C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

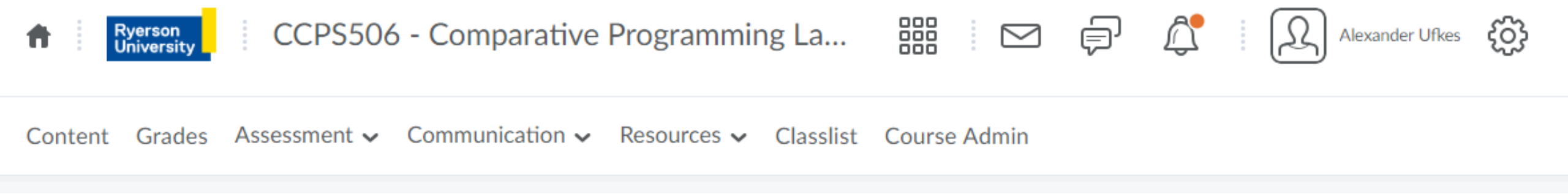
Topic 4: Lists, pattern matching, functions

Ryerson University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Course Administration (CCPS)



Labs 1 & 2 are due tomorrow!

Labs 3 & 4 posted this week

Previously

Functional paradigm,
Elixir intro

Today

Continuing Elixir:

- Basic types, Elixir scripts
- Lists and tuples, heads and tails
- Pattern matching
- Functions and modules
- Named and anonymous functions

Hello World

`IO.puts “Hello, World!”`

```
Erlang
File  Edit  Options  View  Help

Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]

Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> IO.puts "Hello, World!"
Hello, World!
:ok
iex(2)>
```

- **`IO.puts`** prints to the shell
- When executing in the shell, return values get echoed.
- In this case, **`IO.puts`** returns the atom **`:ok`**
- Atoms in Elixir are similar in concept to symbols in Smalltalk.

Elixir Scripts

Command Prompt - iex

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello, World!
Hello, World!
Hello, World!
Hello, World!

C:\Users\aufke\Desktop\ElixirScripts>
```

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs

```
1 IO.puts "Hello, World!"
2 IO.puts "Hello, World!"
3 IO.puts "Hello, World!"
4 IO.puts "Hello, World!"
```

Elixir Syntax: Basic Types

Typing literals into the shell will echo them back, assuming they are valid.

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [d

Interactive Elixir (1.6.5) - press Ctrl+C to e
iex(1)> 1
1
iex(2)> 0b10101
21
iex(3)> 0o777
511
iex(4)> 0x1F
31
iex(5)>
```

Decimal, binary, octal, and hexadecimal integers

Elixir Syntax: Basic Types

Typing literals into the shell will echo them back, assuming they are valid.

More accurately:

- Everything in Elixir is an expression, even single literals.
- Evaluating a literal simply results in that value.
- In the interactive shell, the return value is printed for us.
- **:ok** is the return value of **`IO.puts`**. The actual printing to the screen is a side effect!

Elixir Syntax: Basic Types

Erlang

File Edit Options View Help

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [c

Interactive Elixir (1.6.5) - press Ctrl+C to e
iex(1)> 1.0
1.0
iex(2)> true
true
iex(3)> "Hello, World!"
"Hello, World!"
iex(4)>
```

Floating point, Boolean, strings

Floating Point

Floating point numbers in Elixir are 64-bit, double precision

```
iex(4)> 1.0
1.0
iex(5)> 1.0e-3
0.001
iex(6)> 1.0e2
100.0
iex(7)> round 3.58
4
iex(8)> trunc 3.58
3
```

Elixir supports scientific notation

Rounding and truncate functions

Floating Point

Floating point numbers in Elixir are 64-bit, double precision

```
iex(4)> 1.0
1.0
iex(5)> 1.0e-3
0.001
iex(6)> 1.0e2
100.0
iex(7)> round 3.58
4
iex(8)> trunc 3.58
3
```

Notice:

- We can omit parentheses around function arguments.
- Multiple arguments are still separated by commas.

Boolean

Erlang

File Edit Options View Help

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]

Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> true
true
iex(2)> false
false
iex(3)> true == false
false
iex(4)> true == true
true
iex(5)> is_boolean(false)
true
iex(6)> is_boolean(1.0)
false
iex(7)>
```

Comparison operator works the way we're used to

We can check if a value is Boolean using the `is_boolean` function

Types, Values, Truthiness

In Elixir we have Boolean values **`true`** and **`false`**.
Not all languages have a Boolean type.

C does not have a Boolean type. It still supports Boolean expressions, of course.

- In C, numeric 0 is considered *False*, and everything else is considered *True*.

In Java, we have Boolean. Logical operators are only valid with Boolean operands.

Elixir complicates things by combining both approaches:

- We have Boolean True and False, but values of every other type are considered either true or false.

Boolean Expressions

Boolean:
`true, false` } `&&, ||, !`

```
iex> "gh" && false
   false
iex> "gh" || false
   "gh"
```

With these operators:

- non-false and non-nil are **true**.
- nil and false are **false**.
- 0 is considered true!

Except…

- The result isn't true or false
- It's the ***value that decided the result*** of true or false

What we actually get is the ***value that determined the truthiness of the expression***

Test Type

```
iex(10)> is_integer(8)
true
iex(11)> is_float(8)
false
iex(12)> is_number(0xFFF)
true
iex(13)>
```

Elixir is ***dynamically*** typed!

- Type errors occur at run-time, not at compile time.
- I.e., attempting some operation on incompatible types results in a run-time error.
- A static type system catches type errors at compile time

Basic Arithmetic

Erlang

File Edit Options View Help

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8

Interactive Elixir (1.6.5) - press Ctrl+C to exit (
iex(1)> 1 + 2
3
iex(2)> 5 * 5
25
iex(3)> 10 / 2
5.0
iex(4)>
```

Addition, multiplication

Division:

- Notice 5.0, despite integer operands
- / operator returns floating point in Elixir

Basic Arithmetic

div and **rem** functions

```
iex(3)> 10 / 2
5.0
iex(4)> div 10, 2
5
iex(5)> div 10, 3
3
iex(6)> rem 10, 3
1
iex(7)>
```

div:

- Result of integer division
- Like / in Java

rem:

- Remainder operator
- Same as % in C
- Requires integer arguments

```
iex(1)> rem 10, 3.0
** (ArithmeticError) bad argument in arithmetic expression
    :erlang.rem(10, 3.0)
iex(1)>
```

Erlang

File Edit Options View Help

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]

Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h is_integer/1
* def is_integer(term)

    @spec is_integer(term()) :: boolean()

Returns `true` if `term` is an integer; otherwise returns `false`.

Allowed in guard tests. Inlined by the compiler.

iex(2)> h is_integer/2
No documentation for Kernel.is_integer/2 was found
iex(3)>
```

Getting Help:

:math.pow

Elixir inherits this from Erlang:

```
Erlang
File  Edit  Options  View  Help

Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]

Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h :math.pow
* :math.pow/2

    @spec pow(x, y) :: float() when X: number(), Y: number(), x: var, y: var

Documentation is not available for non-Elixir modules. Showing only specs.
iex(2)> :math.pow(2, 8)
256.0
iex(3)>
```

Returns float

Precedence?

It’s typical:

https://hexdocs.pm/elixir/master/operators.html

OPERATOR	ASSOCIATIVITY
`@`	Unary
`.`	Left to right
`+` `-` `!` `^` `not` `~~~`	Unary
`*` `/`	Left to right
`+` `-`	Left to right
`++` `--` `..` `<>`	Right to left
`in` `not in`	Left to right
`|>` `<<<` `>>>` `~>>` `<<~` `~>` `<~` `<~>` `<|>`	Left to right
`<` `>` `<=` `>=`	Left to right
`==` `!=` `=~` `===` `!==`	Left to right
`&&` `&&&` `and`	Left to right
`||` `|||` `or`	Left to right
`&`	Unary
`=`	Right to left
`=>`	Right to left
`|`	Right to left
`::`	Right to left
`when`	Right to left
`<-` `\\`	Left to right

Function Arity

Elixir functions are described in terms of their ***name*** and ***arity*** ?

Arity refers to the number of arguments a function takes

Elixir functions are commonly described by the following notation: `name/arity`

```
round/1
rem/2
trunc/1
div/2
is_boolean/1
```

This is not language syntax, just a documentation style.

More Types: Atoms

Atoms:

A constant whose name is its value:

```
iex> :hello
  :hello
iex> :world
  :world
iex> :hello == :world
  false
```

Boolean literals are Atoms:

```
iex> :true == true
  true
iex> :false == false
  true
iex> is_boolean(:false)
  true
```

Elixir atoms are the equivalent of Smalltalk symbols.
Atoms with same value exist only once in memory.

More Types: Strings

Can have line breaks in them:

```
iex> "Hello,
...> World!"
  "Hello,\nWorld!"
iex> "Hello,\nWorld"
  "Hello,\nWorld"
```

We can use `IO.puts/1` to print a string:

```
iex> IO.puts "Hello\nWorld!"
  Hello
  World!
  :ok
```

Newline is evaluated when we use the puts function.

`IO.puts/1` returns the atom **`:ok`** after printing

More Types: Strings

Unicode: length VS number of bytes

Strings in Elixir are represented by sequences of bytes.

However. Unicode characters beyond 255 require two bytes to represent:

```
iex> byte_size "Hello"
  5
iex> byte_size "Hellö"
  6
iex> String.length "Hellö"
  5
```

Use `String.length/1` to find the number of characters.

More Types: Strings

Interpolation, concatenation

```
iex> name = "Alex"
  "Alex"
iex> "Hello, #{name}!"
  "Hello, Alex!"
```

Interpolation

```
iex> name = "Alex"
  "Alex"
iex> "Hello, " <> name
  "Hello, Alex"
```

Concatenation

Type Summary

Integer:

```
2, -7, 0x1F, 0o777, 0b10101
```

Float:

```
2.0, 1.1e-3, 3.14e7
```

```
+, -, *, /
div/2, rem/2
>, >=, <, <=, ==, !=
```

Boolean:

```
true, false
```

Atom:

```
:Hello, :world, :false, :true
```

String:

```
“Hello, World!”, “123”, “tr\nue”
```

```
Concatenation <>,
interpolation #{}
```

Collections: Lists

- Use [ ] to define a list of values.
- Like Smalltalk, values can be anything (heterogeneous).
- Operating on lists is **central** in functional languages

```
iex> [1, 2, true, 3, false]
  [1, 2, true, 3, false]
iex> length [1, 2, 3]
  3
```

Use **`length/1`** to print the number of items in the list.

Lists are implemented in Elixir as ***linked*** lists.

*Lists are implemented in Elixir as **linked** lists.*

Heads 

Tails

```
iex> list = [1, 3.14, true, "Hello", :World]
   [1, 3.14, true, "Hello", :World]
iex> hd list
   1
iex> tl list
   [3.14, true, "Hello", :World]
```

Return first element of list

Return all but first element of a list

*Lists are implemented in Elixir as **linked** lists.*

Heads

Tails

```
iex(30)> hd [1]
1
iex(31)> tl [1]
[]
```

List Concatenation & Subtraction

```
iex> list = [1, 2, true, :Hello, "World", false, 5]
   [1, 2, true, :Hello, "World", false, 5]
iex> list -- [true, false]
   [1, 2, :Hello, "World", 5]
iex> list ++ [6]
   [1, 2, true, :Hello, "World", false, 5, 6]
iex> list -- [:Hello, "abcd"]
   [1, 2, true, "World", false, 5]
```

Safe to attempt removal of an item not in the list!

Huh?

```
iex> [104, 101, 108, 108, 111]
   'hello'
```

When the Erlang shell sees a list of printable ASCII values (0-127), it displays them as a ***charlist*** (single quotes).

IO.puts VS IO.inspect

IO.puts can't handle arbitrary lists:

```
iex> x = [1, 2.0, "Hello", :world]
[1, 2.0, "Hello", :world]
iex> IO.puts x
** (ArgumentError) argument error
    (stdlib) :io.put_chars(:standard_io, :unicode,
    [[1, 2.0, "Hello", :world], 10])
```

IO.puts wants a list containing things it can convert to Unicode.

`IO.puts` VS `IO.inspect`

We can use `IO.inspect`:

```
iex> x = [1, 2.0, "Hello", :world]
[1, 2.0, "Hello", :world]
iex> IO.inspect x
[1, 2.0, "Hello", :world]
[1, 2.0, "Hello", :world]
```

IO.inspect prints ***and*** returns the list.

`IO.puts` VS `IO.inspect`

We can use `IO.inspect`:

```
iex> x = [104, 101, 108, 108, 111]
'hello'
iex> IO.inspect x
'hello'
'hello'
```

IO.inspect still prints charlists!

IO.puts VS IO.inspect

We can use `IO.inspect`:

```
iex> x = [104, 101, 108, 108, 111]
'hello'
iex> IO.inspect x
'hello'
'hello'
iex> IO.inspect (x, charlists: :as_lists)
[104, 101, 108, 108, 111]
'hello'
```

Invoke IO.inspect thusly:

Prints list as a list, rather than converting to Unicode.

Collections: Tuples

A sequence of values whose elements are stored <u>contiguously</u> in memory

```
iex> tup = {1, "2", :three}
   {1, "2", :three}
iex> elem tup, 0
   1
iex> elem tup, 1
   "2"
iex> elem tup, 2
   :three
iex> elem tup, 3
   ** (ArgumentError) argument error
   :erlang.element(4, {1, "2", :three})
```

This means we can directly access individual elements using **`elem/2`**:

Lists & Tuples are *Immutable*

Operations result in new lists/tuples:

```
iex> tup
   {1, "2", :three}
iex> put_elem(tup, 1, 55)
   {1, 55, :three}
iex> tup
   {1, "2", :three}
```

`put_elem/3` to change an element

`put_elem/3` returned a different tuple. The original hasn't changed.

Bind a new label (or re-bind tuple) to save the result.

```
tup = put_elem(tuple, 1, 55)
```

Elixir Variables are *Immutable*

When we say: x=1

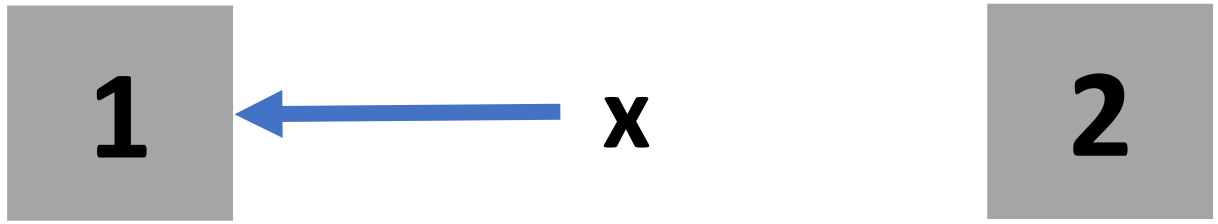
- The value 1 is created in memory
- **x** is a ***label*** for the value 1

If we then say: x=2

- We are ***NOT*** changing the value 1 in memory.
- We are creating the value 2 at a different place in memory
- **x** is now a label for the value 2
- This is called ***re-binding***.
- We change the label, not the value.

Elixir Variables are *Immutable*

```
x = 1
x = 2
```

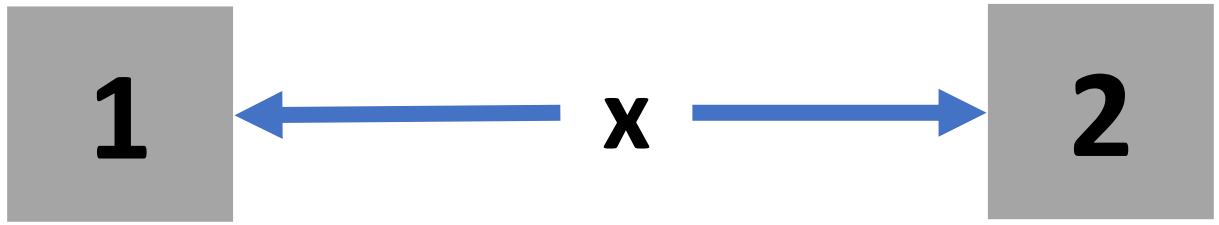


Elixir Variables are *Immutable*

x = 1

x = 2

Garbage:



Elixir Variables are *Immutable*

- In imperative languages, variables can be thought of as containers.
- We can put whatever we want into the container (if the type system allows it, of course)
- Reassigning a variable means mutating the value in the container

- In Elixir and other functional languages, variables are *labels*.
- We can change the value that we apply the label to, but we can't change the value itself.

Lists or Tuples?

Lists in Elixir are ***linked*** lists:

- Linear time to access and append, constant time to *pre*-pend
- Use to store a collection of values

Tuples are ***contiguous***:

- Constant time to access, linear time to update
- Use when size and contents are fixed at compile time.
- Not meant to be iterated over, direct element access only.

Both are immutable, updating creates new list/tuple. ***However:***

- If we change one element in a 10-element tuple, we don't actually duplicate values for the 9 unchanged elements.
- Behind the scenes, elements can be shared.

Lists or Tuples?

Just like Python:

- Fixed, small number of elements, need fast random access? **Use a tuple.**
- Large number of elements, changing in size over time, don't need random access? **Use a list.**
- Always keep in mind the cost of operations on arrays VS linked lists (C/CPS305)

Pattern Matching

=

This is ***not*** the assignment operator. It is the *match* operator.

Pattern matching is a fundamental part of Elixir

x = 1

When a name is on the left-hand side of the match operator, we ***bind*** or ***rebind*** the name.

(Variable) Name on the Right?

```
iex> x = 2
  2
iex> 2 = x
  2
iex> 3 = x
  ** (MatchError) no match of right hand side value: 2
iex> 3 = x + 1
  3
```

This is a valid expression!

If a match is successful, it returns the value of the right-hand side of the expression. If not, a **`MatchError`**.

(Variable) Name on the Right?

```
iex> x = 2
  2
iex> 2 = x
  2
iex> 3 = x
  ** (MatchError) no match of right hand side value: 2
iex> 3 = x + 1
  3
```

This is a valid expression!

Names on the left? Bind or rebind to value on the right.
Names on the right? Pattern match with value on the left.

Matching Lists

Let's see matching with lists:

```
iex> list = [1, 2, 3, 4, 5]
   [1, 2, 3, 4, 5]
iex> [1, 2, 3, 4, 5] = list
   [1, 2, 3, 4, 5]
```

Matching Lists

```
iex> list = [1, 2, 3, 4, 5]
   [1, 2, 3, 4, 5]
iex> [1 | tail] = list
   [1, 2, 3, 4, 5]
iex> tail
   [2, 3, 4, 5]
iex> [2 | tail] = list
   ** (MatchError) no match of right hand side
   value: [1, 2, 3, 4, 5]
```

- Separates list into head and tail.
- In this case, the head must be 1!

A pattern match will error if the sides can't be matched

```
iex> tail
   [2, 3, 4, 5]
iex> [2 | _] = tail
   [2, 3, 4, 5]
iex> [2, 3 | test ] = tail
   [2, 3, 4, 5]
iex> test
   [4, 5]
iex> [_ | test ] = tail
   [2, 3, 4, 5]
```

- Not equating the tail (of `tail`) with anything
- '_' can never be read from. Value discarded.

Match first two values

Match **`test`** with tail of **`tail`**

Matching Tuples

```
iex> tup = {:OK, "Hello"}
   {:OK, "Hello"}
iex> {:OK, value} = tup
   {:OK, "Hello"}
iex> value
   "Hello"
```

- When matching tuples, the comma is used as a separator.
- Tuples don't deal in head/tail
- They aren't linked lists.
- Comma for tuples, | for lists.

```
iex> {a | b} = {1, 2, 3, 4, 5}

** (CompileError) iex: misplaced operator |/2
```

The | operator is typically used between brackets as the cons operator:

```
    [head | tail]
```

where head is a single element and the tail is the remaining of a list. It is also used to update maps and structs, via the %{map | key: value} notation, and in typespecs, such as @type and @spec, to express the union of two types

Matching Tuples

```
iex> {a, b, c} = {:hello, "World", 42}
   {:hello, "World", 42}
iex> {a, b} = {:hello, "World", 42}
   ** (MatchError) no match of right hand side
   value: {:hello, "World", 42}
```

This is called *destructuring*. **a**, **b**, **c** are now bound to individual elements of the tuple.

Pin Operator

If we use the match operator with a variable on the left side of the expression, that variable is simply re-bound to that value. For example:

```
iex> x = 3
   3
iex> x = 2
   2
```

This is often undesirable!

```
iex> x = 3
  3
iex> ^x = 2
  ** (MatchError) no match
  of right hand side value: 2
```

Use ^ operator to force x to hold its binding

Pin Operator: Lists & Tuples

```
iex> x = 2
   2
iex> [^x, y] = [1, 3]
   ** (MatchError) no match of right hand side value: [1, 3]
iex> y
   ** (CompileError) iex:2: undefined function y/0
iex> [^x, y] = [2, 3]
   [2, 3]
iex> y
   3
```

Functions

$$f(x)$$

Recall: First Class VS Higher Order

Higher Order functions

- Can accept a first-class functions as an arguments
- Can return a first-class function

First Class functions

- Can be passed to higher order functions as arguments
- Can be returned from higher order functions

Named Functions

Defined within a module

- Modules can contain multiple functions
- Modules can be compiled independently, making functions available for later use.
- Named functions are not first class!
  - Cannot be passed as arguments, cannot be returned
- Named functions of same name can have different arity, unlike anonymous functions (coming up)

Modules and Named Functions

Module name:

```
defmodule HelloWorld do
  def hello do
    IO.puts "Hello world!"
  end
end
```

Function name and parameter list

Function expressions

D:\GoogleDrive\Teaching - Ryerson\CCPS 506\Resources\Code\Elixir\HelloWorld.ex -...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

HelloWorld.ex

```
1 defmodule HelloWorld do
2   def hello do
3     IO.puts "Hello world!"
4   end
5 end
6
```

length : 81 line Ln : 6 Col : 1 Sel : 0 | 0

Windows PowerShell

```
PS C:\_cps506\elixir> elixirc HelloWorld.ex
PS C:\_cps506\elixir>
```

View

C > OS_Install (C:) > _cps506 > elixir

Search elixir

Name	Date modified	Type	Size
Elixir.HelloWorld.beam	2/1/2019 3:33 PM	Erlang beam code	2 KB
HelloWorld.ex	2/1/2019 3:32 PM	EX File	1 KB

Erlang bytecode

C:_cps506\elixir\HelloWorldScript.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro

HelloWorld.ex | HelloWorldScript.exs

```
1 HelloWorld.hello
2
```

In a script file (.exs)

Windows PowerShell

```
PS C:\_cps506\elixir> elixirc HelloWorld.ex
PS C:\_cps506\elixir> elixir HelloWorldScript.exs
Hello world!
PS C:\_cps506\elixir>
```

Command Prompt - iex

```
C:\_cps506\elixir>iex
Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> HelloWorld.hello
Hello world!
:ok
iex(2)>
```

Named Functions

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugin

simple.exs | ElixirScriptList.txt | module.ex

```
1 defmodule Greeter do
2   def hello(name) do
3     "Hello, " <> name
4   end
5 end
6
7 IO.puts Greeter.hello("Alex")
8
9
```

length : 114 lines : 9 Ln : 9 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello, Alex

C:\Users\aufke\Desktop\ElixirScripts>_
```

Named functions (and modules) can be defined in a script, but then we can't use them later (without including their source code)

Named functions (and modules) can be defined in a script, but then we can't use them later (without including their source code)

C:\Users\aufke\Desktop\ElixirScripts\Greeter.ex - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | Greeter.ex | ElixirScriptList.txt

```
1 defmodule Greeter do
2   def hello(name) do
3     "Hello, " <> name
4   end
5 end
6
7
```

length : 81 lines : 7 Ln : 7

- Define Greeter module in the file "Greeter.ex"
- Compile it with `elixirc`:

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixirc Greeter.ex

C:\Users\aufke\Desktop\ElixirScripts>
```

This PC › Desktop › ElixirScripts

Search ElixirScripts

Quick access
GoogleDrive
Downloads
Desktop
Labs

Name	Date modified	Type	Size
Elixir.Greeter	5/10/2018 12:02 PM	Erlang beam code	2 KB
ElixirScriptList	5/10/2018 10:33 AM	TXT File	1 KB
Greeter.ex	5/10/2018 11:58 AM	EX File	1 KB
simple.exs	5/10/2018 11:59 AM	EXS File	1 KB

As long as the script is in the same folder,
we can invoke functions from Greeter:

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

simple.exs | Greeter.ex | ElixirScriptList.txt

```
1 IO.puts Greeter.hello("Alex")
2 IO.puts Greeter.hello("Alex")
3 IO.puts Greeter.hello("Alex")
4
5
```

length : 95 lines : 5 Ln : 5 Col : 1 Sel : 0 | 0 Windows (CR LF) UT

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixirc Greeter.ex

C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello, Alex
Hello, Alex
Hello, Alex

C:\Users\aufke\Desktop\ElixirScripts>_
```

Private Functions, Default Arguments

Private function:
Can only be invoked inside Greeter module

```
defmodule Greeter do
  defp hello(), do: "Hello "
  def greet(name \\ "Bill"), do: hello() <> name
end
```

Default argument:
If no argument is provided, name will be “Bill”

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | Modules.ex

```
IO.puts Greeter.greet("Alex")
IO.puts Greeter.greet()
IO.puts Greeter.hello()
```

length : 83 lines : Ln : 5 Col : 1 Sel : 0 | 0 Windo

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello Alex
Hello Bill
** (UndefinedFunctionError) function Greeter.hello/0 is
undefined or private
          Greeter.hello()
    simple.exs:3: (file)
    (elixir) lib/code.ex:677: Code.require_file/2

C:\Users\aufke\Desktop\ElixirScripts>
```

Return Values

- We don't have an imperative-style return statement in Elixir
- The result of the final expression is returned.

```
defmodule Silly do
  def print() do
    "Hello"
    ","
    " "
    "World!"
  end
end
```

Four expressions

```
IO.puts Silly.print()
```

↓

```
World!
```

Overloading

C:\Users\aufke\Desktop\ElixirScripts\Modules.ex - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | Modules.ex | ElixirScriptList.txt

```
1 defmodule Greeter do
2   def hello(), do: "Hello!"
3   def hello(name), do: "Hello, " <> name
4   def hello(name1, name2) do
5     "Hello " <> name1 <> " and " <> name2
6   end
7 end
8
9
```

Nor length : 181 lines : 9 Ln : 9 Col : 1 Sel : 0 | 0

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window

simple.exs | Modules.ex | ElixirScriptList.txt

```
1 IO.puts Greeter.hello()
2 IO.puts Greeter.hello("Alex")
3 IO.puts Greeter.hello("Alex", "Mike")
4
5
```

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixirc Modules.ex

C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello!
Hello, Alex
Hello Alex and Mike

C:\Users\aufke\Desktop\ElixirScripts>_
```

Multiple Modules

C:\Users\aufke\Desktop\ElixirScripts\Modules.ex - Notepad++

```
1  defmodule Greeter do
2    def hello(), do: "Hello!"
3    def hello(name), do: "Hello, " <> name
4    def hello(name1, name2) do
5      "Hello " <> name1 <> " and " <> name2
6    end
7  end
8
9  defmodule Length do
10   def of([]), do: 0
11   def of([_ | tail]), do: 1
12 end
13
14
```

ElixirScripts

This PC › Desktop › ElixirScripts

Name	Date modified	Type
Elixir.Greeter	5/10/2018 12:22 PM	Erlang
Elixir.Length	5/10/2018 12:22 PM	Erlang
ElixirScriptList	5/10/2018 10:33 AM	TXT Fi
Modules.ex	5/10/2018 12:22 PM	EX File
simple.exs	5/10/2018 12:15 PM	EXS Fil

5 items

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixirc Modules.ex

C:\Users\aufke\Desktop\ElixirScripts>
```

Anonymous Functions

Can be created live, inline:

Delimited by **fn** and **end** keywords

```
iex> add = fn a, b -> a + b end
```

Function parameters

Function behavior

- "Anonymous" functions can still be named.
- They are ***first class***
- Can be passed to another function and ***invoked*** there.

Anonymous Functions

Invoke using the dot operator:

```
iex> add = fn (a, b) -> a + b end
   #Function<12.99386804/2 in :erl_eval.expr/5>
iex> add.(1, 2)
   3
iex> add.(8, 9)
   17
```

Arguments are passed in the usual manner

Can’t use this syntax with anonymous functions:

```
iex> add 8, 9
** (CompileError) iex:8: undefined function add/2
```

Shorthand

```
iex> add = &(&1 + &2)
   &:erlang.+/2
iex> add.(3, 4)
   7
iex> add.(8, -4)
   4
```

Function Composition

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

simple.exs | ElixirScriptList.txt

```
1 inc = fn (a) -> a + 1 end
2 x = inc.(inc.(inc.(inc.(0))))
3 IO.puts x
4
```

(Maybe) not the most readable

length : 72 line Ln : 5 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
4

C:\Users\aufke\Desktop\ElixirScripts>
```

Function Composition

The pipe operator:

```
x = inc.(inc.(inc.(inc.(0))))
```

Can be written as:

```
x = 0 |> inc.() |> inc.() |> inc.() |> inc.()
```

Result becomes first argument of next function call

Very useful with Enums and Streams (later)

Higher Order & First Class Functions

A function accepting a function as an argument?

```
defmodule UserMath do

  def hof(val, func) do
    func.(val)
  end

end
```

Two arguments:

- A numeric value and a function
- (Or so our function assumes)

Invoke **`func`** with **`val`** as argument

- If **func** is not actually a function?
- We will get a run-time type error if/when we try to use it as such.

Higher Order & First Class Functions

A function accepting a function as an argument?

```
defmodule UserMath do

  def hof(val, func) do
    func.(val)
  end

end
```

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

- Anonymous function to do some operation
- Recall – only anonymous functions can be args

```
1 sq = fn a -> a*a end
2
3 IO.puts UserMath.hof(8, sq)
4
5
```

- Pass value 8 and function sq to hof

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
64

C:\Users\aufke\Desktop\ElixirScripts>
```

Same Thing?

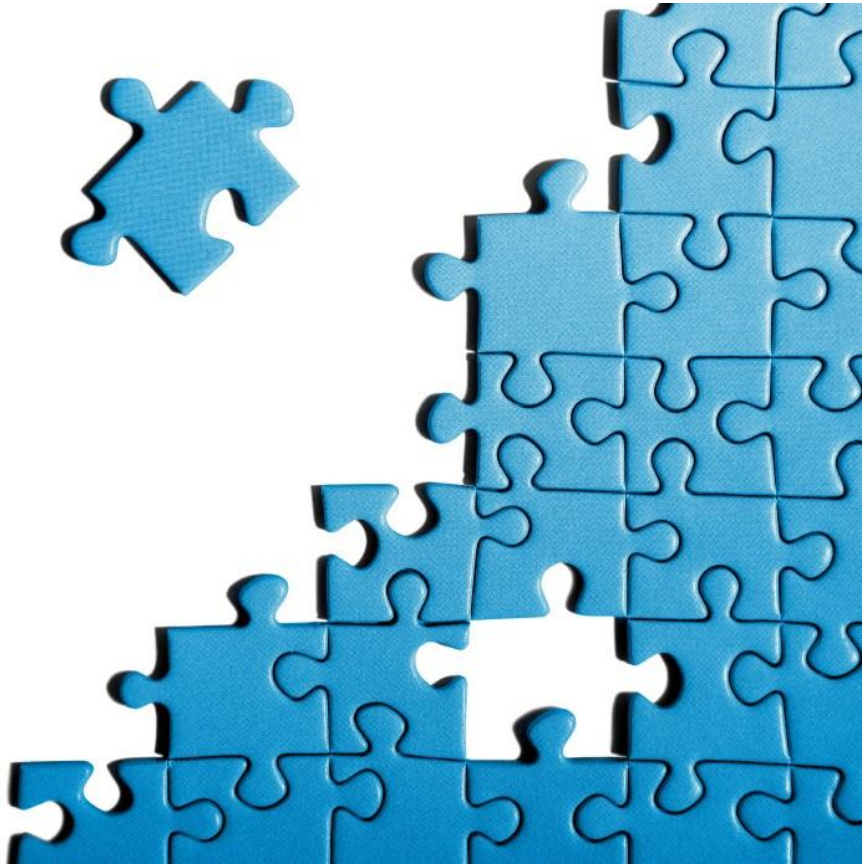
```
public class Recursion
{
    public static double add(double n1, double n2) {
        return n1 + n2;
    }

    public static void main(String[] args)
    {
        System.out.println(add(Math.pow(2.0, 2.0), 2.0));
    }
}
```

No! the <u>result</u> of pow( ) is passed as an argument, ***not*** the function itself.

Are we not passing a function to another function here?

Functions & Patterns

$f(x)$



Pattern Matching: Function Signatures

Function “overloading” is just pattern matching on the signature

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | ElixirScriptList.txt

```
handler = fn
   {:ok, result} -> IO.puts "Handled: #{result}"
   {:error} -> IO.puts "An error occurred!"
end

handler.({:ok, "Hello!"})
handler.({:error})
```

length : 165 lines : 9 Ln : 9 Col : 1 Sel : 0 | 0 Windows (CR LF)

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Handled: Hello!
An error occurred!

C:\Users\aufke\Desktop\ElixirScripts>
```

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | ElixirScriptList.txt

```
1  handler = fn
2     {:ok, result} -> IO.puts
3     {:error} -> IO.puts "An
4  end
5
6  handler.({:ok, "Hello!"})
7  handler.({:error})
8  handler.({:uhoh})
9
10
```

length : 184 lines : 10 Ln : 10 Col : 1 Sel : 0 | 0

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Handled: Hello!
An error occurred!
** (FunctionClauseError) no function clause matching in anonym
ous fn/1 in :elixir_compiler_0.__FILE__/1

    The following arguments were given to anonymous fn/1 in :e
lixir_compiler_0.__FILE__/1:

        # 1
        {:uhoh}

    simple.exs:1: anonymous fn/1 in :elixir_compiler_0.__FILE_
_/1
    (elixir) lib/code.ex:677: Code.require_file/2

C:\Users\aufke\Desktop\ElixirScripts>_
```

Ideas? What can we do?

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | ElixirScriptList.txt

```
handler = fn
   {:ok, result} -> IO.puts "Handled: #{result}"
   {:error} -> IO.puts "An error occurred!"
   {_} -> IO.puts "A SERIOUS error occurred!"
end

handler.({:ok, "Hello!"})
handler.({:error})
handler.({:uhoh})
```

length : 231 lines : 11 Ln : 11 Col : 1 Sel : 0 | 0

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Handled: Hello!
An error occurred!
A SERIOUS error occurred!

C:\Users\aufke\Desktop\ElixirScripts>
```

What about…

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | ElixirScriptList.txt

```
sum = fn
    (a, b) -> IO.puts a + b
    (a) -> IO.puts a
end
```

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
** (CompileError) simple.exs:1: cannot mix clauses with
different arities in anonymous functions

op\ElixirScripts>
```

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | ElixirScriptList.txt

```
handler = fn
    {:ok, result} -> IO.puts "Handled: #{result}"
    {:error} -> IO.puts "An error occurred!"
end

handler.({:ok, "Hello!"})
handler.({:error})
```

?

Single argument, a tuple

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

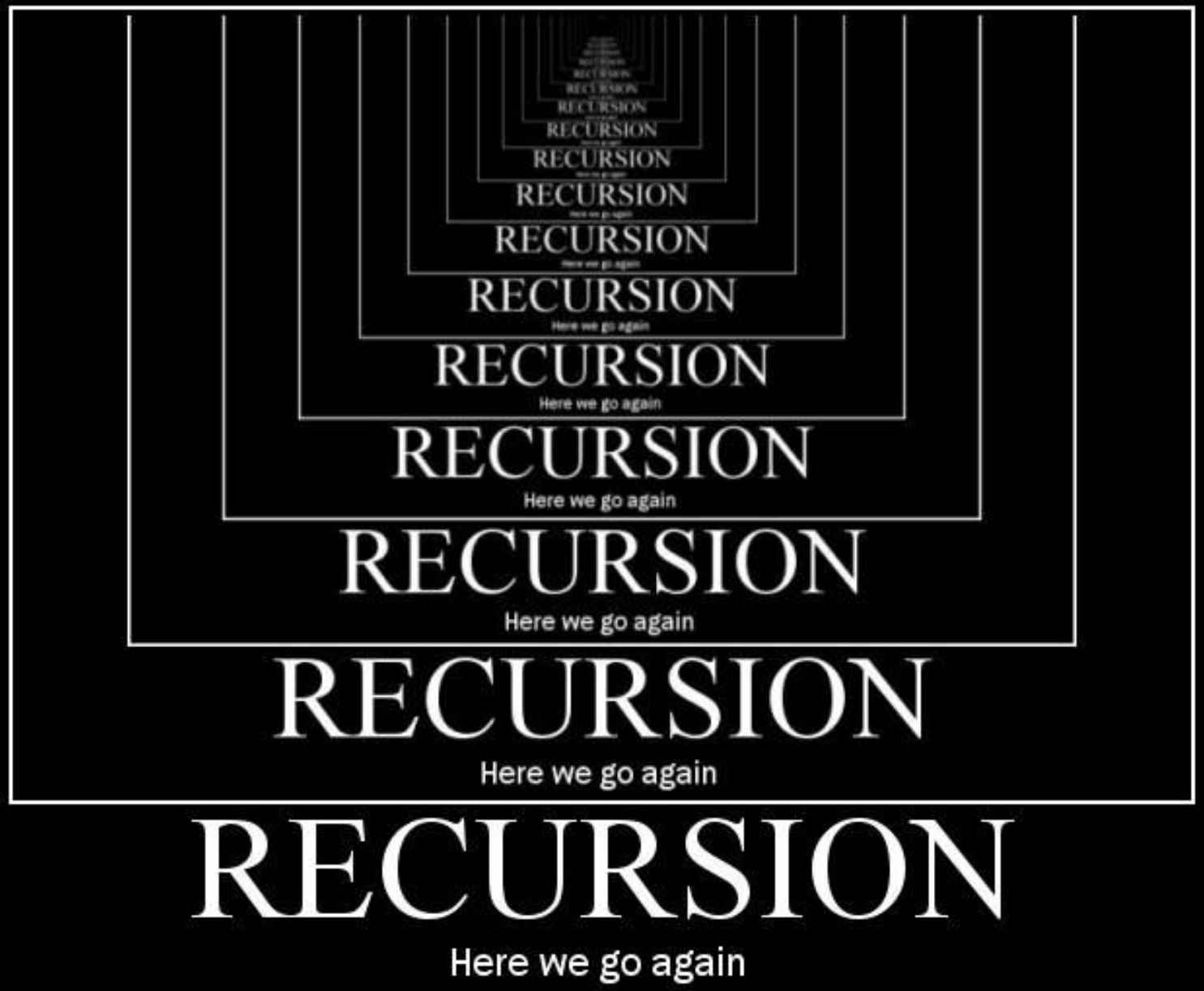
simple.exs | ElixirScriptList.txt

```
sum = fn
   (2, b) -> IO.puts 2 + b
   (a, b) -> IO.puts a*b
end

sum.(2, 1)
sum.(4, 2)
```

length : 97 line Ln : 9 Col : 1 Sel : 0 | 0 Windows (CR

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
3
8

C:\Users\aufke\Desktop\ElixirScripts>
```


RECURSION
Here we go again

Google

recursion

All Images Videos Maps Books More Settings Tools

About 7,800,000 results (0.29 seconds)

Did you mean: ***recursion***

Dictionary

Enter a word, e.g. 'pie'

re·cur·sion

/rəˈkərZHən/

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

Translations, word origin and more definitions

Feedback

Recursion in Elixir

Who needs looping anyway?

```
defmodule Length do
  def of([]), do: 0
  def of([_ | t]), do: 1 + of(t)
end
```

When there's one value left in the list, t will be [ ]

Argument pattern matching makes recursion straightforward:

C:\Users\aufke\Desktop\ElixirScripts\Modules.ex - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs | Modules.ex | ElixirScriptList.txt

```
13
14 defmodule UserMath do
15
16   def fib(0), do: 0
17   def fib(1), do: 1
18   def fib(n), do: fib(n-2) + fib(n-1)
19
20   def fac(0), do: 1
21   def fac(n), do: n*fac(n-1)
22
23 end
```

Base cases

Recursive case. ($O(2^n)$, I know)

Nc length : 438 lines : 23 | Ln : 13 Col : 1 Sel : 0 | 0 | Windows (CR LF) | UTF-8

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir s
21
720

C:\Users\aufke\Desktop\ElixirScripts>
```

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugin

simple.exs | Modules.ex | ElixirScriptList.txt

```
1 IO.puts UserMath.fib(8)
2 IO.puts UserMath.fac(6)
3
```

Tail Recursion?

Consider `UserMath.fac()`

```
defmodule UserMath do
  def fac(0), do: 1
  def fac(n), do: n*fac(n-1)
end
```

Wrapper function so user can invoke without initializing the running product

```
defmodule UserMath do
  def fac(num), do: fac(num, 1)
  def fac(0, prod), do: prod
  def fac(num, prod), do: fac(num-1, num*prod)
end
```

Pass running product as argument

Private Functions, Default Arguments

```
defmodule UserMath do
  def fac(num), do: fac(num, 1)
  defp fac(0, prod), do: prod
  defp fac(num, prod), do: fac(num-1, num*prod)
end
```

Hide the tail helper functions from the outside world

In Summary:

Continuing Elixir:

- Lists and tuples, heads and tails
- Pattern matching
- Functions and modules
- Named and anonymous functions