Erlang - Leex and Yecc

February 8, 2011

New Version Available! (edit: 2014-01-18)

A new version of this blog post is available.
Visit http://relops.com/blog/2014/01/13/leex_and_yecc/
For historical purposes, the original blog post is preserved below…

**Note: THIS IS NOT MY WORK. ** This article was originally posted at *http://hopper.squarespace.com/blog/2008/5/29/leex-and-yecc.html*, but the author seems to have turned off his Squarespace blog. I pulled this from Google cache and am mirroring here until the site is back on, or until the author puts it up somewhere else, as it's one of the few examples online of using leex and yecc.

leex and **yecc** are the Lex and Yacc of the Erlang toolset. Whilst there is ample documentation about using the standard Unix lexer and parser generators, the Erlang equivalents are far less

documented. This article demonstrates how to use **leex** and **yecc** to build a parser in Erlang that can process a subset of SQL-92.

Introduction

The goal of this mini-project is to:

1. Parse fragments of SQL WHERE clauses such as “A in (22,33,44) and B in (‘abc’,’xyz’)”;
2. Apply this as a match specification on key value entries in a map.

So, for example, if your SQL predicate is “A = 22”, then the Erlang code would look like this:

```
TestData = dict:store('A',22,dict:new()),
Spec = selector:parse("A = 22"),
Bool = selector:matches(Spec,TestData).
```

And you would expect the variable Bool to be true. If you changed the predicate to “A > 22”, then would you expect the matches/2 function to return false for the same input data.

In this project we will:

1. Generate a lexer using **leex**;
2. Generate a parser using **yecc**;
3. Combine the two together in our own module;
4. Use our module to test some different input statements.

Using Leex And Yecc

Recently, Robert Virding made the first proper release of the lexical analyzer generator for Erlang called **leex**. Although this is not part of OTP yet, you can download the code from this Trapexit topic.

Strictly speaking, you could build this parser without **leex**, by just using a **yecc** grammar. However, you still need some way to supply **yecc** with a tokenized input stream, because it does not scan raw streams by itself. This can be achieved by using the string/1 function in the erl_scan module. The drawback to this approach is that you have no control over the way that it tokenizes the input. erl_scan tokenizes characters into Erlang tokens, which you then have to interpret in your **yecc** grammar.

By using a lexer grammar, you have full control over the way the input stream is tokenized. The upshot is that you have a more natural fit for your input language and you simplify the parser grammar that is consuming the tokens because you are supplying it with language specific tokens rather than generic Erlang tokens.

Getting The Source

The source for this example project can be downloaded here. It contains the code for **leex**, so you do not need to download that separately. After downloading and unpacking it, you can compile it using the supplied Makefile. If it compiles successfully, you should see the following output:

```
$ make
mkdir -p ebin
# Compile the lexer generator because it is not part of OTP
erlc -I include -o ebin -W0 -Ddebug +debug_info src/**leex**.erl
# Generate the lexer
erl -I -pa ebin -noshell -eval
'**leex**:file("src/selector_lexer",[{outdir,"src"}]), halt().'
Parsing file src/selector_lexer.xrl, contained 10 rules.
NFA contains 62 states, DFA contains 27 states, minimised to 22 states.
Writing file src/selector_lexer.erl, ok
# Generate the parser
erl -I -pa ebin -noshell -eval '**yecc**:file("src/selector_parser"), halt().'
src/selector_parser.yrl: Warning: conflicts: 1 shift/reduce,
0 reduce/reduce
# Compile everything
erlc -I include -o ebin -W0 -Ddebug +debug_info src/*.erl
```

If you have Eunit installed in your OTP installation, you can also run the unit tests using the Makefile:

```
$ make test
....
  All 5 tests successful.
```

Generating The Lexer

To start off with, you need a lexer grammar to generate the lexer. A lexer grammar is essentially a set of rules defining how to tokenize an input stream based on regular expressions. The grammar used in this example looks like this:

```
Definitions.

D   = [0-9]
L   = [A-Za-z]
WS  = ([\000-\s]|%.*)
C   = (<|<=|=|=>|>)

Rules.
```

```
in     : {token,{set,TokenLine,list_to_atom(TokenChars)}}.
or     : {token,{union,TokenLine,list_to_atom(TokenChars)}}.
and    : {token,{intersection,TokenLine,list_to_atom(TokenChars)}}.
{C}    : {token,{comparator,TokenLine,list_to_atom(TokenChars)}}.
'{L}+' : S = strip(TokenChars,TokenLen),
         {token,{string,TokenLine,S}}.
{L}+   : {token,{var,TokenLine,list_to_atom(TokenChars)}}.
{D}+   : {token,{integer,TokenLine,list_to_integer(TokenChars)}}.
[(),]  : {token,{list_to_atom(TokenChars),TokenLine}}.
{WS}+  : skip_token.

Erlang code.

atom(TokenChars) -> list_to_atom(TokenChars)

strip(TokenChars,TokenLen) ->
    lists:sublist(TokenChars, 2, TokenLen - 2).
```

The lexer grammar contains three sections:

1. Definitions - these are the atoms for each character class that can composed into match expressions;
2. Rules - these tell the lexer what series of characters to expect and what token to output when a match occurs;
3. Erlang code - this contains functions that can be used in the rule definitions.

So, for example, if the input stream contains the literal ‘abc’ (including the quotation marks), the rule ‘{L}+’ would fire. This rule would then strip off the leading and trailing quotation marks and pass the token {string,TokenLine,”abc”} to the parser.

The TokenLine variable is just the token’s position in the text and will not be used by our parser grammar. However, this must be

supplied in the lexer grammar because the parser that **yecc** generates expects tokens in the following form:

- Syntactic category of the token;
- Position where it was found in the input;
- The actual terminal symbol that was found in the input.

Once you have your grammar, save it with the ending .xrl to denote that it is a **leex** grammar. In this mini-project, the grammar is called selector_lexer.xrl. In an Erlang shell, you can now use **leex** to generate a scanner:

```
1>**leex**:file(selector_lexer).
Parsing file src/selector_lexer.xrl, contained 10 rules.
NFA contains 62 states, DFA contains 27 states, minimised to 22 states.
Writing file selector_lexer.erl, ok
```

This has converted the lexer grammar into an Erlang module of the same name but with the ending .erl. This can be compiled as a normal Erlang module.

This whole step is done for you if you use the Makefile that is supplied in the project source.

Generating The Parser

Now that you have an input stream that is tokenized in the way you want it, you can create your parser grammar. A grammar for **yecc** consists of five components:

1. Nonterminals - these are syntactic elements that are composed of terminal systems in various patterns;
2. Terminals - these are indivisible symbols that form building blocks for composed structures. These are the tokens supplied by the lexer;
3. Rootsymbol - this is the root of the syntax tree, i.e. the contextual starting point for interpreting the input symbols;
4. Rules - these tell the parser how to classify and structure the tokens it is consuming. A rule has a name, the pattern it matches and the action to be taken when the rule fires. These return the final data structures to the application, so this is the point where you can perform any contextual transformations that are necessary;
5. Erlang code - as with the lexer grammar, this is where you can define functions to abbreviate the actions defined in the rules.

In our example, we are going to group the input tokens into constructs called predicates. An example predicate is “A = 22”, which syntactically consists of a variable, a comparator and an integer. In the subset of SQL, predicates can be joined together using “and” and “or”. In this case, an intersection or a union of predicates is built, depending on which conjunctive joined them together.

For example, the SQL statement “A = 22 or B < 10” converts syntactically to a union of two predicates, “A = 22” and “B < 10”.

If you look at the following grammar you will see that “A = 22” matches the rule “predicate -> var comparator element” and “A =

22 or B < 10” matches the rule “predicates -> predicate union predicate”:

```
Nonterminals
predicates predicate list element elements.

Terminals '(' ')' ','
atom var integer string set union intersection comparator.

Rootsymbol predicates.

predicates -> predicate : '$1'.
predicates -> predicate union predicate : {union, '$1', '$3'}.
predicates -> predicates union predicate : {union, '$1', '$3'}.

predicates -> predicate intersection predicate : \
              {intersection, '$1', '$3'}.

predicate -> var set list : \
            {predicate, {var, unwrap('$1')}, memberof, '$3'}.

predicate -> var comparator element : \
            {predicate, {var, unwrap('$1')}, unwrap('$2'), '$3'}.

list -> '(' ')' : nil.
list -> '(' elements ')' : {list,'$2'}.

elements -> element : ['$1'].
elements -> element ',' elements : ['$1'] ++ '$3'.
element -> atom : '$1'.
element -> var : unwrap('$1').
element -> integer : unwrap('$1').
element -> string : unwrap('$1').

Erlang code.

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

Now that you have a parser grammar, you can use **yecc** to generate a parser. **Yecc** grammars are saved with the ending .yrl. In our mini-project the grammar file is called selector_parser.yrl. In an Erlang shell, execute the following command:

```
2>**yecc**:file(selector_parser).
selector_parser.yrl: Warning: conflicts: 1 shift/reduce, 0 reduce/reduce
{ok,"selector_parser.erl"}
```

This has generated a parser with the same name as the base name of the grammar but with the ending .erl. This can now be compiled as a normal Erlang module. As with the lexer grammar, this step is also performed automatically using the Makefile.

Combining The Lexer And The Parser

Combining the generated lexer and parser to process an input string is a straightforward step. Because we are dealing with a very short input text that can be easily held in memory, we can use the string/1 function that **leex** has generated. Parsing an input string is as simple as:

```
{ok,Tokens,EndLine} = ?LEXER:string("A = 22 or B < 10"),
{ok, ParseTree} = ?PARSER:parse(Tokens)
```

where ?LEXER and ?PARSER are the names of the modules generated by **leex** and **yecc** respectively. The value of the ParseTree in this example is:

```
{union,{predicate,{var,'A'},'=',22},
       {predicate,{var,'B'},'<',10}}
```

Should you not want to read the input from a string, but prefer to read from a stream, you can use the following code:

```
parse(FileName) ->
    {ok, InFile} = file:open(FileName, [read]),
    Acc = loop(InFile,[]),
    file:close(InFile),
    selector_parser:parse(Acc).

loop(InFile,Acc) ->
    case io:request(InFile,{get_until,prompt,?LEXER,token,[1]}) of
        {ok,Token,EndLine} ->
            loop(InFile,Acc ++ [Token]);
        {error,token} ->
            exit(scanning_error);
        {eof,_} ->
            Acc
    end.
```

This uses the io:request/2 call that uses the lexer to tokenize the input stream. The use of the *token* flag indicates that you want read the stream on a token-by-token basis. If you want to read all tokens in the current line, then use the *tokens* flag instead. This will return a list of tokens. This is an undocumented feature in OTP, probably because it is intended for internal usage only, so use it with caution because the interface may change.

Using The Parse Tree

Once you have your parse tree, you can use it to process data in your application. The structure on the parse tree depends on how you write the rules in the parser grammar. Obviously you can do more or less inside the grammar or in your application code to achieve a data structure that is useable by your application. That is a pure design decision.

If you recall from the introduction to this mini-project, we wanted to:

1. Parse a mini SQL statement to form a match specification object;
2. Perform a match operation using this specification against a key value map.

The code for the first step is straightforward and has been illustrated in the preceding section:

```
parse(String) ->
    {ok,Tokens,EndLine} = ?LEXER:string(String),
    {ok, Spec} = ?PARSER:parse(Tokens),
    Spec.
```

The code that performs the second step looks like the following:

```
matches({predicate,{var,Var},Comparator,Critereon}, Props) ->
    case dict:is_key(Var,Props) of
        false ->
            false;
        true ->
            Value = dict:fetch(Var,Props),
            compare(Comparator, Value, Critereon)
    end;
matches({intersection,P1,P2}, Props) ->
    matches(P1,Props) andalso matches(P2,Props);
matches({union,P1,P2}, Props) ->
    matches(P1,Props) or matches(P2,Props);
matches(_,_,_) -> false.
compare(memberof, Value,{list,List}) -> lists:member(Value,List);
compare('=', Value,Critereon) -> Value == Critereon;
compare('>', Value,Critereon) -> Value > Critereon;
compare('<', Value,Critereon) -> Value < Critereon;
compare('>=', Value,Critereon) -> Value >= Critereon;
compare('=<', Value,Critereon) -> Value =< Critereon;
compare(_,_,_) -> false.
```

Here you can see that we are using standard Erlang pattern matching that depends on the specification that is parsed from the SQL statement. Each predicate has a variable that it needs to compare against a specific value in a specific way. It retrieves the value of the variable to be tested from the map. If there is no value under that key in the map, it returns false because it cannot perform a comparison.

After it has the test value, it performs a comparison using the comparator specified by the parse tree. In this example we have the following comparators: memberof, =, >, <, >=, =<. The value that the test value is compared with is specified by the Critereon variable.

Testing The Application

There is a complete test suite based on Eunit in the project source called selector_test.erl. For example, to test an SQL “and” clause, you can use the following:

```
and_test() ->
    String = "A in (22,33,44) and B in ('abc','xyz')",
    Spec = selector:parse(String),
    TestData0 = dict:store('A',33,dict:new()),
    TestData1 = dict:store('B',"abc",TestData0),
    ?assert(selector:matches(Spec,TestData1)).
```

This shows the usage of the assert macro from Eunit that evaluates the boolean return value from matches/2. You can find more test examples in the test module.

Outlook

The goal of this article was to demonstrate the use of **leex** and **yecc** in a practical context. The input grammar was kept simple in order to keep the focus on the overall process flow from an end-to-end perspective. Hopefully, more examples of more complicated parser generation will be published in the future.

New Version Available! (edit: 2014-01-18)

A new version of this blog post is available.
Visit http://relops.com/blog/2014/01/13/leex_and_yecc/
For historical purposes, the original blog post is preserved below…

Back
