Go Functional! – The Elixir of Programming

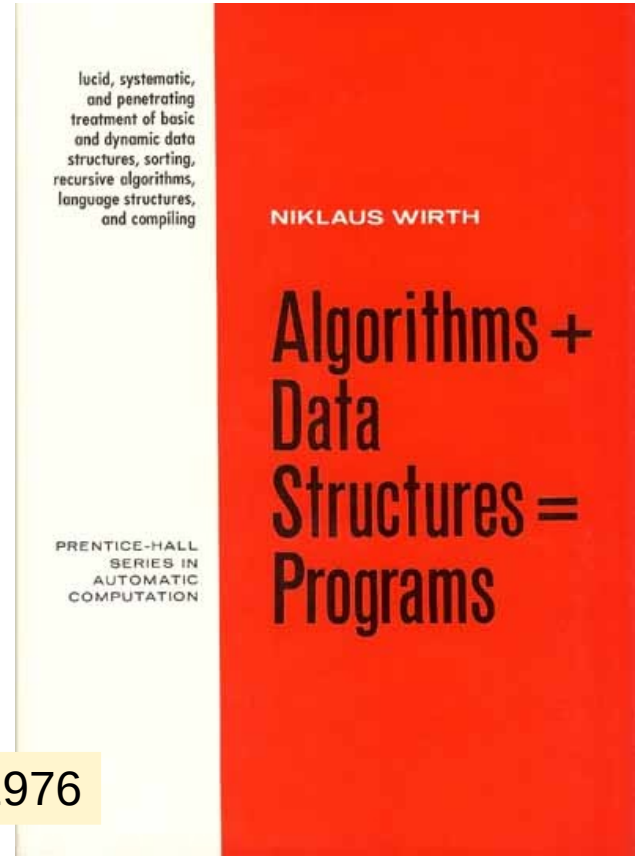
dr. Péter Hanák, Viktor Gergely
Budapest University of Technology and Economics, Hungary
phanak@edu.bme.hu
viktor.gergely@erlang-solutions.com

Algorithms + Data Structures = Programs

In functional programming:

- algorithms are ***functions,***
- data are ***immutable,*** and
- each program is an ***expression*** composed of other expressions.

- Each function has a ***return value,***
- an ***expression*** is composed of functions applied on data,
- expressions are ***evaluated.***



1976

Teaching Functional Programming (FP) at BME

Part of the course ***Declarative Programming*** (DP) at the Faculty of Electrical Engineering and Informatics, BME, since 1994:

- Standard ML (New Jersey and Moscow ML), 1994-2008
  - non-strict, modular, statically typed, formally defined, open-source, academic
- Erlang (2008-2020)
  - non-academic, dynamically typed, interpreted, outdated syntax
- Elixir (from 2021)
  - practice-oriented, functional, modern syntax, built on top of Erlang and its virtual machine, BEAM

Motivations

Originally: ***proving the correctness of ...***

- (imperative) programs is difficult,
- ***recursive*** (stateless) functions is much easier, as it can be based on ***induction.***

Later, additionally: ***by writing functional programs,***

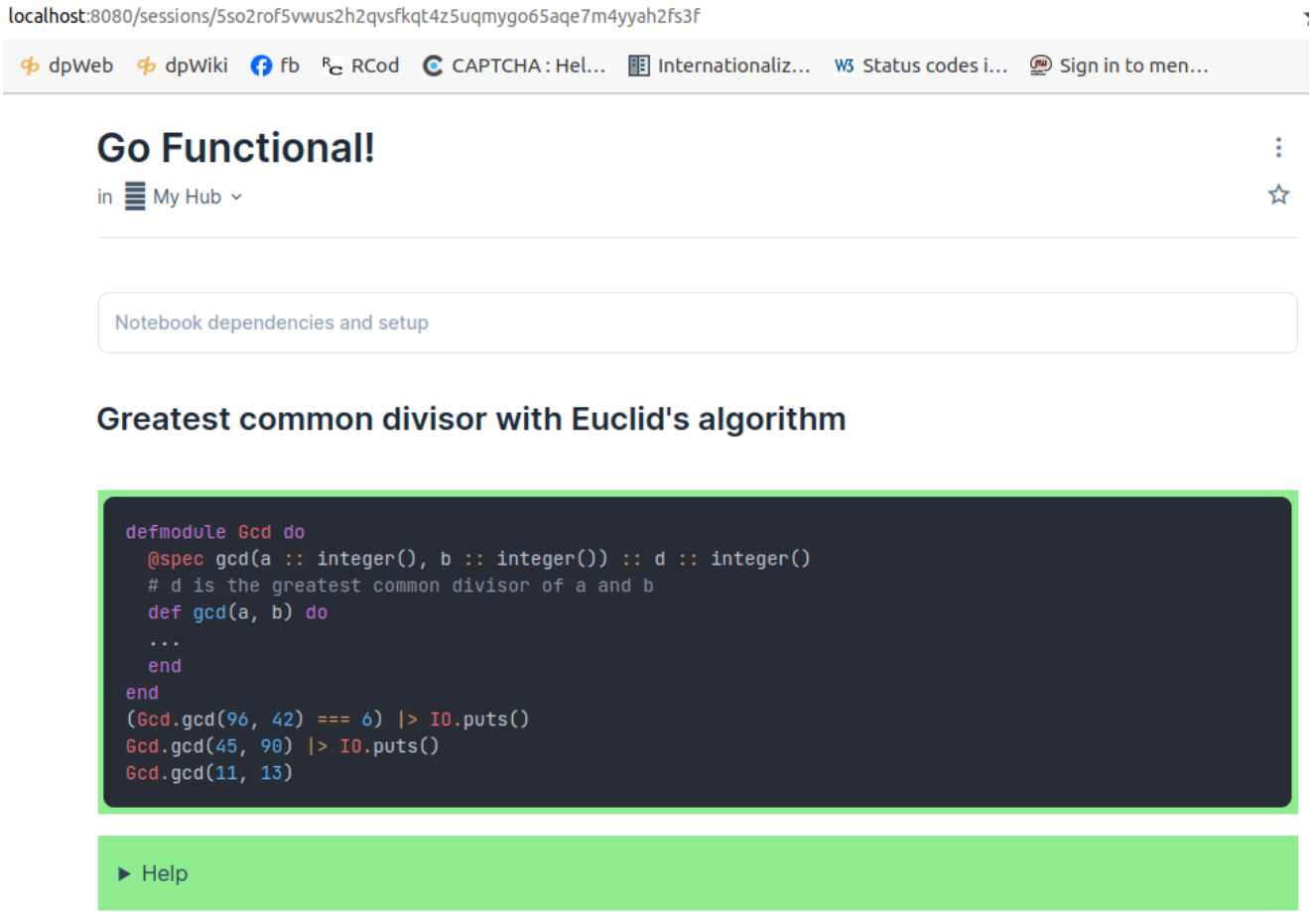
- the student acquires a good style and discipline of programming, ***as a functional program is ...***
  - built on well-established *mathematical notions* such as values, expressions, immutable data, names (unbound or bound to values), side-effect free and stateless functions,
  - *modular,* i.e. composed of many small functions performing small steps of transformation on immutable data, and returning results,
  - built on *recursivity,* i.e. composed of recursive functions and data structures.

First steps with interactive Elixir

```
hanak@gondola:~/Edu/dp/24s/2ictpl$ iex
Interactive Elixir (1.15.7) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 3+4
7
iex(2)> seven = 3+4
7
iex(3)> seven
7
iex(4)> myLinkedList = [1,3,7,11]
[1, 3, 7, 11]
iex(5)> hd(myLinkedList)
1
iex(6)> tl(myLinkedList)
[3, 7, 11]
iex(7)> myLinkedList |> tl() |> hd()
3
iex(8)> myLinkedList |> tl |> Enum.map(fn(x) -> 2*x*x end)
[18, 98, 242]
iex(9)> for x <- tl(myLinkedList) do 2*x*x end
[18, 98, 242]
iex(10)> Enum.sum(tl(myLinkedList))
21
iex(11)> myLinkedList |> tl |> Enum.sum
21
iex(12)> Enum.reduce(tl(myLinkedList), 0, fn(x,y) -> x+y end)
21
iex(13)> myLinkedList |> tl |> Enum.reduce(0, fn(x,y) -> x+y end)
21
iex(14)>
```

Shopping carts form a linked list

GCD with Elixir Livebook



GCD with Elixir Livebook

Greatest common divisor with Euclid's algorithm

```elixir
defmodule Gcd do
  @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
  # d is the greatest common divisor of a and b
  def gcd(a, b) do
  ...
  end
end
(Gcd.gcd(96, 42) === 6) |> IO.puts()
Gcd.gcd(45, 90) |> IO.puts()
Gcd.gcd(11, 13)
```

▼ Help

If $a = b \cdot q + r$, the $gcd(a, b) = gcd(b, r)$, where a, b, q and r are integers.

In each recursive step, subtract the smaller parameter from the larger one until they become equal.

See: https://en.wikipedia.org/wiki/Euclidean_algorithm

+ Elixir | + Block | + Smart

GCD with Elixir Livebook

Greatest common divisor with Euclid's algorithm

```elixir
defmodule Gcd do
  @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
  # d is the greatest common divisor of a and b
  def gcd(a, b) do
  ...
  end
end
(Gcd.gcd(96, 42) === 6) |> IO.puts()
Gcd.gcd(45, 90) |> IO.puts()
Gcd.gcd(11, 13)
```

▼ Help

If $a = b \cdot q + r$, the $gcd(a, b) = gcd(b, r)$, where a, b, q and r are integers.

In each recursive step, subtract the smaller parameter from the larger one until they become equal.

See: https://en.wikipedia.org/wiki/Euclidean_algorithm

Evaluate

```
1
```

GCD with Elixir Livebook

▼ Help

If $a = b \cdot q + r$, the $gcd(a, b) = gcd(b, r)$, where a, b, q and r are integers.

In each recursive step, subtract the smaller parameter from the larger one until they become equal.

See: https://en.wikipedia.org/wiki/Euclidean_algorithm

Evaluate

```elixir
defmodule Gcd do
  @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
  # d is the greatest common divisor of a and b
  def gcd(a, b) do
  ...
  end
end
(Gcd.gcd(96, 42) === 6) |> IO.puts()
Gcd.gcd(45, 90) |> IO.puts()
Gcd.gcd(11, 13)
```

GCD with Elixir Livebook

▼ Help

If $a = b \cdot q + r$, the $gcd(a, b) = gcd(b, r)$, where a, b, q and r are integers.

In each recursive step, subtract the smaller parameter from the larger one until they become equal.

See: https://en.wikipedia.org/wiki/Euclidean_algorithm

Evaluate

```elixir
defmodule Gcd do
  @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
  # d is the greatest common divisor of a and b
  def gcd(a, a), do: a
  def gcd(a, b) when b > a, do: gcd(b - a, a)
  def gcd(a, b), do: gcd(a-b, b)
end
(Gcd.gcd(96, 42) === 6) |> IO.puts()
Gcd.gcd(45, 90) |> IO.puts()
Gcd.gcd(11, 13)
```

GCD with Elixir Livebook

In each recursive step, subtract the smaller parameter from the larger one until they become equal.

See: https://en.wikipedia.org/wiki/Euclidean_algorithm

Reevaluate

```elixir
defmodule Gcd do
  @spec gcd(a :: integer(), b :: integer()) :: d :: integer()
  # d is the greatest common divisor of a and b
  def gcd(a, a), do: a
  def gcd(a, b) when b > a, do: gcd(b - a, a)
  def gcd(a, b), do: gcd(a-b, b)
end
(Gcd.gcd(96, 42) === 6) |> IO.puts()
Gcd.gcd(45, 90) |> IO.puts()
Gcd.gcd(11, 13)
```

Evaluated

```
true
45
```

```
1
```

Concurrent programming with Elixir

- With Elixir's language constructs, it is easy to
  - spawn lightweight processes, locally or remotely,
  - send messages from a process to other processes, incl. remote ones,
  - receive messages from processes, incl. remote ones,
  - detect the death of processes,
  - remove and restart processes;
- so Elixir is suitable for learning the basics of concurrent programming;
- but, because of time limitations, *we do not teach* the concurrent and distributed features of Elixir in our DP course.
  -

Summary, conclusions

- ***Functional programs are ...***
  - built on well-established *mathematical notions* such as values, expressions, immutable data, names (unbound or bound to values), side-effect free and stateless functions,
  - *modular,* i.e. composed of many small functions performing small steps of transformation on immutable data, and returning results,
  - built on *recursivity,* i.e. composed of recursive functions and data structures.
- ***Learning FP*** is a ***natural way to programming*** for all students who learned elementary mathematics such as arithmetic, expressions, math functions and math variables.
- ***Elixir*** is a modern FP language utilizing Erlang's modules and its virtual machine, BEAM. Elixir's ***Livebook*** is a superb interactive environment for learning programming with Elixir.
- Starting with FP, the ***student acquires a good style of programming,*** characterised by *separation of concerns*, *modularity*, *granularity*, *recursivity*, *immutability*, etc.

Thank you for your attention!

elixir

Elixir’s home: https://elixir-lang.org/
Elixir’s Livebook: https://livebook.dev/