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

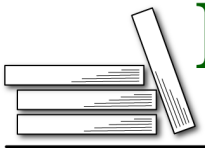
Metaprogramming Elixir

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.

Building an Internationalization Library

Almost all user-facing applications are best served by an internationalization layer where language snippets can be stored and referenced programmatically. Let's use code generation to produce an internationalization library in fewer lines of code than you thought possible. This is the most advanced exercise you've done so far, so let's start by breaking down our implementation into a rubric that you can use to attack complex metaprogramming problems.

Step 1: Plan Your Macro API

The first step of our Translator implementation is to plan the surface area of our macro API. This is often called README Driven Development. It helps tease out our library goals and figure out what macros we need to make them happen. Our goal is to produce the following API. Save this file as i18n.exs.

advanced_code_gen/i18n.exs

```
defmodule I18n do
  use Translator

  locale "en",
    flash: [
      hello: "Hello %{first} %{last}!",
      bye: "Bye, %{name}!"
    ],
    users: [
      title: "Users",
    ]

  locale "fr",
    flash: [
      hello: "Salut %{first} %{last}!",
      bye: "Au revoir, %{name}!"
    ],
    users: [
      title: "Utilisateurs",
    ]
end
```

Eventually we want to be able to call our module like this:

```
iex> I18n.t("en", "flash.hello", first: "Chris", last: "McCord")
"Hello Chris Mccord!"

iex> I18n.t("fr", "flash.hello", first: "Chris", last: "McCord")
"Salut Chris McCord!"

iex> I18n.t("en", "users.title")
"Users"
```

We'll support use Translator to allow any module to have a dictionary of translations compiled directly as t/3 function definitions. At minimum, we need to define a __using__ macro to wire up some imports and attributes, and a locale macro to handle locale registrations. Head back over to your editor, and let's write some code.

Step 2: Implement a Skeleton Module with Metaprogramming Hooks

Our next step is to implement the skeleton of our Translator module by defining the __using__, __before_compile__, and locale macros that we planned when fleshing out the surface area of our API. The skeleton will simply set up the compile-time hooks and module attribute registrations, but delegate the code generation bits to functions to be implemented later. Defining the metaprogramming skeleton first will allow us to structure our module in a way that isolates the advanced code generation to a function. This will keep our implementation clear and reusable.

Create a translator.exs file with the following skeleton API:

advanced_code_gen/translator_step2.exs

```
Line 1  defmodule Translator do
-
-     defmacro __using__(_options) do
-       quote do
5        Module.register_attribute __MODULE__, :locales, accumulate: true,
-                                                        persist: false
-        import unquote(__MODULE__), only: [locale: 2]
-        @before_compile unquote(__MODULE__)
-       end
10    end
-
-     defmacro __before_compile__(env) do
-       compile(Module.get_attribute(env.module, :locales))
-     end
15
-     defmacro locale(name, mappings) do
-       quote bind_quoted: [name: name, mappings: mappings] do
-         @locales {name, mappings}
-       end
20    end
-
-     def compile(translations) do
-       # TBD: Return AST for all translation function definitions
-     end
25  end
```

Just like our accumulated @tests attribute in our Assertion module from the code on page ?, we registered an accumulated @locales attribute on line 5.

Next, we wired up the __before_compile__ hook in our Translator.__using__ macro. On line 13, we added a placeholder to delegate to a compile function to carry out the code generation from our locale registrations, but we left the compile implementation for a later step. Finally, we defined our locale macro that will register a locale name and list of translations to be used by compile in our __before_compile__ hook.

With the accumulated attribute registrations wired up, we have all the necessary information to produce an AST of t/3 function definitions. If you like recursion, you're in for a treat. If not, pay attention and we'll break it down.

Step 3: Generate Code from Your Accumulated Module Attributes

Let's begin the bulk of our implementation by transforming the locale registrations into function definitions within our compile placeholder from Step 2. Our goal is to map our translations into a large AST of t/3 functions. We also need to add catch-all clauses that return {:error, :no_translation}. This will handle cases where no translation has been defined for the provided arguments.

Update your compile/1 function with the following code:

advanced_code_gen/translator_step3.exs

```
Line 1  def compile(translations) do
        translations_ast = for {locale, mappings} <- translations do
          deftranslations(locale, "", mappings)
        end
    5
        quote do
          def t(locale, path, bindings \\ [])
          unquote(translations_ast)
          def t(_locale, _path, _bindings), do: {:error, :no_translation}
   10   end
        end

        defp deftranslations(locales, current_path, mappings) do
          # TBD: Return an AST of the t/3 function defs for the given locale
   15   end
```

On line 1, we defined our compile function to carry out the locale code generation. We used a for comprehension to map the locales into an AST of function definitions and stored the result in translations_ast for later injection. Next, we stubbed a deftranslations call that we'll implement later to define the t/3 functions. Finally, we produced an AST for the caller on lines 6–10 by combining our translations_ast with our catch-all functions.

Before we implement deftranslations, load your implementation in iex and let's check our progress:

```
iex> c "translator.exs"
[Translator]

iex> c "i18n.exs"
[I18n]

iex> I18n.t("en", "flash.hello", first: "Chris", last: "McCord")
{:error, :no_translation}

iex> I18n.t("en", "flash.hello")
{:error, :no_translation}
```

We're on the right track. Any call to I18n.t returns {:error, :no_translation} because we haven't yet generated the functions for each locale. We've confirmed that our catch-all t/3 definitions on line 9 were properly generated. Let's continue by implementing deftranslations to recursively walk our locales and define translation functions.

Fill in your deftranslations function with this code:

advanced_code_gen/translator_step4.exs

```
Line 1  defp deftranslations(locale, current_path, mappings) do
        for {key, val} <- mappings do
          path = append_path(current_path, key)
          if Keyword.keyword?(val) do
5           deftranslations(locale, path, val)
          else
            quote do
              def t(unquote(locale), unquote(path), bindings) do
                unquote(interpolate(val))
10            end
            end
          end
        end
      end

15
      defp interpolate(string) do
        string # TBD interpolate bindings within string
      end

20
      defp append_path("", next), do: to_string(next)
      defp append_path(current, next), do: "#{current}.#{next}"
```

We started by mapping over our translation key value pairs. Within our comprehension on line 4, we first checked whether the value is a keyword list. This would indicate a nested list of translation mappings, just like we saw in our original high-level API.

```
flash: [
  hello: "Hello %{first} %{last}!",
  bye: "Bye, %{name}!"
],
```

The :flash key above points to a nested keyword list of translations. To handle this, we would append "flash" to our accumulated current_path variable, which we handled by an append_path helper function on lines 20–21. Then we continue by recursively calling deftranslations until we encounter a string translation. We used quote on line 7 to generate the t/3 function definitions for each string and unquote to inject the proper current_path, such as "flash.hello", into the function clause. Our t/3 body called a stubbed interpolate function that we'll implement in a moment to take care of placeholder interpolations.

This required only a handful of lines of code, but the recursion can be a little mind-bending. Let's take a break and see where we're at in iex.

```
iex> c "translator.exs"
[Translator]

iex> c "i18n.exs"
[I18n]

iex> I18n.t("en", "flash.hello", first: "Chris", last: "McCord")
"Hello %{first} %{last}!"
```

We're nearly there. Our t/3 functions were correctly generated, and we just need to handle variable interpolation to complete our library. You might be wondering how we can keep track of all this code that we just generated. Like always, Elixir has us covered. When you start generating large amounts of code, it's often necessary to see the final source that is being produced. For this, you use Macro.to_string.