

What is a functional programming language?

A functional programming language…

supports

and encourages

a functional style

Declarative programming

- Functional programming tends to be more *declarative*
- Say what you want
- Not how you want it

Imperative example

- Sum of numbers 1 to n

```
count := 0
total := 0
repeat
  count := count + 1
  total := total + count
until
  count = n
```

Declarative example

- Sum of numbers 1 to n

sum [1..n]

Haskell

Haskell is…

… a *pure* functional programming language

… a general-purpose language

… a static typed language

… widely used in academia

… used in industry

The Haskell Tool Stack

Glasgow Haskell Compiler

- The main implementation of Haskell
- Both an interpreter and compiler
- Multi-platform
- Noted for rich feature set and high performance

Where to download?

technology that matters

https://docs.haskellstack.org/en/stable/README/

Conventions for this course

- We use GHCi (the GHC interpreter)
- Files have the .hs-extension
- We provide type descriptions as much as possible (more on that later)

Demonstration

Basics

- Comments start with --
- Functions in scripts can span multiple lines
- Whitespace matters in multi-line functions!

Types and classes

Types in Haskell (1)

- In Haskell everything has a type
- Types can be *inferred*
- Types can (but don’t have to be) made explicit

Types in Haskell (2)

- We denote the type of something with the notation v :: T
- This means v has type T
- Functions have types too!
- f :: T -> U
- This means function f maps from something of type T to type U

Types in Haskell (3) – basic types

- Bool – logical values
- Char – single characters
- String – string of characters
- Int – fixed-precision integers
- Integer – arbitrary-precision integers
- Float – single-precision floating point numbers
- Double – double-precision floating point numbers

Types in Haskell (4) - Lists

- Sequence of elements of the same type
- A list is denoted by elements enclosed in [ ]
- The type of a list is denoted by another type in [ ]
- For example:

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
["Michiel", "Joris", "Liesbeth", "Ron"] :: [String]
```

- Lists can be infinite! (we get to that later)

Convenient prelude list functions (1)

- *head xs* – returns head of list xs
- *tail xs* – returns tail of list xs
- *[1, 2, 3] !! 1* - !! returns nth element of list (zero-based)
- *take n xs* – returns first n elements of list
- *drop n xs* – remove first n elements of list
- *length xs* – calculate length of list xs
- *sum xs* – calculate the sum of elements of list xs

Convenient prelude list functions (2)

- *product xs* – calculate the product of elements of list xs
- *[1, 2, 3] ++ [4, 5]* - ++ concatenates lists
- *reverse xs* – reverses list xs

Types in Haskell (5) - Tuples

- Finite sequence of components of possibly different types
- Components are comma-separated, enclosed by ( ), as are the types
- For example:

```
(False, True) :: (Bool, Bool)
(False, ‘a’, True) :: (Bool, Char, Bool)
(“Michiel”, False) :: (String, Bool)
```

Types in Haskell (6) - Functions

- Functions (obviously) have types as well
- We can provide a type for a function as follows:
```
add :: (Int, Int) -> Int
add (x,y) = x + y
```
- Another example
```
zeroto :: Int -> [Int]
zeroto n = [0..n]
```
- Note that functions don't have to be *total* functions for their provided types

Currying and partial application

- Functions in Haskell (and many other functional languages) can return functions
- Consider the following function:
  ```
  add :: Int -> (Int -> Int)
  add x y = x + y
  ```
- This is called a *curried* function
- Why is this useful?
- Partial application!
- The arrow is right-associative, e.g. Int -> Int -> Int == Int -> (Int -> Int)

Polymorphic types

- The Prelude function *length* calculates the length of any type of list
- *length* is polymorphic!
- This is done through *type variables*
- Type variables are denoted by lower-case letters in a type definition
- *length :: [a] -> Int*

Overloading and type classes

- Let's redefine add (again)
- Add should obviously be able to add any number, but not other types
- We use a type class and overloading to redefine add:
  add :: Num a => a -> a -> a
  add x y = x + y
- *Num a* is called a class constraint, and constrains the type variable a to type class Num

Basic type classes

- Eq – equality types (can be compared using == and /=)
- Ord – ordered types (can be compared using < <= >= >, *min* and *max*)
- Show – types convertable to String
- Read – types convertable from String
- Num – numeric types
- Integral – integral types
- Fractional – fractional types

Defining functions

- There are various tools for defining functions:
  - Combine existing functions
  - Using conditional expressions
  - Using guarded equations
  - Using pattern matching

Combine existing functions

- Very simple: use existing functions in the body of your new function. For example:

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0

splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)

recip :: Fractional a => a -> a
recip n = 1 / n
```

Conditional expressions

- *If*-statements. They work similar to Java. For example:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n

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

- Haskell if-statements *always* require an *else*-branch!

Guarded equations

- Guarded equations are often a more elegant way to express conditions than if-statements. Examples:

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

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

- Otherwise is similar to *default* in switch-statements in Java

Pattern matching (1)

- Many functions can be expressed particularly intuitive using *pattern matching*
- Pattern matching defines a function as a sequence of patterns, which are evaluated in order. For example:

```
and'                :: Bool -> Bool -> Bool
and' True True       =  True
and' True False      =  False
and' False True      =  False
and' False False     =  False
```

Pattern matching (2)

- Patterns can be simplified using *wildcards* and the evaluation order.
- A simpler version of *and'*:

```
and'              :: Bool -> Bool -> Bool
and' True True     =  True
and' _ _           =  False
```

Pattern matching (3)

- Patterns can be used with tuples as well

```
fst          :: (a, b) -> a
fst (x, _) =   x


snd          :: (a, b) -> b
snd (_, y) =   y
```

Pattern matching (5)

- Patterns are even more powerful with lists.
- A function that tests if a list consists of 3 characters starting with a:

```
test                :: [Char] -> Bool
test [‘a’, _, _] =  True
test _              =  False
```

Interlude – the cons-operator

- Lists are actually not primitive, but constructed using the *cons-operator :*

```
  [1, 2, 3]
=
  1 : [2, 3]
=
  1 : (2 : [3])
=
  1 : (2 : (3 : []))
```

Pattern matching (6)

- Patterns can use the cons-operator
- Cons-patterns need to be parenthesised because of function precedence
- A more general test function:

```
test            :: [Char] -> Bool
test (‘a’: _) =   True
test _          =   False
```

Pattern matching (6)

- Head and tail use pattern matching!

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

tail           :: [a] -> [a]
tail (_:xs) =  xs
```

Local definitions

- The *where*-keyword can be used in functions to define a local function or value.
- A function that returns the first n odd integers can be defined as:

```
odds     :: Int -> [Int]
odds n   =  map f [0.. n – 1]
            where f x = x * 2 + 1
```

Lambda expressions

- Alternative to regular functions
- Nameless
- Can be used to formalise currying
- Can be used when returning functions from a function, for example defining a constant function:
```
const    :: a -> (b -> a)
const x =  \_ -> x
```
- Can be used to avoid naming a function that is used only once, for example with *map*

List comprehensions

- Allow functions on lists to be defined in a simple manner
- Allow for the construction of new lists from existing lists
- Example:
  [x ^ 2 | x <- [1..5]]
- | is read as ‘such that’
- <- is read as ‘drawn from’

Generators (1)

- x <- [1..5] is called a *generator*
- List comprehensions can have multiple generators, for example:
  [(x, y) | x <- [1,2,3], y <- [4,5]]
- The order of generators matters, this produces a different result:
  [(x,y) | y <- [4,5], x <- [1,2,3]]
- Generators can be dependant (right can depend on left):
  [(x,y) | x <- [1..3], y <- [x..3]]
- Another example:

```
concat     :: [[a]] -> [a]
concat xss =  [x | xs <- xss, x <- xs]
```

Generators (2)

- Wildcards can be used as well
- For example, to get the first elements of each pair in a list:

```
firsts    :: [(a, b)] -> [a]
firsts ps =  [x | (x,_) <- ps]
```

Guards (1)

- Used to filter the products of generators
- For example finding all positive factors of an integer:
```
factors   :: Int -> [Int]
factors n =  [x | x <- [1..n], n ‘mod’ x == 0]
```
- Finding prime if a number is prime (using factors):
```
prime   :: Int -> Bool
prime n =  factors n == [1, n]
```
- Finding primes up to n:
```
primes   :: Int -> [Int]
primes n =  [x | x <- [2..n], prime x]
```

Guards (2)

- Finding values by key in a lookup table
- Represent lookup table as a list of key-value pairs
- Implementation of ‘find’:

```
find      :: Eq a => a -> [(a, b)] -> [b]
find k t =  [v | (k’, v) <- t, k == k’]
```

The *zip* function

- The library function *zip* creates a list by pairing successive elements of two existing lists, until one or both are exhausted
- For example, creating a list of all adjacent pairs of a list:
  pairs :: [a] -> [(a, a)]
  pairs xs = zip xs (tail xs)
- This can be used to see if a list is sorted. How?

String comprehensions

- Strings are actually lists of characters, so everything that works on lists, also works on Strings!