



Functional Programming

Introduction to Haskell

Markos Vassilis, Mediterranean College

Week 01

A Brief Intro

Desperate Times, Desperate Measures

Since you might not yet have access on the college's educational platform, things might be a bit quirky. So, to make sure we keep track of who is here and who is not, please **scan the QR code shown next or click the link below it** and fill in this form with your information (confidential).



<https://forms.gle/JsdNsAXEFPbjN72L9>

Desperate Times, Desperate Measures

For similar reasons, we will also be using a public shared repository to keep our materials as long as our platform is a bit unstable. To visit the platform and download this lecture's materials please use the QR shown right or the link below.

<https://github.com/vmarkos-mc/functional-programming>



sudo echo whoami



- Vassilis (name) Markos (surname): it also works the other way around.
- My interests include: AI / XAI, Quantum Computing, Data Science, Operator Theory, Statistics, Software Development, CS Teaching... (this might be of your interest too in the future, in case you are looking for a **thesis supervisor**).
- I am mostly a Linux / UNIX user, so, please, be kind towards my ignorance regarding MS Windows. :)

What Is This Course?



This course is about many (many) things:

- Advanced Programming Paradigms.
 - Functional Programming, in particular.
- Advanced problem solving techniques.
- Advanced languages.
 - Haskell, in particular.
- All-things functional programming, in general.

Module(s) Assessment



All coursework will be **submitted directly on UDO** (you will get your credentials soon if you have not yet). This means that:

- Delayed submissions are not possible (including submissions by email etc).
- The only way to get an extension beyond the late submission 168 hours is by formally applying for one **at the University of Derby**.
- All UDO submissions are by default checked by Turnitin.

Module(s) Assessment



So, in order for things to run smoothly:

- You should **work** on coursework sufficiently **prior to the deadline!**
- In case of extension, **make sure you have all necessary documents** available, e.g., doctor's written diagnosis, in case of a medical condition.
- I will be accepting **drafts** which can be discussed during **office hours** to make sure things are okay for submission.
- Do not make (excessive) use of Generative AI!

Module's Coursework



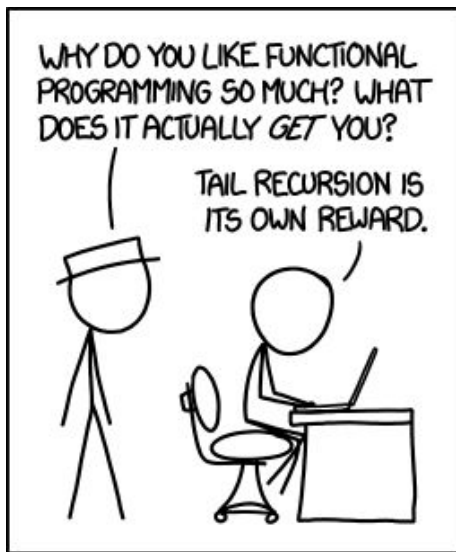
There is a two-part coursework for this module:

- **Coursework 1 (40%):** Most probably, a math expression parser developed in Haskell. Due this semester.
- **Coursework 2 (40%):** Most probably, a data processing pipeline utilising Functional Programming, in Python. Due next semester.
- **In-class labs (20%):** 4 live lab sessions, each worth 5% of your grade.
 - 2 this and 2 next semester.
 - Lab dates are strict and will be announced soon.

More to be announced soon...

Functional Programming

Some Programmer's Humour First



This course's goal is to understand functional programming so deeply that you actually consider this *funny*: Source: <https://xkcd.com/1270/>

What Is Functional Programming?



As per Wikipedia's words:

"In computer science, functional programming is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program."

https://en.wikipedia.org/wiki/Functional_programming

So? In your own words?

A Core Concept: Immutability



- When do we say that a variable is **mutable**?
 - When we can change its value.
- When do we say that a variable is **immutable**?
 - When we cannot change its value.
- When a function / expression mutates a variable we say that its invocation has **side effects**.
- In functional programming **side effects are not allowed**.
- One way to achieve this is to make sure that everything is immutable (yet, not the only way).

Immutability and Flow Control



For instance, the following simple if expression is actually mutating a variable:

```
x = 0
if x > 0:
    y = 1
else:
    y = -1
```

How can we write it without mutation?

Python's Inline if...else...



Python offers a way to avoid mutating when it comes to control flow clauses, as shown below:

```
x = 0  
y = 1 if x > 0 else -1
```

- But..., what's the difference with the previous piece of code?
- Essentially, this inline flow control statement behaves like **a function that has no side effects**:
 - it accepts a single argument, `x`, and;
 - it returns a certain value without affecting any other variable in our program.

What About Loops?



Consider the following simple while loop:

```
x = 4
while x > 0:
    x -= 1
print(x)
```

How can we write this **without mutating** things?

Take your time, please...

A Friend From The Past



Consider the following function:

```
def foo(x):  
    if x == 0:  
        return 0  
    return foo(x - 1)  
x = foo(4)
```

Is there any mutation here?

Recursion



- Recursion has (at least) two stages:
 - The base case, in which recursion terminates (e.g., the if condition in our previous example).
 - The recursive case(s), in which we call the function itself with different arguments.
- Every loop can be implemented using recursion.
- Recursion includes only function calls and some if statements.
 - if all these function calls are to immutable functions (or functions with no side effects), and;
 - since if statements can be written in an immutable manner;
 - then, recursion provides a way to represent loops immutably.

Fun Time!

Exercise #001



Implement the following in a no-side-effect manner:

```
x = int(input("Please, enter an integer: "))
if x % 3 == 0:
    y = 2024
elif x % 3 == 1:
    y = 2025
else:
    y = 4
print(y)
```

Exercise #002



Write the following loop using no loops:

```
n = int(input("Please, enter a positive integer: "))  
for i in range(n):  
    print(i, i ** 2)  
print("Done!")
```

Exercise #003



Write the following without any mutations:

```
n = int(input("Please, enter a positive integer: "))
x = 1
for i in range(n):
    if i % 4 == 1:
        x += 5
    elif i % 4 == 2:
        x -= 2
    else:
        x -= 1
print("Done!")
```

The Actual Fun...



Functional Programming is a really cool, mathematics oriented and very impressive programming paradigm.

Even if you are not convinced, as our next lab exercise, follow this tutorial:

<https://realpython.com/python-functional-programming/>

Homework 1



As your homework:

- Complete any pending in-class exercises.
- Follow the tutorial found below:

<https://realpython.com/courses/functional-programming-python/>

For any questions, or anything relevant, contact me at: `v.markos@mc-class.gr`

Homework 2



We have discussed so far what Functional Programming is, however, why is it useful?
Write a 500 - 1000 words essay addressing the following:

- a brief history of functional programming;
- reasons about why functional programming is useful;
- what are its benefits over other programming paradigms.

Submit your works at: `v.markos@mc-class.gr`

Haskell Basics

Why Learn Haskell?



Asking Gemini (Google's Generative AI Chatbot), we get the following response:

- Deepens Your Understanding of Programming Concepts (this checks out)
- Boosts Your Problem-Solving Skills (this checks out, too)
- Enhances Career Opportunities (??)
- Opens Doors to Cutting-Edge Technologies (?)
- **Personal Satisfaction (!)**

Full response here: <https://g.co/gemini/share/a9101c82bdc1>


Haskell Machinery

- Write GHC on some search engine.
- A typical result might look like what is shown next.
- Even so, GHC also stands for **Glasgow Haskell Compiler**, which is a native (evidently) Haskell compiler, and also the state of the art compiler for Haskell.
- It also offers an interactive mode, GHCi, where you can run scripts interactively. Interactive.
- Official page: <https://www.haskell.org/ghc/>

The screenshot shows a search engine result for the query 'GHC'. The top result is for 'General Health Care' with the URL 'https://ghc.gr'. The title is 'General Health Care - Πολυϊατρεία - Διαγνωστικά Κέντρα'. The description states that the diagnostic centers are accredited by the Ministry of Health and offer high-quality medical services. Below the description are several sections: 'Επικοινωνία' (Contact) with a phone number, 'Τα Κέντρα Μας' (Our Centers) listing various locations, 'Θέσεις Εργασίας' (Job Positions), 'Εργαστηριακά Τμήματα' (Laboratory Departments), 'Η Εταιρεία' (The Company), and a Facebook link. The bottom result is for 'GHC - General Health Care | Athens' on Facebook, showing a rating of 4.8 stars and a description of the services offered.

GHC

Αξιολογήσεις

 General Health Care
https://ghc.gr

General Health Care - Πολυϊατρεία - Διαγνωστικά Κέντρα
Τα διαγνωστικά κέντρα GHC είναι συμβεβλημένα με τον ΕΟΠΥΥ & ΕΔΟΕΑΠ. Με την Ιατρική καθοδήγηση από έμπειρο ιατρικό δυναμικό και μηχανήματα υψηλής τεχνολογίας ...

Επικοινωνία
Για Αθήνα: ▪ Τηλέφωνο επικοινωνίας: +30 210 689 0089 ...


Τα Κέντρα Μας
GENERAL HEALTH CARE ΠΟΛΥΙΑΤΡΕΙΑ - ΔΙΑΓΝΩΣΤΙΚΑ ...

Θέσεις Εργασίας
Τα διαγνωστικά κέντρα GHC (General Health Care ...


Εργαστηριακά Τμήματα
Γιατί Να Επιλέξετε Το GHC - Διοικητικό Συμβούλιο ...

Η Εταιρεία
ΙΑΤΡΙΚΗ ΣΤΗΝ ΠΡΑΞΗ Τι Είναι Το GHC Τα διαγνωστικά κέντρα ...

Περισσότερα αποτελέσματα από το ghc.gr »

 Facebook
https://www.facebook.com/... > Medical Center

GHC - General Health Care | Athens
Στο General Health Care έχουμε την Θεραπεία !! Κλείστε ραντεβού! Αθήνα: Δρυάδων 4 & Λ. Πεντέλης 49, Χαλάνδρι 15233 210 689 0089 generalathens@ghc.gr ...
4,8 ★★★★★ (21)

 Instagram
https://www.instagram.com/ghc

ghc.gr - General Health Care
Στο General Health Care φροντίζουμε συνολικά για την υγεία σου!! www.ghc.gr Αθήνα: Δρυάδων 4 & Λ. Πεντέλης 49, Χαλάνδρι 15233 210 689 0089 generalathens@ghc ...

Installing Haskell



To install Haskell on your machine:

- You need GHC (or some other compiler).
- You need cabal-install (the Haskell manager tool to manage packages, the Haskell installation etc).
- Probably Stack, a Haskell-oriented development tool.
- You can find further instructions for your OS here:
<https://www.haskell.org/downloads/>
- There is no actual implementation of GHCi on JavaScript, so you want find an online Haskell compiler based on GHCi.
- However, we can also use Haskell playground: <https://play.haskell.org/>

Hello, World!



As it is customary, we start our Haskell journey with the cornerstone of each programming language: `helloWorld.hs` (Haskell files typically use the `.hs` extension, in case you wondered):

```
main :: IO ()  
main = putStrLn "Hello, world!"
```

Write the above in <https://play.haskell.org/> and hit the “Run” button to observe its output!

Some First Notes...



- The first line, `main :: IO ()` actually informs Haskell about the entry point for your script.
- As with most languages, we will treat this as a necessary initiation spell for a while and explain what it actually does as our understanding of Haskell improves.
- `main = putStr "Hello, world!"` is the actual program being executed, which, as you might suspect, prints "Hello, world!" on our screens.
- It is important for the **entry point** in a Haskell script to be **named main**, as, e.g., in C / C++, so stick to that, at least for now.
- If you run GHCi locally (i.e., GHC in interactive mode), you need not worry about that; just type in your code in the GHCi console.

helloWorld.hs (Reloaded)



You can also use `print`, which, in general, should be preferred (for various reasons left to explained in the near future):

```
main :: IO ()  
main = print "Hello, world!"
```

Again, run this to see how things work!

A Thing About Parentheses



You might have observed that neither with `putStr` nor with `print` did we actually use any parentheses:

- That's correct and valid Haskell syntax. Actually, no parentheses should be used to enclose function arguments in Haskell.
- Parentheses are used for operation prioritisation and some other stuff, but, in general, we separate function arguments by whitespace.
- This might trouble you a bit in your first steps in Haskell (especially when reading code), but it will come quite in handy in the future.

Playing Around With Haskell



What do you expect the following to print?

```
add2 x = x + 2
```

```
main :: IO ()
```

```
main = print (add2 5)
```

Thankfully, it prints 7.

Our First Function!



Above, we just defined our first Haskell function:

$$\text{add2 } x = x + 2$$

Syntax in the above can be thought of as follows:

- `add2` is the function's name;
- `x` is the single argument it accepts, written on the left of the `=` symbol;
- `x + 2` is the function's output, written on the right of the `=` symbol.
- This might remind you of Python's `lambda` functions, or JavaScript's arrow function syntax, etc.

Parentheses, Again...



What if we remove parentheses from the above program?

```
add2 x = x + 2
```

```
main :: IO ()
```

```
main = print add2 5
```

What is this printing?

A Neat Error

It should have probably printed something like this:

```
Main.hs:5:8: error:
  • Couldn't match expected type: t0 -> IO ()
    with actual type: IO ()
  • The function 'print' is applied to two value arguments,
    but its type '(a0 -> a0) -> IO ()' has only one
    In the expression: print add2 5
    In an equation for 'main': main = print add2 5
|
5 | main = print add2 5
|           ^^^^^^^^^^^
```

Congratulations! Our first error! But, what is it about?

A Neat Error



- What the compiler tries to tell us is that, somehow, we have messed up with the arguments provided to `print`.
- Actually, it clearly states that it expected a single argument but it got two:
 - The first one is the function `add2`.
 - The second one is `5`.
 - Without parentheses, the compiler just consumed code left-to-right, without making any further distinctions.
- So, be careful to use parentheses appropriately to denote operation prioritisation!

Another One!



- What if we want to define a function with multiple arguments?
- Just use white space to separate them:

```
myAdd x y = x + y
```

```
main :: IO ()  
main = print (myAdd 5 6)
```

- As before, this should print some really interesting output...

Operations In Haskell




What do you expect the following to print?

```
main :: IO ()  
main = do  
    print (5 == 6)  
    print (7.4 + 5.9)  
    print (6 * 5.8)  
    print ("Hello" ++ ", " ++ "world!")  
    print 'a'
```

Did you guess correctly?

What About This One?



```
main :: IO ()
main = do
  print (5 /= 6)
  print (True && False)
  print (5 == 6 || 5 /= 6)
```

This should print:

True
False
True

Chaining Output



In general, chaining output in Haskell can be done as follows:

```
main :: IO ()  
main = print (5 == 6) >> print (7.4 + 5.9) >> print (6 * 5.8)
```

- However, this is really cumbersome and makes reading code quite difficult.
- So, Haskell offers a syntactic sugar for such cases: **do**.
- For the time being, think of >> as just this: string concatenation – things are not that simple regarding >> and its likes, but we shall delve into more details in the sequel of this module.

Infix / Prefix Notation



Haskell is a functional language, hence, functions are of great importance. Observe the following (functional) expressions:

- `foo x y` → function name, followed by function arguments.
- `x + y` → arguments hugging the function name.
- The first one is an example of **prefix notation**, i.e., the function name is written prior to all arguments.
- The second one is an example of **infix notation**, i.e., the function name is positioned between its two arguments.
 - Typically, this is useful for functions of two arguments, especially maths operations.

Infix To Prefix Notation



What will this print?

```
main :: IO ()  
main = print ((+) 5 8)
```

This prints, as you might have guessed, 13.

- If you want, for some reason, to write a typically infix function using prefix notation, you should enclose its name in parentheses.
- This is not common (the other way around works better sometimes).

Prefix To Infix Notation



What will this print?

```
main :: IO ()  
main = print (6 `min` 4)
```

As you might have guessed, 4.

- You can also call typically prefix functions using infix notation, by just enclosing their name in backticks (that key lying usually left of your 1 / ! key on a US keyboard layout).
- But, why?

Types



- Haskell is a typed language, i.e., everything has its own type which is determined prior to compilation.
- So, unlike Python, we cannot leave a type undetermined and let the compiler infer what is going on at runtime.
- Haskell in the background uses all common (and uncommon) types:

```
import Data.Typeable (typeof)
```

```
main :: IO ()
```

```
main = do
```

```
  print (typeof 'a')
```

```
  print (typeof "foo")
```

```
  print (typeof True)
```

Lists



Haskell supports lists and, as we shall see next, several list-specific operations.


- But, first, how can we define a list?

```
main :: IO ()  
main = print a
```

```
a = [1, 4, 2, 7, 3]
```

- This just defines a simple list and prints it on screen.
- Observe how **definition order does not actually matter**. This is **almost always** true in Haskell (but, not always, though).

What Will This Print?




```
main :: IO ()
main = print (a ++ b)

a = [1, 4, 2, 7, 3]
b = [5, 2, 6]
```

- As you might have expected, this prints a new list resulting from the concatenation of our two lists.
- In general, ++ is used as a concatenation operator for anything that it makes sense to concatenate (e.g., lists, strings).

What Will This Print?



```
main :: IO ()  
main = print (a ++ b)
```

```
a = [1, 4, 2, 7, 3]  
b = ["Hell", "o, w", "rld!"]
```

- In Python (and other high level languages) it is possible to have lists of more than one type of elements.
- **In Haskell this is not the case.**
- This actually prints a nice error informing us just about this fact.
- The same applies to lists of lists (of lists (of lists (...))).

Everything Comes At A Cost!



- Bear in mind that ++ has to transverse its first argument in its entirety in order to concatenate two lists - think of it as appending stuff at the end of a linked list.
- So, use ++ wisely, as it costs a lot of time for larger lists!
- An alternative that takes constant time is appending at the beginning of the list, using the following syntax:

```
main :: IO ()  
main = print (6:a)
```

```
a = [1, 4, 2, 7, 3]
```

- Beware that ++ takes as arguments two lists of the same type, while : takes an element and a list.

List Examples



What will the following print?

```
main :: IO ()  
main = print (sum a)  
  
a = [3, 2, 6, 7, -2]
```

List Examples



What will the following print?

```
main :: IO ()  
main = print (product a)  
  
a = [3, 2, 6, 7, -2]
```

Ranges



What will the following print?

```
main :: IO ()  
main = print (sum [1..5])
```

- As with Python, Haskell offers ranges, which are shortcuts to write long lists that might contain elements adhering to a pattern.
- The syntax is quite intuitive:
 - We first provide the starting element of the list, followed by two dots (..).
 - Then we write the last element of the list

Ranges



What if we want to get only the sum of odd numbers starting from 1 up to 21?

```
main :: IO ()  
main = print (sum [1,3..21])
```

- Just provide the second element of the sequence and you are done!
- This is all about Haskell ranges. We cannot generate more complex patterns other than arithmetic progressions.
- So, don't be tempted to write stuff like `[1,3,7,11,...,101]`, since it won't generate all primes (or whatnot...).

Factorials?



Can you define a Haskell function that can compute the factorial of a number, n ?

```
main :: IO ()
```

```
main = print (factorial 5)
```

```
factorial n = product [1..n]
```

Simple as that!

Heads, Tails, Init, Last...



What will the following print?

```
main :: IO ()
main = do
  print (head [2, 5, 1, 7, 8])
  print (tail [2, 5, 1, 7, 8])
  print (init [2, 5, 1, 7, 8])
  print (last [2, 5, 1, 7, 8])
```


Haskell Resources



Some interesting and really cool Haskell resources (not as cool as Haskell, though):

- <https://learnyouahaskell.com/> – large parts of our course will be based on this book, so, give it a read yourselves!
- A list of Haskell resources I could not copy-paste here due to spacing:
https://wiki.haskell.org/Learning_Haskell
- Some notes: <https://www.seas.upenn.edu/~cis1940/spring13/lectures.html>
- For visual learners:
<https://www.youtube.com/playlist?list=PLF1Z-APd9zK7usPMx3LGMZEHR3UGodd3>
- The “official” Haskell community: <https://www.haskell.org/community/>
- There’s a subreddit, too: <https://www.reddit.com/r/haskellquestions/>

Fun Time!

Self-Study Exercise #001



Read about list comprehensions in Haskell. Using your freshly acquired knowledge:

- Write a function that computes the length of a list without using Haskell's native functions.
- Write a function that computes the maximum of a list without using Haskell's native functions.
- Write a function that computes the minimum of a list without using Haskell's native functions.
- Write a function that checks whether a number is prime.

Self-Study Exercise #002



Read about pattern matching in Haskell and how that can be used to implement recursive functions. Then, re-implement the functions of the previous exercise using recursion:

- Write a function that computes the length of a list without using Haskell's native functions.
- Write a function that computes the maximum of a list without using Haskell's native functions.
- Write a function that computes the minimum of a list without using Haskell's native functions.
- Write a function that checks whether a number is prime.

Any Questions



Don't forget to fill in the questionnaire shown right!



<https://forms.gle/YU1mjxjBjBf8hyvH7>