elixir

cheat sheet

elixir-lang.org

v1.2

Updated 1/4/2016

Command line

elixir [options] file.ex/file.exs

iex

iex -S script (e.g., iex -S mix)

iex --name local

iex --sname fully.qualified.name
    --cookie cookie.value or use
    $HOME/.erlang.cookie

mix *new / run / test / deps / etc.*

mix.exs specifies build details

iex Commands

#iex:break	— back to prompt
c "filename.exs"	— compile
r Module	— reload
h function_name	— help
i var	— display type info
v [n]	— session history

Operators

=== !== and or not	(strict)
== != && || !	(relaxed)
>, >=, <, <=	
+, -, *, /	(float)
div, rem	(integer)
binary1 <> binary2	(concat)
list1 ++ list2	(concat)
list1 -- list2	(set diff)
a in enum	(membership)
^term	(no reassign)

Types

Integer	1234 0xcafe 0177 0b100 10_000
Float	1.0 3.1415 6.02e23
Atom	:foo :me@home :"with spaces"
Tuple	{ 1, 2, :ok, "xy" } *(like array)*
List	[1, 2, 3] *(like linked list)*
	[head | tail]
	'abc'
	''' here doc '''
	(see Enum and List modules)
Keyword List (can duplicate keys)	[a: "Foo", b: 123]
Map (no duplicate keys)	%{ key => value, key => value }
Binary	<< 1, 2 >> or "abc"
	""" here doc """
	"#{interpolated}"
	<< name::prop-prop-prop … >>
	binary, bits, bitstring, bytes, float, integer, utf8, utf16, utf32, size(n), signed/unsigned, big/little native
Truth	true, false, nil
Range	a..b

Anonymous Functions

```
fn parms [guard] -> body
   parms [guard] -> body
end
```

call with func.()

Shortcut: &(...)
    &1,&2 as parameters

Named Functions

(Only in modules, records, etc)

```
def name(parms) [guard] do
  expression
end
```

def name(parms) [guard], do: expr

Default params: parameter \\ default

defp for private functions

Multiple heads with different params and/or guards allowed.

Capture a function with:
    &mod_name.func_name/arity
    (Can omit mod_name)

Modules

```
defmodule mod_name do
  @moduledoc "description"
  @doc "description"
  function/macro
end
```

require Module *(used for macros)*

use Module
    calls Module.__using__

import Module [,only:|except:]

alias mod_path [, as: Name]

alias mod_path.{ Name, Name, Name... }

@attribute_name value

Call Erlang using:
    :module.function_name

Guard Clause

Part of pattern match

when expr

where operators in expr are limited to:

==, !=, ===, !==, >, <, <=, >=,
or, and, not, !, +, -, *, /, in,
is_atom, is_binary, is_bitstring, is_boolean,
is_exception, is_float, is_function,
is_integer, is_nil, is_list, is_number, is_pid,
is_port, is_reference, is_tuple,
abs(num), bit_size(bits), byte_size(bits),
div(num,num), elem(tuple, n), float(term),
hd(list), length(list), node(),
node(pid|ref|port), rem(num,num),
round(num), self(), tl(list), trunc(num),
tuple_size(tuple)
<> and ++ `(left side literal)`

Comprehensions

for generator/filter [, into: value], do: expr

Generators are:
    pattern <- list

With binaries as:
    for << ch <- "hello" >>, do: expr

do: vs do/end

something do
  expr
end
	something, do: expr

else, rescue, try, ensure also generate
keyword args, and are then compiled

Pragmatic Bookshelf

Maps

```
%{ key => value, key => value }
value = map[key] (can return nil)
value = map.key (if key is atom; can fail)
newmap = %{ oldmap | key => newval }
or
newmap = Map.put(oldmap, key, newval)
Map.put_new/3 to add a key
```

Protocols

```
defprotocol module.name do
 @moduledoc description
  @only [list of types] (optional)
 def name(parms)
end
defimpl mod.name, for: type do
   @moduledoc description
   def name(type, value) do
       expr
   end
end
```

Allowed types:
    Any Atom BitString Function List
    Number PID Port Record Reference

Regexp

```
~r{pattern}opts
```

f	match beg of ml string
g	use named groups
i	case insensitive
m	^ and $ match each line in multiline
r	reluctant (not greedy)
s	. matches newline
u	Unicode patterns
x	ignore whitespace and comments

Processes

```
pid = spawn(anon_function)
pid = spawn(mod, func, args)
(also spawn_link)

receive do
 { sender, msg, … } ->
   send sender { :ok, value }
 after timeout ->
   …
 end
```

Pipelines

```
expr |> f1 |> f2(a,b) |> f3(c)
```

(same as)

```
f3(f2(f1(expr), a, b), c)
```

Control Flow

```
if expr do
   exp
else

   exp
end
```

```
unless expr do
   exp
else

   exp
end
```

```
case expr do
 match [guard] -> exp
 match [guard] -> exp
 …
end
```

```
cond do
   bool -> exp
   bool -> exp
end
```

```
with match <-  exp,
        match <- exp,
        …,
        do: exp
```

executes all exp until a match fails (and is returned), or the do: is run.

Metaprogramming

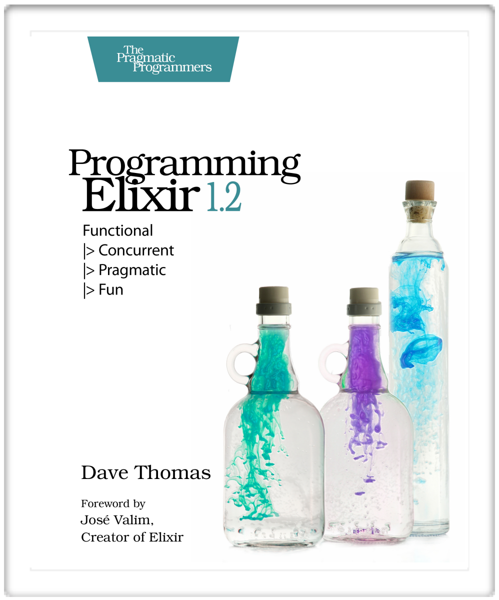
```
defmacro macroname(parms) do
           parms are quoted args
           return quoted code which
           is inserted at call site
end
```

```
quote do: … returns internal rep.
quote bind_quoted: [name: name]
do: …
```

```
unquote do: … only inside quote, injects
code fragment without evaluation
```

Predefined Names

```
__MODULE__ __FILE__ __DIR__ __ENV__
__CALLER__ (macros only)
```


The Pragmatic Programmers

Programming Elixir 1.2

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

pragprog.com/books/elixir12

Structs

```
defmodule Name do
  defstruct field: default, …
end

%Name{field: value, field: value, …}

new_struct = %{ var | field: new_value }
```

Sigils

~type{ content }

Delimiter: { }, [ ], ( ), / /, | |, " ", or ' '

~S	string (no interpolation)
~s	string (with interpolation)
~C	character list (no interpolation)
~c	character list (with interpolation)
~R	regexp
~r	regexp w/interpolation
~W	words (white space delim)
~w	words w/interpolation

Pragmatic Bookshelf