CAMERON PRICE

JANUARY 24, 2016

How to use leex and yecc in Elixir

Leex and Yecc are incredibly powerful if you find yourself needing to parse something. Unfortunately, they also require a little bit of understanding. If you're an elixir developer like me, you may find the DSL they use to describe tokens and grammars to be less than obvious. I spent a couple of days coming to terms with these tools, but since I'm not writing parsers very often, I decided to write this explanation to help me remember how these work for the next time I need them. This is the first part in a two part article. In this article we'll use leex and yecc to parse a very simple grammar and explain how they work together. In the next article, I'll talk about how I used them to create a parser for a more complex grammar, some tricks I learned about how to unit test, etc. Fair warning, I'm not an expert on these tools, just a guy who struggled to use them at first, and eventually came to understand how they work.

If you'd like to follow along, the code is available here:

https://github.com/cameronp/leex_yecc_tutorial

Step 0: Get set up to run these tools.

Mix actually handles this nicely for you. All you need to do is create `src` in your project root. When you put `.xrl` and `.yrl` files in that folder, mix will notice them, recognize that they are leex and yecc files respectively, compile them to .erl files, and then compile those erl files, all automatically. Easy peasy.

Step 1 : Understand what these tools do.

leex is a lexer. It takes your input data, applies the rules you write to identify tokens in the input data, and transform those tokens into something you

deem to be usable, in the form of a list. The list can contain any elements you like, but in this case, you know you're going to be using yecc, so the sensible thing is to transform the tokens into the form that yecc is going to expect, so let's look at that first.

I've written a simple lexer. All this lexer is capable of identifying are ints and floats while ignoring whitespace and commas. Before we look at the code, let's take a look at it in action:

```
iex(1)> :number_lexer.string('12 42 23.24 23')
{:ok, [{:int, 1, 12}, {:int, 1, 42}, {:float, 1, 23.24}, {:int, 1,
23}], 1}
iex(2)>
```

So my lexer takes a char list, and returns a tuple `{:ok, list_of_tokens, line}`. Each token in `list_of_tokens` is of the form `{type, line_number, value}`. This is the format that yecc is going to be expecting.

When we take this output from our lexer, and pass it to our parser, the parser will apply its rules and produce an Abstract Syntax Tree (AST).

3. Let's look at how we built the lexer:

```
%% src/number_lexer.xrl

Definitions.

Whitespace = [\s\t]
Terminator = \n|\r\n|\r
Comma = ,

Digit = [0-9]
NonZeroDigit = [1-9]
NegativeSign = [\-]
Sign = [\+\-]
FractionalPart = \.{Digit}+

IntegerPart = {NegativeSign}?0|{NegativeSign}?{NonZeroDigit}{Digit}*
IntValue = {IntegerPart}
FloatValue = {IntegerPart}{FractionalPart}|{IntegerPart}
{ExponentPart}|{IntegerPart}{FractionalPart}{ExponentPart}

Rules.

{Comma} : skip_token.
```

```
{Whitespace} : skip_token.
{Terminator} : skip_token.
{IntValue} : {token, {int, TokenLine, list_to_integer(TokenChars)}}.
{FloatValue} : {token, {float, TokenLine, list_to_float(TokenChars)}}.

Erlang code.
```

There are three sections to an `.xrl` file, and they are all required. The first section is Definitions. In this section we are defining the patterns we want the lexer to identify as it scans the input data. Each line such as `Whitespace = [\s\t]` defines the term on the left as the regular expression on the right. Note, these are Erlang regexes, which only support a subset of traditional regex, so you may have to get creative. Details in the leex docs. To reference a definition, simply enclose it in {'s, either in the Definitions section, or in the Rules.

The rules are where we actually identify a token, and tell leex what do with it. These rules must be of the form:

```
<pattern> : <result>.
```

The spaces before and after the ':' and the period at the end are required.

is actually Erlang code, so we have a lot of flexibility to produce what we need. Here's the line that creates the token for a float when leex has matched on one:

```
{FloatValue} : {token, {float, TokenLine, list_to_float(TokenChars)}}.
```

All this says is, when you match the pattern described above in FloatValue, produce a token of the form `{<atom>, <line#>, <value>}` where atom is :float (in elixir), line # is provided by leex, and value is computed using Erlang's builtin list_to_float.

For patterns like Whitespace and Comma, we can simply discard them by setting the result to skip_token.

Step 3: The parser.

Before we start on the parser, let's come up with something worth parsing. Let's add the ability to represents lists in our little language, by enclosing ints or floats, separated with ,'s, inside ['s. I'd like to be able to parse `[1,2,3,[2.1,2,2]]` for example. First we need to modify the lexer to recognize the square bracket tokens. We add this line to our Definitions:

```
Bracket = [\[\]]
```

This is going to match on either square bracket, and we'll just pass the bracket (converted to an atom... this is important, it must be an atom or the parser won't recognize it) along to the parser:

```
{Bracket} : {token, {list_to_atom(TokenChars), TokenLine, TokenChars}
}.
```

Now that we have something a bit more interesting to parse, let' s move onto yecc. First consider our little language as a grammar:

```
Document ::
   Value(list)


Value ::
  Int
  Float
  List

List ::
  Value(list)
```

We have Document, which contains one or more Values, and each value can be either an Int, a Float, or a List of values.

Now let's look at the structure of a `.yrl` file which will define our parser.

There are four sections in our `yrl` file.

```
Nonterminals
Terminals
```

```
Rootsymbol
Erlang code.
```

The terminals are the building blocks of the grammar. They cannot be and need not be further decomposed. They are our tokens which were produced by leex, but they must be listed here. Thus:

```
Terminals
int float '[' ']'.
```

The nonterminals are the complex structures we'll be building up with our rules here. The obvious ones are `document`, `value`, and `list`, but we'll add some auxiliary nonterminals in order to properly implement this grammar.

The Rootsymbol represents the top of the AST. This is the first rule the parser will attempt to apply, and it should lead to all the lower level rules which will be evaluated in this context.

Below the Rootsymbol statement, we have all of the rules, each of which is of the form:

```
<nonterminal> -> <pattern> : <result
```

This says, essentially, "When attempting to capture a try to apply the pattern , and if it succeeds, return which represents the parsed nonterminal. That sounds confusing. Let's take a look at how it works for our little grammar:

```
%% src/number_parser.yrl
Nonterminals
document
values value
list list_items.

Terminals
int float '[' ']'.

Rootsymbol document.


document -> values : '$1'.

values -> value : ['$1'].
```

```
values -> value values : ['$1'] ++ '$2'.

value -> int : {int, unwrap('$1')}.
value -> float : {float, unwrap('$1')}.
value -> list : '$1'.

list -> '[' list_items ']' : '$2'.
list_items -> value : ['$1'].
list_items -> value list_items: ['$1'] ++ '$2'.



Erlang code.

unwrap({_,_,V}) -> V.
```

We have set document as our rootsymbol, since it's the top level of the grammar, and we've set int, float, and the brackets as our terminals, since those are the only tokens emitted by our lexer. Then we've set up rules, some of which may look a little odd. Let's start with the rules for `value`.

```
value -> int : {int, unwrap('$1')}.
value -> float : {float, unwrap('$1')}.
value -> list : '$1'.
```

The first line says that if you have an int token, then you can construct a value. The code after the : is what does that construction, so let's break it down. `'$1'` represents the 1st element in the pattern that matched. In this case, there is only one element in the pattern (int), so that's what it represents. Now, that int was a token produced by the lexer, and if you recall, the lexer produces tokens of the form {type_atom, line_num, value}. In this case, we know it's an int, and we don't need the line num, so all we really care about is value. We've written the helper method unwrap, and put it right at the bottom of the file, in the Erlang code section. In this case it's going to return a char_list contain a representation of an integer, and we're going to return a tuple with an int atom, and that char list. We might also convert the char list to an integer here, and just return that. It's a design decision at this point, and entirely up to you.

The next possible rule is float, and it works the same way as int, so should be self explanatory. The final rule is list, and in this case you'll notice we are

not unwrapping the value of list, we're just returning it directly, because it was not created by the lexer, it was built by one of our other rules.

Specifically:

```
list -> '[' list_items ']' : '$2'.
list_items -> value : ['$1'].
list_items -> value list_items: ['$1'] ++ '$2'.
```

This is obviously a bit more complicated. The first rule just says "a list is an open square bracket, followed by a list_items, followed by a closed square bracket, and it's value is the value of the 2nd part of the pattern, ie list_items. This rule serves to enforce the presence of the square brackets.

The next rule handles the case of a list_items with only one element in it. It captures the single value, and returns it in a list (`[$1]`), The second rule recursively captures a value, encloses it in a list, and appends that list to an existing list_items. This is very common pattern for capturing lists of things in yecc, and we apply the same pattern to allow our document to contain a list of values, above:

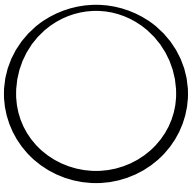
```
values -> value : ['$1'].
values -> value values : ['$1'] ++ '$2'.
```

Let's try putting the lexer and parser together:

```
iex(1)> {:ok, tokens, _} =  :number_lexer.string('1,[2,3]')
{:ok,
 [{:int, 1, 1}, {:"[", 1, '['}, {:int, 1, 2}, {:int, 1, 3}, {:"]", 1,
']'}], 1}
iex(2)> :number_parser.parse(tokens)
{:ok, [{:int, 1}, [int: 2, int: 3]]}
```

It works! In the next article, I'll talk about a couple of tricks I learned while using these tools to represent a more involved grammar.

During the course of this work I found Knut Nesheim's simple calculator app calx and this article on the relops blog to be extremely helpful. I'd recommend taking a look at them both.



99 KUDOS

Tweet

Share 0

@cameronp

SVBTLE

Terms • Privacy • Promise