Extracted from:

Metaprogramming Elixir

Write Less Code,
Get More Done
(and Have Fun!)

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Metaprogramming Elixir

Write Less Code, Get More Done (and Have Fun!)

Chris McCord
(author of the Phoenix framework)

Edited by Jacquelyn Carter

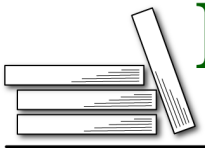
Metaprogramming Elixir

Write Less Code, Get More Done (and Have Fun!)

Chris McCord

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-041-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2015

To my lovely wife, Jaclyn.

It's time to begin our journey to metaprogramming mastery. Ahead lies new Elixir insights and new programming abilities. Perhaps you want to write more productive libraries, build domain-specific languages, or optimize runtime performance. Maybe you simply want to have fun exploring all that Elixir has to offer. If this sounds like you, let's get started!

By now, you're familiar with Elixir; you've experimented with the language and perhaps contributed to a library or two. We're going to take it to the next level by writing code that writes code with macros. Elixir macros are the game-changer. They enable metaprogramming and make it a breeze to write powerful programs.

Code that writes code might sound like a neat trick, but you'll soon see how it forms the basis of Elixir's own construction. Macros open up unique possibilities that simply aren't possible in most languages. We can extend the language with powerful first-class features, save time, and share functionality in fun and productive ways. Used properly, metaprogramming lets us create clear, concise programs that treat source code as building blocks instead of as rote lines of instructions.

We're going to start by covering everything you need to know about Elixir's metaprogramming system before we dive into our advanced exercises.

Let's play.

The World Is Your Playground

Metaprogramming in Elixir is all about extensibility. Have you ever wished your favorite language would adopt that one neat feature? If you're lucky, it might take years to happen. Often it never happens at all. In Elixir, you can introduce new first-class features at will. Take the familiar `while` loop that you find in most languages. It's missing from Elixir, but you can imagine writing one like this:

```
while Process.alive?(pid) do
  send pid, {self, :ping}
  receive do
    {^pid, :pong} -> IO.puts "Got pong"
  after 2000 -> break
  end
end
```

In the next chapter, we make this `while` loop a reality. It doesn't stop there, though. With Elixir, we can define languages with the language, to express all kinds of problems in a natural syntax. This is a valid Elixir program:

```
div do
  h1 class: "title" do
    text "Hello"
  end
  p do
    text "Metaprogramming Elixir"
  end
end
"<div><h1 class=\"title\">Hello</h1><p>Metaprogramming Elixir</p></div>"
```

Elixir makes things like this HTML domain-specific language possible. In fact, we'll create this in just a few chapters. You don't have to understand how these things work just yet—we'll get to that. For now, just remember that macros make all this possible. Code that writes code. Elixir pushes this idea further than you've ever seen.

As with any playground, you need to start small and work your way up to the advanced areas. Metaprogramming can be a difficult concept to grasp, and its use requires a high level of care. Throughout this book, we'll unveil the mystery by going from simple exercises all the way through advanced code-generation tutorials. Before we start writing code, we need to review the two essential concepts of Elixir's metaprogramming system and how they fit together.

The Abstract Syntax Tree

To master metaprogramming, you first have to understand how Elixir code is represented internally by the abstract syntax tree (AST). Most languages you've worked with have an AST, but you're typically not aware of it. When your programs are compiled or interpreted, their source is transformed into a tree structure before being turned into bytecode or machine code. This process is usually masked away, and you never need to think about it.

José Valim, the creator of Elixir, chose to do something very different. He exposed the AST in a form that can be represented by Elixir's own data structures and gave us a natural syntax to interact with it. Having the AST accessible by normal Elixir code lets you do very powerful things because you can operate at the level typically reserved only for compilers and language designers. You interact with Elixir's AST at every step of the metaprogramming process, so let's jump in and find out what it's all about.

Metaprogramming in Elixir revolves around manipulating and inspecting ASTs. You can access the AST representation of any Elixir expression by using the *quote* macro. Code generation relies heavily on `quote`, and we'll be using

it throughout the book to carry out our exercises. Let's use it to return the AST representation of a couple of basic expressions.

Type the following into iex and let's look at the results:

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

iex> quote do: div(10, 2)
{:div, [context: Elixir, import: Kernel], [10, 2]}
```

We can see that the AST representation of `1 + 2` and `div` produced simple data structures in Elixir's own terms. Let that sink in for a moment. You can access the representation of any code you write as an Elixir data structure. Quoting expressions gives you something you've probably never seen from a language before: the ability to peer into the internal representation of your code, within a data structure you already know and understand. This lets you infer meaning, optimize performance, or extend functionality while staying within Elixir's high-level syntax.

With full AST access, we can perform neat tricks during compilation. For example, the `Logger` module in Elixir's standard library can optimize logging by completely removing the expressions from the AST. Let's say we're writing to a file and would like to print the file path in development but ignore the expression in production. We might write something like the following:

```
def write(path, contents) do
  Logger.debug "Writing contents to file #{path}"
  File.write!(path, contents)
end
```

In production, the `Logger.debug` expression would be completely removed from the program. This is because we can interact with the AST during compilation to skip this development-related call. Most languages would have to invoke the `debug` function and waste CPU cycles checking for ignored log levels at runtime, because their source code cannot interact with the underlying AST.

Finding out how `Logger.debug` is able to perform this feat brings us to the next essential ingredient of the metaprogramming process: macros.

Macros

Macros are code that writes code. Their purpose in life is to interact with the AST using Elixir's high-level syntax. This is how `Logger.debug` can perform its optimization tricks while appearing like normal Elixir code.

Macros are used for everything from building Elixir's standard library to serving as core infrastructure of a web framework. In either case, the same metaprogramming rules apply. You don't have to make a decision between complex, performant code or slower, elegant APIs. Elixir macros let you write simple code with high performance. They turn you, the programmer, from language consumer to language creator. No longer are you merely a user of the language. You have access to all the tools and power that José used to write the standard library. He opened the language up for your own extension. Once you experience that level of power, it's hard to go back.

You might think you've largely avoided macros until now, but they've been hiding in plain sight all along. Consider this simple block of code:

```
defmodule Notifier do
  def ping(pid) do
    if Process.alive?(pid) do
      Logger.debug "Sending ping!"
      send pid, :ping
    end
  end
end
```

It might look unremarkable, but we're looking right at four macros. Internally, `defmodule`, `def`, `if`, and even `Logger.debug` are implemented as macros, like most of Elixir's top-level constructs. You can see for yourself by looking up the documentation in `iex`:

```
iex> h if

           defmacro if(condition, clauses)

Provides an if macro. This macro expects the first argument to be a condition
and the rest are keyword arguments.
...
```

You might be wondering what the advantage is of Elixir using macros for its own constructs, since you get by fine in most languages without this structure. The most powerful advantage is that macros allow you to extend the language with your own keywords while using existing macros as building blocks.

The best way to think about metaprogramming in Elixir is to throw away the notion of rigid keywords and opaque language internals. Elixir was designed with extension in mind. The language is open to your exploration and custom features. This is what makes metaprogramming in Elixir so pleasantly natural.