Welcome!

If you haven't installed Elixir, follow the instructions on:

https://elixir-lang.org/install.html

Drinking the Elixir

Agenda

- 15:00 - 15:30 Presentation
- 15:30 - 17:30 Challenges
- 17:30 - 18:30 Food!
- 18:30 - 19:00 Announcing winners

Elixir?

- build on top of Erlang
- inspired by Ruby
- immutable
- functional
- dynamically typed

Syntax

```
iex> 1           # integer
iex> 0x1F        # integer
iex> 1.0         # float
iex> true        # boolean
iex> :hello      # atom
iex> "elixir"    # string
iex> ?A          # char
iex> <<0, 255>>  # binary
iex> [1, 2, 3]   # list
iex> {1, 2, 3}   # tuple
iex> 1 + 2
3
iex> Kernel.+(1, 2)
3
iex> "Hello " <> "there!"
"Hello World!"
iex> is_binary("hellö")
true
iex> String.length("hellö")
5
iex> String.upcase("hellö")
"HELLÖ"
```

Pipes!

```
iex> "2" |> String.to_integer() |> Kernel.*(2)
4
```

Functions

```
iex> greet = fn x -> "Hello #{x}!" end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex> greet.("there")
"Hello there!"
```

Modules

```elixir
# greeting.ex
defmodule Greeting do
  def hello(), do: hello("World")
  def hello(thing) do
    "Hello " <> thing <> "!"
  end
end

iex> load("greeting.ex")
iex> Greeting.hello()
"Hello World!"
iex> Greeting.hello("there")
"Hello there!"
```

```
iex> nested = [ [0], [1], [2] ]
iex> Enum.map(nested, &Kernel.hd/1)
[0, 1, 2]
```

Pattern matching

Destructuring

```
iex> [1, a] = [1, 2]
iex> a
2
iex> {:ok, [hello: a]} = {:ok, [hello: "world"]}
iex> a
"world"
```

Pin operator

```
iex> n = 1
iex> [1, ^n] = [1, 2]
** (MatchError) no match of right hand side value: [1, 2]
iex> [1, ^n] = [1, 1]
[1, 1]
```

Responses

```
iex> case File.read("path/to/file") do
iex>   {:ok, binary_contents} -> IO.puts(contents)
iex>   {:error, reason} -> IO.puts("Error: " <> reason)
iex> end
```

```elixir
defmodule Fibonacci do
  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n), do: fib(n-2) + fib(n-1)
end
```

```elixir
defmodule Fibonacci do
  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n) when n > 0, do: fib(n-2) + fib(n-1)
end
```

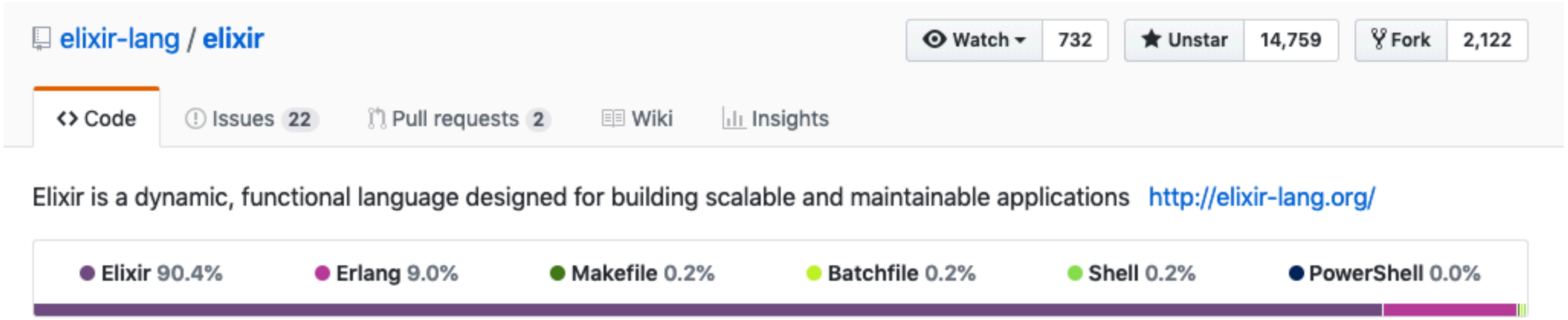
```elixir
defmodule ID3Parser do
  def parse(file_name) do
    case File.read(file_name) do
      {:ok, contents} ->
        mp3_byte_size = (byte_size(contents) - 128)
        << _ :: binary-size(mp3_byte_size), id3_tag :: binary >> = contents

        << "TAG",
            title   :: binary-size(30),
            artist  :: binary-size(30),
            album   :: binary-size(30),
            year    :: binary-size(4),
            comment :: binary-size(30),
            _rest   :: binary >> = id3_tag

      _ ->
        IO.puts "Couldn't open #{file_name}"
    end
  end
end
```

Meta-programming

Most of Elixir is written in Elixir!



Elixir is a dynamic, functional language designed for building scalable and maintainable applications http://elixir-lang.org/

● Elixir 90.4% ● Erlang 9.0% ● Makefile 0.2% ● Batchfile 0.2% ● Shell 0.2% ● PowerShell 0.0%

```
if working?() do
  do_something()
else
  do_something_else()
end

# becomes:

case(working?()) do
  x when x in [false, nil] ->
    do_something_else()
  _ ->
    do_something()
end
```

Quoted Expressions

```
iex> quote do: 1 + 2
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

```
iex> quote do: sum(1, 2 + 3, 4)
{:sum, [], [1, {:+, [context: Elixir, import: Kernel], [2, 3]}, 4]}
```

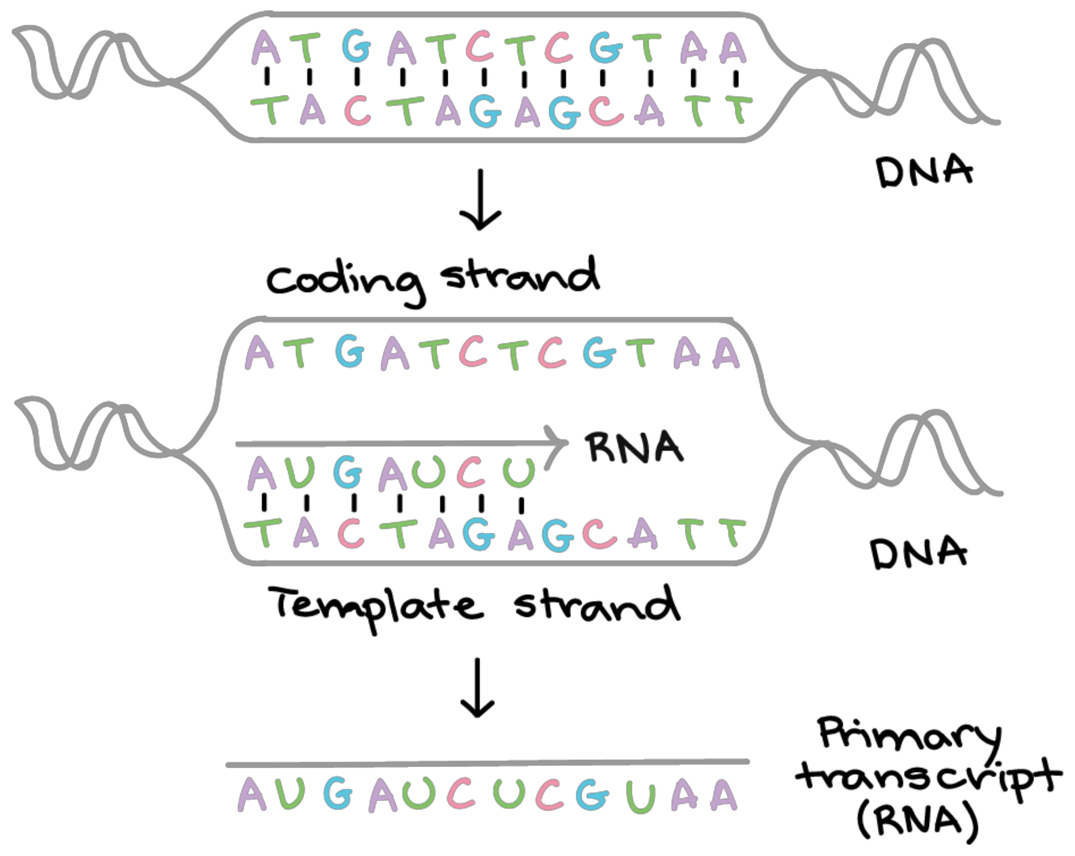
Unquoted Expressions

```
iex> number = 2
iex> quote do: 1 + unquote(number)
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

```
iex> number = 2
iex> ast = quote do: 1 + unquote(number)
iex> Macro.to_string(ast)
"1 + 2"
```

Compile-time functions

"Convert a given DNA strand to its RNA complement."



DNA	RNA
G	C
C	G
T	A
A	U

```
# rna.ex
defmodule RNATranscription do
  def to_rna(?G), do: ?C
  def to_rna(?C), do: ?G
  def to_rna(?T), do: ?A
  def to_rna(?A), do: ?U
end

iex> load("rna.ex")
iex> RNATranscription.to_rna(?T)
?A
```

```
# rna.ex
defmodule RNATranscription do
  mapping = %{ ?G => ?C, ?C => ?G, ?T => ?A, ?A => ?U }
  for { dna, rna } <- mapping do
    def to_rna(unquote(dna)), do: unquote(rna)
  end
end

iex> load("rna.ex")
iex> RNATranscription.to_rna(?T)
?A
```

Processes

OTP/Processes

- Erlang virtual machine processes
- follows the actor model

```
iex> self
#PID<0.103.0>
iex> for num <- 1..1000 do
...>   spawn(fn -> IO.puts("#{num * 2}") end)
...> end
2
4
6
8
10
12
...
```

```
iex> send(self(), "Hello!")
"Hello!"
iex> flush()
"Hello!"
:ok
```

```
iex> pid = spawn(fn ->
...>   IO.puts "Waiting for messages"
...>   receive do
...>     msg -> IO.puts "Received: " <> msg
...>   end
...>end)
Waiting for messages
#PID<0.1134.0>

iex> send(pid, "Hello world!")
Received "Hello world!"
iex> send(pid, "Hello world!")
iex>
```

```
# my_process.ex
defmodule MyProcess do
  def start, do: loop()

  def loop do
    receive do
      msg -> IO.puts "Received #{inspect msg}"
    end
    loop()
  end
end

iex> load("my_process.ex")
iex> pid = spawn(MyProcess, :start, [])
iex> send(pid, "Hello world!")
Received "Hello world!"
iex> send(pid, "Hello there!")
Received "Hello there!"
```

Remote nodes

```
$ iex --name bar@10.1.0.1 --cookie secret

$ iex --name foo@10.1.0.2 --cookie secret
```

```
iex(bar@10.1.0.1)> Node.list
[]
iex(bar@10.1.0.1)> Node.connect :"foo@10.1.0.2"
true
iex(bar@10.1.0.1)> Node.list
[:"foo@10.1.0.2"]
```

$ iex --name bar

```
iex(bar@10.1.0.1)> greetings = fn -> IO.puts "Hello from #{Node.self}" end
iex(bar@10.1.0.1)> Node.spawn :"foo@10.1.0.2", greetings
#PID<9071.68.0>
Hello from foo@10.1.0.2
```

```
iex(bar@10.1.0.1)> pid = Node.spawn(:"foo@10.1.0.2", fn ->
...>   receive do
...>     {:ping, client} -> send client, :pong
...>   end
...> end)
#PID<9014.59.0>

iex(bar@10.1.0.1)> send pid, {:ping, self}
{:ping, #PID<0.73.0>}
iex(bar@10.1.0.1)> flush
:pong
:ok
```

Challenges!

```
git clone https://github.com/avisi/techday_elixir
```