

Functional Programming

Fall 2021

Funkcinis Programavimas

2021 Ruduo

Functional Programming: Course introduction

Advanced level course for students in Computer Science

- The course webpage, lecture slides, lecture video recordings and other materials will be available in Moodle (the link – <https://emokymai.vu.lt>)
- The course channel is created in the MS Teams environment, to be used for online consultations and presentations
- Lecturer: Linas Laibinis, e-mail: linas.laibinis@mif.vu.lt
- Office: MIF Building, Didlaukio 47, room 508

Functional Programming: Course introduction

- Lecture time: Mondays 12.00 – 13.45, remotely in MS Teams
- Exercise sessions: Mondays 10.00 – 11.45 (every second week), 14.00 – 16.00 (every week), remotely in MS Teams (going until 17.00 or 17.30, if necessary)
- It will be possible to reserve specific 15 minute time slots for exercise sessions in advance
- Exercise tasks (or assignments) are given in advance and should be submitted in Moodle before the given deadline
- **Note:** no exercise sessions this week, only one exercise session (14.00 – 16.00) next week

Functional Programming: How to pass the course

- **Final exam** is planned at the end of the course. Unlike other times, the exam time will be used as just the last opportunity to improve your course grades by presenting your exercise solutions.
- The course grade (0-10) will completely depend on the accumulated points from your given exercises (assignments) during the semester
- **4 different exercise sets** will be given and their solutions evaluated by points (0-100). Deadlines will be set for each exercise set.
The penalty for missing a deadline: 1 point for each extra day
- Solving the 1st exercise set will give up 20% course grades, 2nd one – also up to 20% grades, 3rd one – up to 30% grades, and 4th one – up to 30% grades.

Exercise points-to-grades formulas:

$(points/100) * 2$, for exs. 1-2, or $(points/100) * 3$, for exs. 3-4

Functional Programming: Course goals

- Learning the key concepts and principles of the functional programming (e.g., functional composition, recursion and induction, higher-order functions, pattern matching, polymorphism, etc.) using the Haskell language
- Solving problems and writing programs in a functional style
- Building inductive user-defined data types and write efficient functional programs for them
- Overviewing practical applications of functional programming

Functional Programming: Literature

- S. Thompson. Haskell: The Craft of Functional Programming. Addison Wesley, 2011
- M. Lipovača. Learn You a Haskell for Greater Good! (Available online). No Starch Press, 2010
- B. Sullivan et al. Real World Haskell (Available online). O'Reilly, 2008
- C. Allen and J. Moronuki. Haskell Programming: From First Principles. Gumroad, 2017

Functional Programming: What is it?

- A programming style that treats computation as application and evaluation of (mathematical) functions, and avoids changing state and mutable data
- In short, in functional programming (FP), we compute by evaluating expressions which use functions from our area of interest
- A functional program = a collection of definitions of functions and other values
- **No iteration (only recursion), no for loops, no variables.** You have to change your mindset :)

A function? Is it like a method?

- A function is not a method!
- Methods are not independent program units. They are bound to another context, like an object instance or a class
- Methods may depend upon values other than its arguments, may change the values of its arguments or some other static values
- Methods may change the values of its arguments or some other static values
- Pure functions are opposite

A function? Is it like a method? (cont.)

- A defined function "lives" independently
- It does not depend to any value other than its arguments
- Output of a function is a value or another function
- Does not change any argument's value or any other external value.
No side effects!
- Therefore, functions may be composed together regardless of the context

Functional Programming: What is it? (Once more)

- A computer programming paradigm (or style) that relies on the notion of functions modelled on mathematical functions
- Programs are combinations of expressions, which can be concrete values, variables (i.e., bindings), and functions
- Functions are expressions that can be applied to an argument or an input, and once applied, can be *reduced* or *evaluated*
- Functions are *first-class citizens*: they can be used as values or passed as arguments to other functions
- Essential property – *referential transparency*: the same function, given the same values to evaluate, always will return the same result

Functional Programming: mathematical basis and programming style

- Theoretical basis: [lambda calculus](#)
- Different programming paradigm (comparing to imperative and object-oriented), different way to express and solve problems
- Focuses primarily on values (including functions), their relationships and transformations
- Example of declarative programming (functions are also data, although more high level)
- Many programming languages: Lisp, ML family (Standard ML, Moscow ML, PolyML, Ocaml), Scheme, Haskell, Erlang, F#

Functional Programming: What are the advantages?

- Different approach to solving problems (the involved concepts, relationships, ...)
- **Side-effect free** (functions always return the same results for the same inputs, no internal state involved); Immutable data structures
- Precise and concise description of iterative and **recursive calculations**; For-loop disappears and is replaced by more flexible and powerful ways to perform iterative tasks
- Functional **composition** and **higher-order functions**; Functions can be passed as parameters or created as results

Functional Programming: What are the advantages?

- **Pattern matching** by using data constructors
- **Generics** (polymorphic types, type classes)
- **Lazy evaluation** – computation is deferred until it is actually needed;
Working with infinite data structures
- Easy and effective **parallelism** (mostly because of side-effect freeness)
- Close to mathematical definition; Easy to check/verify **correctness**;
- Based on a strong formalism. The proof is the code. There is a saying "If your functional code compiles, you're 99% done"

Why learn functional programming?

- Important to learn many languages over your career; Different perspective, different way to approach the problem
- Functional languages/ techniques become increasingly popular and important in industry, especially in data analytics
- Operate on data structure as *a whole* rather than *piecemeal*
- Good for concurrency, which is very important nowadays

Functional programming – history

- 1950s – the invention of *lambda calculus* as a tool for investigating the foundations of mathematics (Alonzo Church, Haskell Curry)
- 1958 – the first functional language (Lisp)
- 1970s – Robin Milner creates a more rigorous FP language Standard ML to help with automated proofs of mathematical theorems, however, it starts to be used for more general computing and data manipulation tasks
- 1970-80s – many more FP languages appear (Scheme, Miranda, ...)
- 1990 – introduction of the Haskell language

Functional programming – history

- 1990-2000s – influences of FP on development of Python, Perl, Ruby
- 2000s – many new FP languages (Closure, Mathematica, Erlang, Ocaml, PolyML, F# ...)
- 1990s-2000s – started to be actively used in statistics, data analytics, business mathematics
- 2010s – symbiosis of programming styles: the mainstream languages (Java, C#) started to add features from FP; Emergence of Scala = improved Java + functional programming
- 2010s – FP techniques are used for data analysis and transformations in the cloud (MapReduce)

Java vs Scala

Finding out whether a given string contains an upper case character.

In Java:

```
boolean nameUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameUpperCase = true;
        break;
    }
}
```

In Scala:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Here `_.isUpper` is a function that takes a character argument (represented by the underscore character), and tests whether it is an upper case letter.

Functional programming is the new new thing

Erlang, F#, Scala attracting a lot of interest from developers

Features from functional languages are appearing in other languages:

- **Garbage collection** Java, C#, Python, Perl, Ruby, Javascript
- **Higher-order functions** Java, C#, Python, Perl, Ruby, Javascript
- **Generics** Java, C#
- **List comprehensions** C#, Python, Perl 6, Javascript
- **Type classes** C++ "concepts"

Why Haskell?

- One of the most popular FP languages; Stood test of time
- General-purpose programming language
- Concise, precise and yet (comparatively) easy to understand and learn; Many syntactic definitions have migrated to other languages
- Good documentation, literature, and support community (via www.haskell.org)

Why Haskell?

- Very efficient implementation of FP (especially lazy evaluation, parallel computing)
- Statically typed, yet very flexible via generics and type classes
- The most popular free compiler – ghc (Glasgow Haskell Compiler), and its interactive interpreter – ghci
- Recommended setting – the Haskell Platform (ghc, ghci – interpreter, main standard packages/libraries)
- Can be freely downloaded and installed (for Linux, Windows, and MacOS) from www.haskell.org

Who uses Haskell?

- [AT&T](#) – automate form processing
- [Bank of America Merrill Lynch](#) – data transformation and loading
- [Facebook](#) – manipulating PHP code base
- [Google](#) – internal IT infrastructure
- [MITRE](#) – cryptographic protocol analysis
- [NVIDIA](#) – in-house tools
- ...

What is a function?

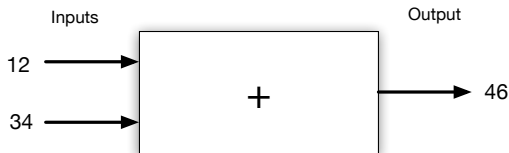
- a recipe for generating an output from inputs, e.g., "Multiply a number by itself":
- an equation, e.g., $f\ x = x^2$
- (for numbers), a graph relating inputs to some output
- In general, a kind of relationship between function input and output data, satisfying the functionality constraint – no more than one output for the same inputs.

What is a function?

A function can be seen as a box that for some given inputs (parameters, arguments) produces the output (result)



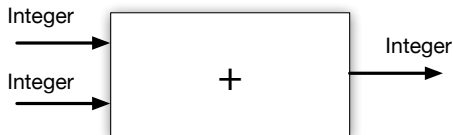
An example: addition



Operators (like +) are all treated as ordinary functions (only infix by default)

Types

A type is collection of values. Each function is defined by giving the intended types of its inputs and outputs:



Haskell is statically typed (never confusion about types)

Function types (*type signature*) in Haskell are given using the following syntax, for instance,

```
square :: Integer -> Integer
```

```
(+) :: Integer -> Integer -> Integer
```


Types (continued)

In general, using Haskell syntax, function types (type signature) is given as

`name :: $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_o$`

where `name` is the function name being defined, T_1, \dots, T_n are the types of function parameters, and T_o is the type of the function result

If `name` is an operator, it is surrounded by parentheses, e.g., `(+)`

Haskell definitions

- A Haskell definition = Type signature + Data or function declaration
- A Haskell module (file) typically consists of a collection of such definitions
- If a type signature is omitted, Haskell tries to infer the (most general) type itself
- In general, a type signature is recommended to ensure strict intended typing

Haskell definitions

- Simple value declaration:

name :: Type

name = expression

size :: Integer


size = 3 ^ 12

- Functional declaration:

fname :: $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_o$

fname p1 p2 ... pn = expression

Examples of Haskell types

- **Integers**: 42, -69
- **Floats**: 3.14
- **Characters** : 'h'
- **Strings** (lists of characters): "hello"
- **Booleans**: True, False
- User defined/implemented types, e.g., **Pictures**: 

Applying a function

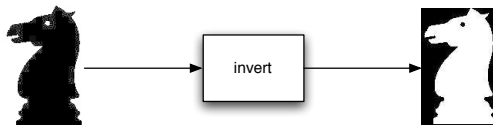
`invert :: Picture -> Picture`

`invert p = ...`

`knight :: Picture`

`knight = ...`

`invert knight`



Composing functions

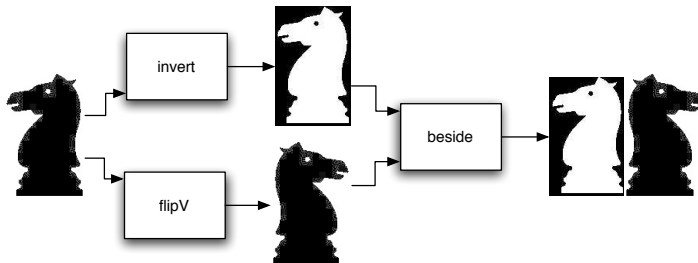
`beside :: Picture -> Picture -> Picture`

`flipV :: Picture -> Picture`

`invert :: Picture -> Picture`

`knight :: Picture`

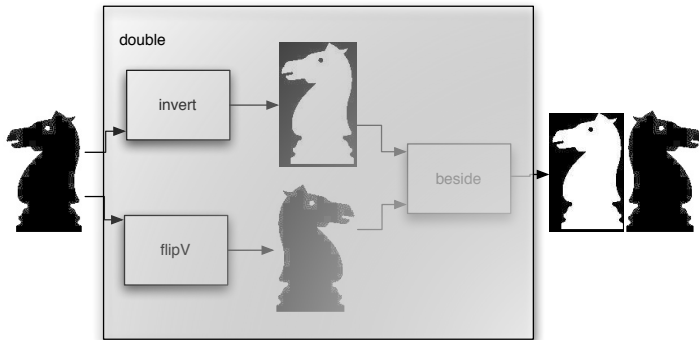
`beside (invert knight) (flipV knight)`



Defining a new function

```
double :: Picture -> Picture  
double p = beside (invert p) (flipV p)
```

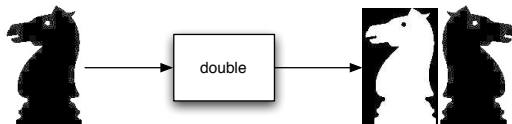
```
double knight
```



Defining a new function

```
double :: Picture -> Picture  
double p = beside (invert p) (flipV p)
```

double knight



Terminology

Type signature

```
double :: Picture -> Picture
```

Function declaration

```
double p = beside (invert p) (flipV p)
```

function name

function body

Polymorphic Types and Type Classes

Sometimes function can be applied to parameters of any type. To express that, Haskell introduces polymorphic types, for example

```
id :: t -> t  
id x = x
```

where t is a type variable, standing for arbitrary type

If we want to constrain a collection of types, the notion of type classes is used. Then the function type declaration becomes

```
name :: Class t => t -> t
```

where *Class* denotes a specific type class t belongs to

A simple module in Haskell

```
{-  
FirstScript.hs  
-}  
  
module FirstScript where  
  
-- The value size is an integer  
size :: Integer  
size = 25  
  
-- The function to square an integer  
square :: Integer -> Integer  
square n = n*n
```

A simple module in Haskell (continued)

```
-- The function to double an integer
double :: Integer -> Integer
double n = 2*n

-- An example using double, square, and size
example :: Integer
example = double (size - square (2+2))
```

Exercise sessions this and the next week

- No exercises to solve this time, thus no exercise session this week
- The first set of exercises will be given the next week (with the deadline in three weeks)
- Before the next lecture, please install the Haskell Platform on your personal computer (from <https://www.haskell.org/downloads>)
- Also – get accustomed with the ghci interpreter of Haskell
- Small pieces of code to try and experiment on can be downloaded after this lecture
- The next week – one exercise/consultation session (from 14.00). The times can be reserved using the Scheduler resource in the course page on Moodle

(Some) ghci interpreter commands

<code>:type <expr> (or :t <expr>)</code>	– the type of data expression or function
<code>:load <file> (or :l <file>)</code>	– load a Haskell module from <i>file</i>
<code>:reload (or :r)</code>	– repeat the last load command
<code>:info <name> (or :i <name>)</code>	– information about the identifier <i>name</i>
<code>:browse <name></code>	– all definitions from the module <i>name</i>
<code>:help (or :h)</code>	– the list of all interpreter commands
<code>!:<shell_command></code>	– run a shell command
<code>:quit (or :q)</code>	– quit the system

Typical ghci error messages

- **Parsing/syntax errors** "Parse error – possibly incorrect ..."
- **Wrong or undefined name** "Variable ... not in scope"
- **Typing errors** "No instance for (Type1, Type2) arising at ..."
- **Typing errors** "Could not match expected type Type1 against inferred type Type2"