A Tour of the Elixir Source Code

Xavier Noria
@fxn

ElixirConf EU 2017

Menu:

* Project structure
* What is Elixir written in?
* Compilation
* Parallel compiler
* Implementation of protocols

Project Structure

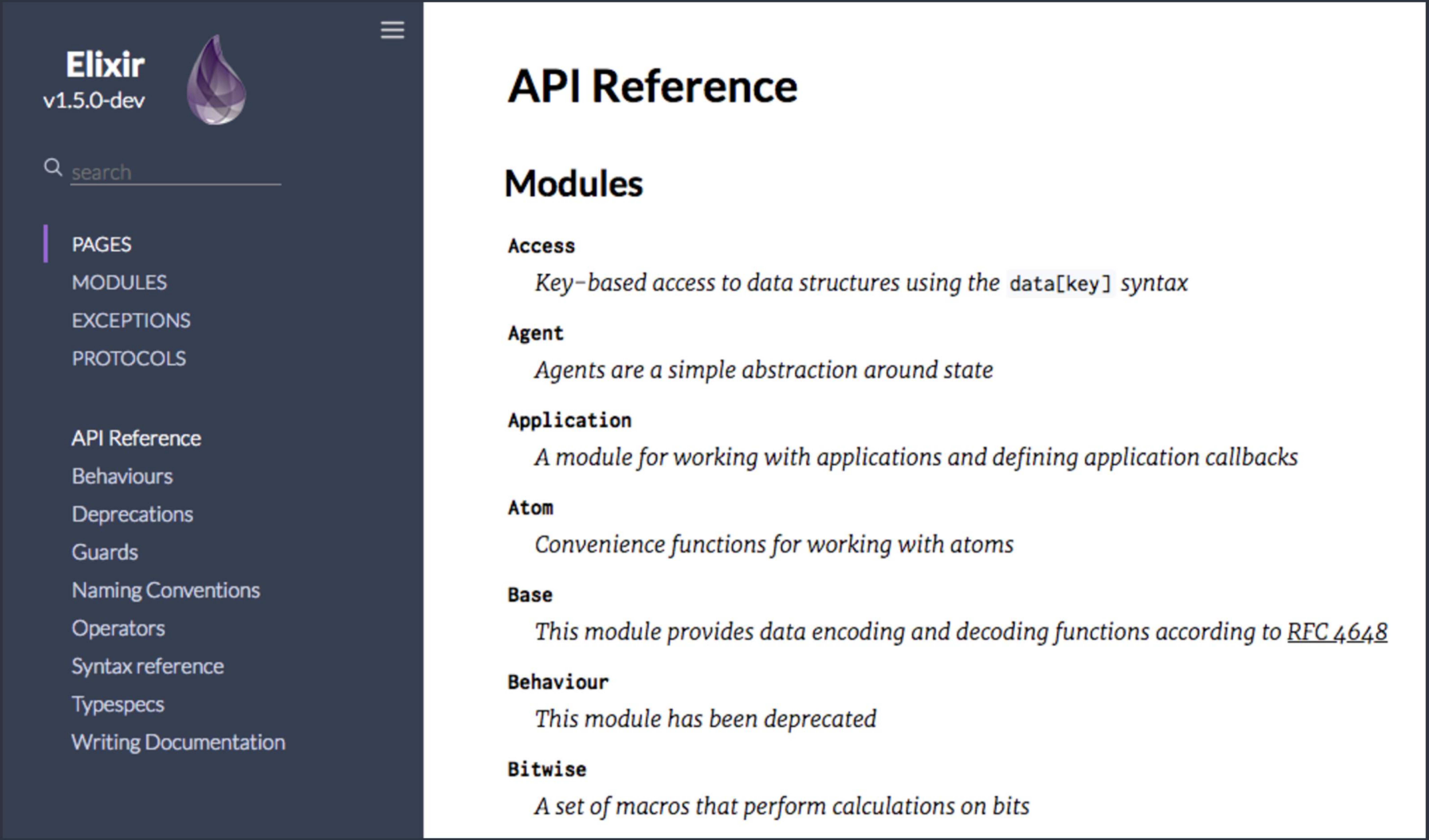
```
bin                  Makefile
lib                  NOTICE
man                  README.md
src                  RELEASE.md
CHANGELOG.md         VERSION
CODE_OF_CONDUCT.md   rebar
ISSUE_TEMPLATE.md    rebar.config
LICENSE              rebar3
```

```
lib
├── eex
├── elixir
├── ex_unit
├── iex
├── logger
└── mix
```

```
bin
├── elixir
├── elixir.bat
├── elixirc
├── elixirc.bat
├── iex
├── iex.bat
├── mix
├── mix.bat
└── mix.ps1
```

```
lib/elixir
├── lib
├── mix.exs
├── pages
├── rebar.config
├── src
├── test
└── unicode
```

```
lib/elixir/pages
├── Behaviours.md
├── Deprecations.md
├── Guards.md
├── Naming Conventions.md
├── Operators.md
├── Syntax Reference.md
├── Typespecs.md
└── Writing Documentation.md
```



API Reference

Modules

`Access`

Key-based access to data structures using the `data[key]` syntax

`Agent`

Agents are a simple abstraction around state

`Application`

A module for working with applications and defining application callbacks

`Atom`

Convenience functions for working with atoms

`Base`

This module provides data encoding and decoding functions according to RFC 4648

`Behaviour`

This module has been deprecated

`Bitwise`

A set of macros that perform calculations on bits

```
lib/elixir/unicode/
├── CompositionExclusions.txt
├── GraphemeBreakProperty.txt
├── GraphemeBreakTest.txt
├── SpecialCasing.txt
├── UnicodeData.txt
├── WhiteSpace.txt
├── graphemes_test.exs
└── unicode.ex
```

From now on, all paths are going to be relative to *lib/elixir*

What is Elixir written in?

Files		
*.erl	*src*	35
*.ex	*lib*	76

Lines of Code		
Erlang	*src*	7,000
Elixir (no docs)	*lib*	22,000
Elixir (w/ docs)	*lib*	35,000

* The core of Elixir is written in Erlang and some Elixir

* The standard library is written in Elixir, delegating to Erlang as needed

In Unix, the command

```
$ elixir foo.ex
```

expands to

```
erl -pa ... \
    -elixir ansi_enabled true \
    -noshell \
    -s elixir start_cli \
    -extra foo.ex
```

src/elixir.erl

Compilation

Main phases of Erlang compilation

Erlang

↓

Erlang AST

↓

Core Erlang

↓

Kernel Erlang

↓

BEAM Assembler

```
erlc +to_pp     foo.erl
erlc +to_exp    foo.erl
erlc +to_core   foo.erl
erlc +to_kernel foo.erl
erlc +to_asm    foo.erl
```

Erlang Tooling	
yecc	Parser generator
compile	Interface to the compiler
code	Interface to the code server
beam_lib	Interface to .beam files

How Elixir works (bird's eye)

Elixir

↓

Elixir AST

↓

Erlang AST

↓

Execution

Elixir always executes:

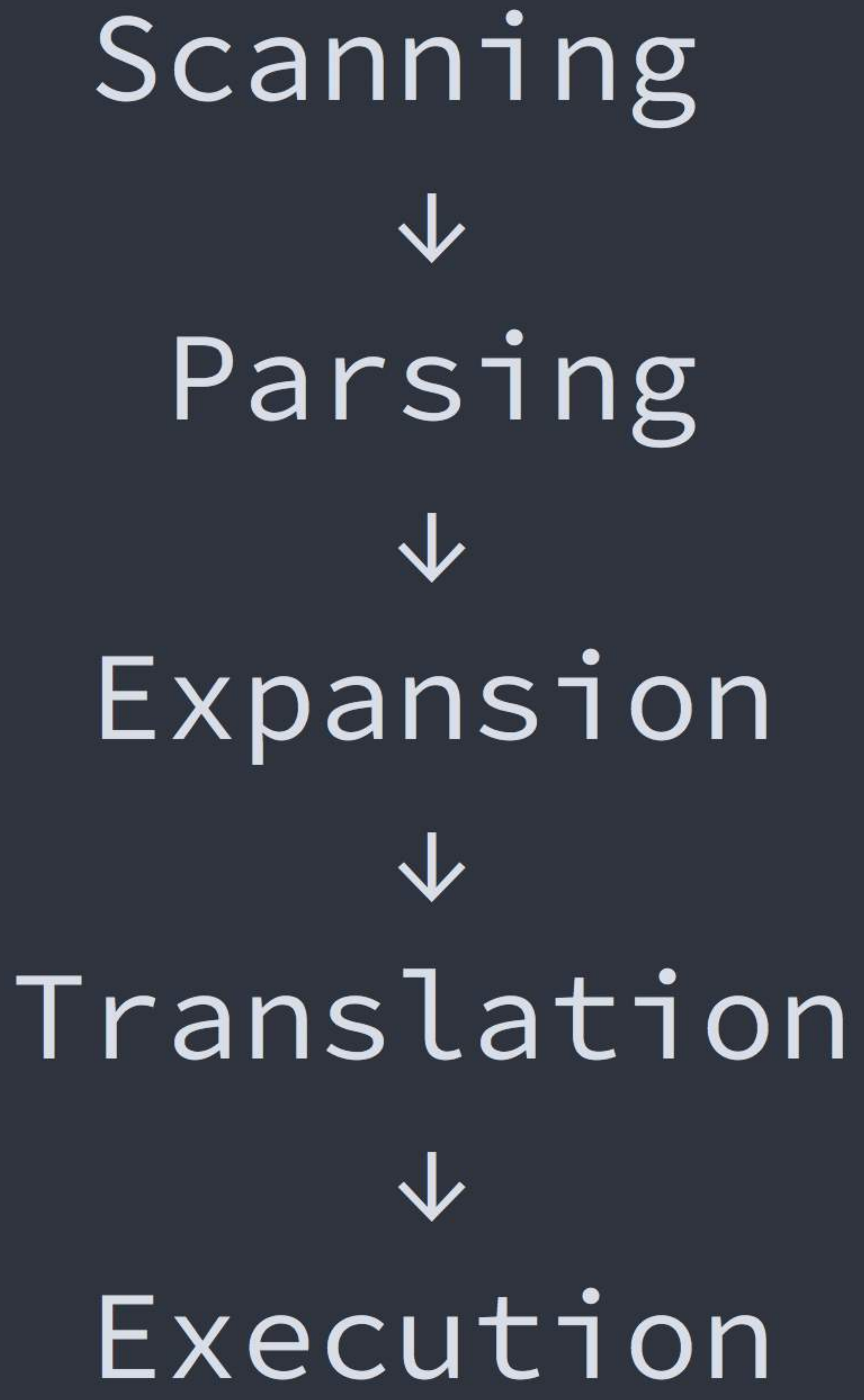
* elixir executes

* elixirc executes

elixirc generates a .beam file per module
as a side-effect of running the program

.ex vs .exs is just a convention

Main phases of Elixir compilation



Scanning

↓

Parsing

↓

Expansion

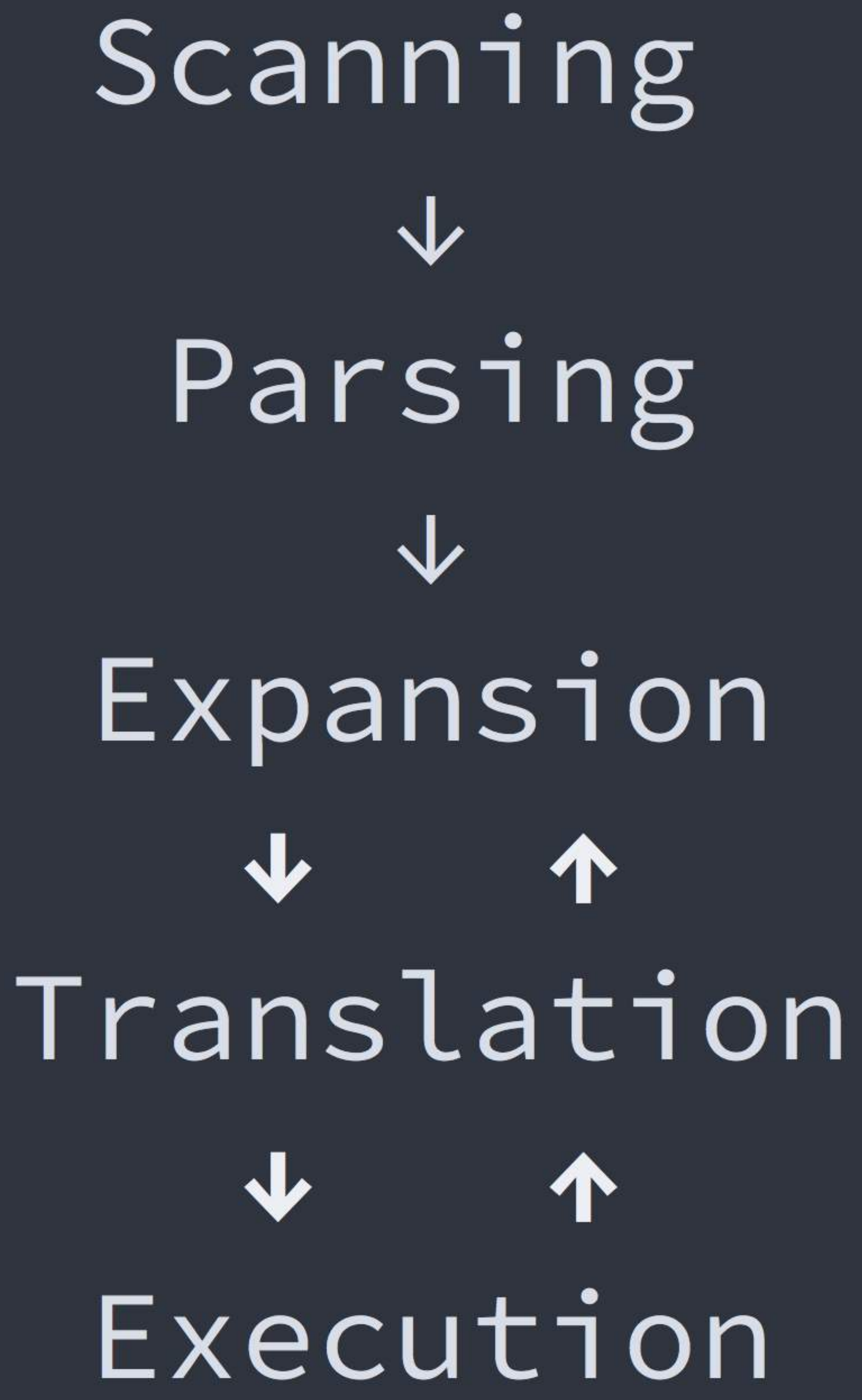
↓

Translation

↓

Execution

Main phases of Elixir compilation



Scanning

↓

Parsing

↓

Expansion

↓ ↑

Translation

↓ ↑

Execution

src/elixir_tokenizer.erl

```elixir
defmodule M do
  @moduledoc "Awesome!"
end
```

```
[{identifier,{1,1,10},defmodule},
 {aliases,{1,11,12},['M']},
 {do,{1,13,15}},
 {eol,{1,15,16}},
 {at_op,{2,3,4},'@'},
 {identifier,{2,4,13},moduledoc},
 {bin_string,{2,14,24},[<<"Awesome!">>]},
 {eol,{2,24,25}},
 {'end',{3,1,4}},
 {eol,{3,4,5}}]
```

```
tokenize(("<<<<<<<" ++ _) = ...) ->
  ...


% (SyntaxError) ... found an unexpected
version control marker
```

Hat tip!

Parsing

src/elixir_parser.yrl

yecc

src/elixir_parser.erl

Elixir AST

- Atoms, numbers, strings, lists, and 2-element tuples appear verbatim

- 3-element tuples represent:
  - structures
  - local calls
  - remote calls
  - variables
  - blocks
  - ...

```elixir
if hd(list) do
  IO.puts("yeah!")
end
```

```
{'if',
  [...], [
    {hd, [...], [{list, [...], nil}]}, [
      {do, {
        {'.', [...], [
          {'__aliases__', [...], ['IO']},
          puts]},
        [...],
        [<<"yeah!">>]}}]]}
```

```
quote do: ...
```

pages/Syntax Reference.md

Expansion

src/elixir_expand.erl

Expansion:

* Resolves aliases
* Processes requires
* Expands macros
* Inlines some function calls
* ...

```
Special Forms:

  {}           __DIR__            __block__
  %{}          __CALLER__         @
  %            ^                  fn
  <<>>         =                  __aliases__
  .            ::                 super
  alias        quote              case
  require      unquote            cond
  import       unquote_splicing   try
  __ENV__      with               receive
  __MODULE__   for
```

lib/kernel/special_forms.ex

Translation

```
src/elixir_erl_*.erl
```

Translation transforms the expanded Elixir AST into Erlang Abstract Format *http://erlang.org/doc/apps/erts/absform.html*

Execution

Elixir defines a dummy function in a dummy module, with the program as body, all directly in Abstract Format

```elixir
{_, mfa} = :erlang.process_info(
  self(),
  :current_function
)
IO.puts(inspect(mfa))

#=> {:elixir_compiler_0, :__FILE__, 1}
```

```
# ...
defmodule M do
  {_, mfa} = :erlang.process_info(
    self(),
    :current_function
  )
  IO.puts(inspect(mfa))
end

#=> {:elixir_compiler_1, :__MODULE__, 1}
```

1. The dummy module is compiled in memory with compile:forms/2

2. The dummy function invoked, which executes the program

3. The dummy module is unloaded from the VM with code:delete/1 and code:purge/1

Parallel Compiler

Erlang modules do not need to declare their dependencies

By default, Erlang loads compiled code on demand when the runtime needs to call an undefined function

Functions in the error_handler Erlang module are called, which ask the code server to load the missing module

Elixir has its own error handler and code server

```
lib/kernel/error_handler.ex
lib/code.ex
```

```
# Kernel.ParallelCompiler/spawn_compilers/3
:erlang.process_flag(
  :error_handler,
  Kernel.ErrorHandler
)
```

How does parallel compilation work?

A coordinator module spawns several compilation processes, one per file (concurrency is bounded)

When compilation of a particular file finishes, a message is sent back to the coordinator

If an undefined function is called Elixir's error handler gets triggered:

- If it belongs to a module that can be autoloaded, do so and move on

- Otherwise, tell the coordinator we are waiting for said module, and wait to be called back

- When called back, apply and move on

Implementation of Protocols

lib/protocol.ex

Kernel.defimpl/2 defines a module with the corresponding protocol implementation

```
defimpl Beautify, for: Atom do
  def beautify(t, opts), do: ...
end
```

```elixir
defmodule Beautify.Atom do
  @behaviour Beautify
  @protocol  Beautify
  @for       Atom

  def beautify(t, opts), do: ...
  def __impl__(:target), do: __MODULE__
end
```

Kernel.defprotocol/2 defines a module with the protocol name, in which:

* Opts out from Kernel.def/2, ...

* Imports Protocol.def/1

* Defines the dispatcher impl_for/1

* Enables :debug_info

* ...

```
defprotocol Beautify do
  def beautify(t, opts)
end
```

```elixir
defmodule Beautify do
  @compile :debug_info

  Kernel.def beautify(t, opts) do
    impl_for!(t).beautify(t, opts)
  end

  Kernel.def impl_for(t) when is_atom(t) do
    case impl_for?(Beautify.Atom) do
      true -> Beautify.Atom.__impl__(:target)
      false -> any_impl_for()
    end
  end
end
```

What is the point of protocol consolidation?

impl_for? is expensive, but needed if you do not know in advance all existing protocol implementations

But when a project has been compiled, you know

What does protocol consolidation do?

Detects all protocols in .beam files including Elixir's, like Enumerable

Detects all implementations in .beam files

Rewrites impl_for/1 in each protocol:

* As many clauses as implementations, no more, no less

* Each one returns the target right away, no longer need to call impl_for?/1

* Final fallback clause, if needed

```
# Original dispatch for atoms, always
# generated.
Kernel.def impl_for(t) when is_atom(t) do
  case impl_for?(Beautify.Atom) do
    true -> Beautify.Atom.__impl__(:target)
    false -> any_impl_for()
  end
end
```

```elixir
# Consolidated dispatch for atoms, only
# present if there is an implementation
# for them.
Kernel.def impl_for(t) when is_atom(t) do
  Beautify.Atom
end
```

Rewrites the __protocol__(:consolidated?) clause to return true (for the predicate Protocol.consolidated?/1)

Finally, removes the :debug_info flag, unless globally set

Mix writes the new .beam for the protocol module to *build/{MIX_ENV}/consolidated*

Rewriting? WTF?_

```
:beam_lib.chunks(
  filename,
  [:abstract_code, ...],
  ...
)
```

```
defp builtin_clause_for(mod, guard, protocol, line) do
  {:clause, line,
    [{:var, line, :x}],
    [[{:call, line,
        {:remote, line,
          {:atom, line, :erlang},
          {:atom, line, guard}},
        [{:var, line, :x}],
  }]],
    [{:atom, line, load_impl(protocol, mod)}]}
end
```

The Erlang AST is rewritten in memory, compiled with compile:forms/2, and the new .beam file written to disk

Thanks José!

Thanks all!

A Tour of the Elixir Source Code

Xavier Noria
@fxn

ElixirConf EU 2017