[Ragel](#_cm1rzoz8kove)

[package ragel](#_6rqqx3ijlu49)

[Modelling State Machines with Ragel](#_3nfbeurlc27n)

[A simple intro to writing a lexer with Ragel](#_3p5vsp9rh2y9)

[Ragel Parsers](#_vnglqc1ht7u6)

[Sql.rl, simple SQL parser using ragel](#_ni505a8j5szq)

[Building tokenizers with Ragel](#_lzj950c9gucd)

[A Hello World for Ruby on Ragel 6.0](#_bcqbpw3yxfi9)

[===== Ragel State Charts =====](#_wgtd8c5h6dc)

[Specifying Servers With Ragel](#_dqoyrcaqvknc)

[Why Ragel Is Cool](#_49ekhttmitq7)

[Ragel In Mongrel](#_1gw15dobjn0j)

[Enter Utu](#_pdpikrec0m85)

[State Machines In Utu](#_y2pbv7cq3fwv)

[State Machines](#_h7wal4dqwbtf)

[State Machines Are Classes (kinda)](#_rfv8rn3d2wgg)

[State Machines Are Explicit](#_796p2sbtl5xu)

[Using State Machines In Protocols](#_jispgp3tv0av)

[Utu Connections and Hubs](#_k60psjyzlm1l)

[Utu Hub State](#_cesu58gbpaa9)

[Actions In Ragel](#_4uwnh58256yj)

[Generating The Code](#_lfohdbmnnwjw)

[Unit Testing The Hub](#_58s6gwn99vcy)

[Utu Connection State](#_4l6mg3kp91av)

[Unit Testing Connection State](#_8lqbnf9h7e8j)

[Next Steps](#_uui3ikjxs6xh)

[===== Building tokenizers with Ragel =====](#_r7wm8byihmnl)

[Why Ragel?](#_hn7eds6qdyef)

[How it works](#_d7j60r2ye5ez)

[The input specification](#_rjbreos6601)

[A Ruby debugger](#_z1g3as9wogdl)

[A C binary](#_7r9q5aw78q0o)

[Parting thoughts](#_lfkc9rxt0y6w)

<https://www.google.com/search?num=60&client=ubuntu&espv=2&q=ragel+sql>

# Ragel

## package ragel

<https://godoc.org/code.google.com/p/biogo.ragel>

import "code.google.com/p/biogo.ragel"

Package ragel provides helper functions and types for building ragel-based parsers.

<http://www.reddit.com/r/golang/comments/3ai2h4/re2dfa_regular_expressions_into_finite_state/csctj39>

<http://www.colm.net/open-source/ragel/>

I spent the time to create a scanner for a toy language so I would have the framework ready for when I needed or for real. <https://github.com/dgryski/dpc/blob/master/lexer.rl>

## Modelling State Machines with Ragel

<https://speakerdeck.com/nelstrom/modelling-state-machines-with-ragel>

by [nelstrom](https://speakerdeck.com/nelstrom)

## A simple intro to writing a lexer with Ragel

<http://thingsaaronmade.com/blog/a-simple-intro-to-writing-a-lexer-with-ragel.html>

It seems that there is a fair variety of tools designed to make writing [Lexers, Scanners and Tokenizers](http://en.wikipedia.org/wiki/Lexical_analysis) easier, but [Ragel](http://www.complang.org/ragel/) has a reputation for being simple and consistently producing the fastest final code. This is a short and simple intro to Ragel for a common use-case: writing a Lexer for a programming language.

Start by making sure you have Ragel installed. This process varies for each OS so I'm not going to cover it in detail. On OS X it's as easy as installing [MacPorts](http://www.macports.org/) and then opening a terminal and typing 'sudo port install ragel'. You should now be able to use Ragel via the command-line.

Next you need to decide on your 'host language'. The host language is preferably the same language that the rest of your project is written in, though many people opt to use C for their lexer because of the dramatic increase in speed that it provides. Ragel supports a number of host languages including: C, C++, Objective-C, D, Java and Ruby. For the purpose of this intro we will be using Ruby as it won't obscure the Ragel-specific code as much as other languages would.

First let's create a blank file called 'lexer.rl' and then define an empty state machine inside it. Ragel state machines are defined inside blocks demarcated with '%%{ }%%' like so:

|  |  |
| --- | --- |
| 12345678 | # lexer.rl  %%{    machine test\_lexer;    }%%    %% write data; |

[**view raw**](https://gist.github.com/aarongough/474637/raw/ragel_lexer_blank.rl)[**ragel\_lexer\_blank.rl**](https://gist.github.com/aarongough/474637#file-ragel_lexer_blank-rl) hosted with ❤ by **[GitHub](https://github.com/)**

In this Ragel code block we are defining a blank state machine called 'test\_lexer', then telling Ragel that the state machine should be compiled in this file using the 'write data' directive. The ability to define where the state machine should be compiled is more useful once we start defining machines that span multiple files. Regions outside of '%%{ }%%' blocks and lines that do not start with '%%' are assumed to be written in the host language.

In Ragel the syntax for defining a lexer/scanner differs from that required for creating a normal state-machine and looks like this:

|  |  |
| --- | --- |
| 12345678910 | %%{    machine test\_lexer;    <scanner\_name> := |\*  <token\_description> => {<action>};  <token\_description> => {<action>};  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474638/raw/ragel_lexer_scanner_example.rl)[**ragel\_lexer\_scanner\_example.rl**](https://gist.github.com/aarongough/474638#file-ragel_lexer_scanner_example-rl) hosted with ❤ by **[GitHub](https://github.com/)**

If the 'scanner\_name' is 'main' then it is automatically run when our state machine is executed. An 'action' is a section of host-language code that is executed whenever the token represented by 'token\_description' is found. It could be something as simple as printing out the token, or it could be code to affect the state of something external like a parser.

Token descriptions can either be a string literal (like 'keyword') or a regular expression literal (like [0-9]\*). However it is much easier to read our final scanner if we store our token descriptions inside variables with useful names like so:

|  |  |
| --- | --- |
| 12345678910111213 | %%{    machine test\_lexer;    integer = <token\_description>;  float = <token\_description>;    main := |\*  integer => {<action>};  float => {<action>};  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474642/raw/ragel_lexer_scanner_example_w_name_tokens.rl)[**ragel\_lexer\_scanner\_example\_w\_name\_tokens.rl**](https://gist.github.com/aarongough/474642#file-ragel_lexer_scanner_example_w_name_tokens-rl) hosted with ❤ by **[GitHub](https://github.com/)**

We can compile our state machine by passing the file to Ragel like so:

|  |  |
| --- | --- |
| 1 | ragel -R lexer.rl |

[**view raw**](https://gist.github.com/aarongough/474643/raw/ragel_compile_command.sh)[**ragel\_compile\_command.sh**](https://gist.github.com/aarongough/474643#file-ragel_compile_command-sh) hosted with ❤ by **[GitHub](https://github.com/)**

The '-R' switch tells Ragel that we're using Ruby as our host language. This command will produce an output file called 'lexer.rb' which we can run by itself or incorporate into a larger project.

In order to actually run a state machine we need to encapsulate several Ragel directives (lines starting with '%%') into a function definition like this:

|  |  |
| --- | --- |
| 12345678910 | def run\_lexer(data)  data = data.unpack("c\*") if(data.is\_a?(String))  eof = data.length  token\_array = []    %% write init;  %% write exec;    puts token\_array.inspect  end |

[**view raw**](https://gist.github.com/aarongough/474644/raw/ragel_lexer_run_lexer.rb)[**ragel\_lexer\_run\_lexer.rb**](https://gist.github.com/aarongough/474644#file-ragel_lexer_run_lexer-rb) hosted with ❤ by **[GitHub](https://github.com/)**

Our 'run\_lexer' function serves several purposes: it unpacks our data string into an array of ordinal values, it tells the state machine how long our data is using the 'eof' variable, it creates a blank array for our tokens, initializes the state machine with 'write init', executes it with 'write exec' and then tells us about the tokens found by outputting a human-readable version of our token\_array to stdout.

Now that these pieces are in place we're ready to start defining our lexer. The first step is defining our token description for an integer:

|  |  |
| --- | --- |
| 1234567 | %%{    machine test\_lexer;    integer = ('+'|'-')?[0-9]+;    }%% |

[**view raw**](https://gist.github.com/aarongough/474645/raw/ragel_lexer_w_token.rl)[**ragel\_lexer\_w\_token.rl**](https://gist.github.com/aarongough/474645#file-ragel_lexer_w_token-rl) hosted with ❤ by **[GitHub](https://github.com/)**

We have to define a description for each token that may be present in the source data. It's worth noting that in Ragel encountering an unknown pattern/token counts as an error, so we will have to later define a pattern for everything that could be in the grammar we're lexing, including whitespace, even though we're not actually doing anything with it.

Now we will create a basic scanner definition that looks for an integer, but does nothing with it:

|  |  |
| --- | --- |
| 1234567891011 | %%{    machine test\_lexer;    integer = ('+'|'-')?[0-9]+;    main := |\*  integer;  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474647/raw/ragel_lexer_definition.rl)[**ragel\_lexer\_definition.rl**](https://gist.github.com/aarongough/474647#file-ragel_lexer_definition-rl) hosted with ❤ by **[GitHub](https://github.com/)**

Next we'll create our first action based on a token:

|  |  |
| --- | --- |
| 1234567891011 | %%{    machine test\_lexer;    integer = ('+'|'-')?[0-9]+;    main := |\*  integer => { puts "Integer" };  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474649/raw/ragel_lexer_w_action.rl)[**ragel\_lexer\_w\_action.rl**](https://gist.github.com/aarongough/474649#file-ragel_lexer_w_action-rl) hosted with ❤ by **[GitHub](https://github.com/)**

Sections of Ragel code inside braces like '{ puts "Integer" }' are actually written in the host language, in this case Ruby. So at this point if we were to run the lexer against a string like "190" you would simply see "Integer" written to stdout.

If we want to do something more useful with the token we have captured then we need to use the 'ts' and 'te' variables that are defined by the Ragel scanner. 'ts' stands for 'token start', while 'te' stands for 'token end'. These represent the indices from our data array that match the start and end of the current token. In Ruby we could use them like so:

|  |  |
| --- | --- |
| 1234567891011 | %%{    machine test\_lexer;    integer = ('+'|'-')?[0-9]+;    main := |\*  integer => { puts "Integer: " + data[ts..te].pack("c\*") };  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474652/raw/ragel_lexer_token_extraction.rl)[**ragel\_lexer\_token\_extraction.rl**](https://gist.github.com/aarongough/474652#file-ragel_lexer_token_extraction-rl) hosted with ❤ by **[GitHub](https://github.com/)**

If we were to run this scanner against a string like "-999" we would see "Integer: -999" on stdout. This shows that it's pretty easy to capture the data we care about, from here we just need to devise a method for storing this data. Let's setup a function called 'emit' that will append the current token to an array:

|  |  |
| --- | --- |
| 123 | def emit(token\_name, data, target\_array, ts, te)  target\_array << {:name => token\_name.to\_sym, :value => data[ts...te].pack("c\*") }  end |

[**view raw**](https://gist.github.com/aarongough/474654/raw/ragel_lexer_emit_token.rb)[**ragel\_lexer\_emit\_token.rb**](https://gist.github.com/aarongough/474654#file-ragel_lexer_emit_token-rb) hosted with ❤ by **[GitHub](https://github.com/)**

And then incorporate it into the action associated with the integer token:

|  |  |
| --- | --- |
| 12345678910111213 | %%{    machine test\_lexer;    integer = ('+'|'-')?[0-9]+;    main := |\*  integer => {  emit(:integer\_literal, data, token\_array, ts, te)  };  \*|;    }%% |

[**view raw**](https://gist.github.com/aarongough/474655/raw/ragel_lexer_action_emmit.rl)[**ragel\_lexer\_action\_emmit.rl**](https://gist.github.com/aarongough/474655#file-ragel_lexer_action_emmit-rl) hosted with ❤ by **[GitHub](https://github.com/)**

You'll notice the action definition is now spread across multiple lines for readability. Running this lexer against a string like "-101" will now produce an array like:

|  |  |
| --- | --- |
| 1 | [{:name => :integer\_literal, :value => "-101" }] |

[**view raw**](https://gist.github.com/aarongough/474656/raw/ragel_lexer_emit_example.rb)[**ragel\_lexer\_emit\_example.rb**](https://gist.github.com/aarongough/474656#file-ragel_lexer_emit_example-rb) hosted with ❤ by **[GitHub](https://github.com/)**

So you can see we now have everything in place to build out our lexer to handle the full target grammar, which we can do quite simply by adding further token descriptions and their associated actions, the code for our full lexer will look like this:

|  |  |
| --- | --- |
| 12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152 | =begin  %%{    machine simple\_lexer;    integer = ('+'|'-')?[0-9]+;  float = ('+'|'-')?[0-9]+'.'[0-9]+;  assignment = '=';  identifier = [a-zA-Z][a-zA-Z\_]+;    main := |\*    integer => {  emit(:integer\_literal, data, token\_array, ts, te)  };    float => {  emit(:float\_literal, data, token\_array, ts, te)  };    assignment => {  emit(:assignment\_operator, data, token\_array, ts, te)  };    identifier => {  emit(:identifier, data, token\_array, ts, te)  };    space;    \*|;    }%%  =end    %% write data;  # %% this just fixes our syntax highlighting...    def emit(token\_name, data, target\_array, ts, te)  target\_array << {:name => token\_name.to\_sym, :value => data[ts...te].pack("c\*") }  end    def run\_lexer(data)  data = data.unpack("c\*") if(data.is\_a?(String))  eof = data.length  token\_array = []    %% write init;  %% write exec;    puts token\_array.inspect  end |

[**view raw**](https://gist.github.com/aarongough/474659/raw/ragel_lexer.rb)[**ragel\_lexer.rb**](https://gist.github.com/aarongough/474659#file-ragel_lexer-rb) hosted with ❤ by **[GitHub](https://github.com/)**

You'll notice that our last token description 'space' is not defined anywhere, that's because it is a token description built into Ragel. There is no action associated with this token because we don't want to do anything with it, we simply need to define it to say that whitespace is valid in our target grammar.

We can run our new lexer against data very easily like so:

|  |  |
| --- | --- |
| 12 | run\_lexer("test = -100")  #=> [{:value=>"test", :name=>:identifier}, {:value=>"=", :name=>:assignment\_operator}, {:value=>"-100", :name=>:integer\_literal}] |

[**view raw**](https://gist.github.com/aarongough/474661/raw/rage_lexer_run_example.rb)[**rage\_lexer\_run\_example.rb**](https://gist.github.com/aarongough/474661#file-rage_lexer_run_example-rb) hosted with ❤ by **[GitHub](https://github.com/)**

This obviously provides a pretty solid base against which we can start implementing something more serious. If you're interested in learning more about Ragel and it's possible applications check out the [Ragel site](http://www.complang.org/ragel/) and the [Ragel user guide](http://www.complang.org/ragel/ragel-guide-6.6.pdf).

# Ragel Parsers

## [Sql.rl](https://gist.github.com/umjasnik/1524986), simple SQL parser using ragel

<https://gist.github.com/umjasnik/1524986>

simple SQL parser using ragel - get all tables that are referenced in a SQL statement

## Building tokenizers with Ragel

<https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel>

## A Hello World for Ruby on Ragel 6.0

<http://www.devchix.com/2008/01/13/a-hello-world-for-ruby-on-ragel-60/>

January 13, 2008 by [**Ana Nelson**](http://www.devchix.com/author/ana-nelson/)

This is an updated version of [this tutorial](http://www.devchix.com/2007/12/13/a-hello-world-for-ruby-on-ragel/). This updated version is compatible with Ruby 1.8 and Ruby 1.9, and Ragel 6.0. A version of this tutorial in Portuguese is available [here](http://artigos.waltercruz.com/a-hello-world-for-ruby-on-ragel/).

By the end of this post, you’ll be able to turn a simple string “h” into the much longer and more interesting string “hello world!” using the magic of Ragel, all from the comfort of Ruby. Ragel is a very powerful state machine compiler and parser generator, which is at the heart of software like Mongrel and Hpricot. It’s able to generate C, C++, Objective-C, D, Java or Ruby code.

Ragel has excellent documentation provided by the author. My goal here is just to give you some context so that the documentation “sticks” when you read it, and to give you a working example which you can modify as you explore Ragel’s functionality. If you want to skip ahead, the full example is[here](http://ananelson.com/blog/2008/01/simple_state_machine.rl).

The first step, of course, is installing Ragel. The [Ragel home page](http://www.complang.org/ragel/) has a Download section which lists ports for various platforms. If you already have Ragel installed, check that the version is 6.0 or higher. You can also compile and install Ragel from the source. Even if you don’t want to install from source it’s handy to have a copy of it to get some examples to play with. The subversion repository for Ragel is located here:

http://svn.complang.org/ragel/trunk/

As usual the test/ directory is your friend, also check out the examples/ directory. As per [this thread](http://www.complang.org/pipermail/ragel-users/2007-June/000252.html), try searching for “LANG: ruby”.

When writing Ragel code, you create a file with a .rl extension. The .rl file is written in the “host” language, in this case Ruby, and the Ragel machine specification is embedded within the Ruby code using special delimiters. There’s actually no obligation to specify a state machine, so a perfectly valid .rl file is:

puts "hello world"

Don’t worry, I’m going to do a better Hello World than that, but this is a good place to start. To convert this .rl file into an executable .rb file, use the “ragel” command with a -R flag to indicate that you want Ruby code.

ragel -R hello\_world.rl

This will create a file entitled hello\_world.rb with the following contents:

# line 1 "hello\_world.rl"  
puts "hello world"

I’ll, er, leave executing that file as an exercise for the diligent student.

Ragel actually does this conversion in 2 stages. First it creates an XML file, then converts the XML to Ruby. If you want to view this intermediate XML then you can pass the -x flag in addition to the -R flag.

ragel -R -x simple\_state\_machine.rl > simple\_state\_machine.xml

Now, let’s write some actual Ragel. Start a new .rl file or [download the example](http://ananelson.com/blog/2008/01/simple_state_machine.rl) and read along. We’re going to create a machine which prints “hello world!” when it’s passed the string “h”, and does nothing otherwise. To indicate to the ragel compiler that we are writing instructions for it, and not Ruby code, we need to place our Ragel code within double-percent-sign-curly-brackets %%{ and }%% , or you can enter a single line instruction by just typing %%. (See page 6 of the User Guide.) Here’s our state machine specification:

%%{  
 machine hello;  
 expr = "h";  
 main := expr @ { puts "hello world!" } ;  
}%%

A quick overview of what’s happening here. The name of this machine is “hello” (Ragel makes us name it). It recognizes a single token, the string “h”. When it encounters that token, it performs (in Ruby) the action:

puts "hello world"

Now, if you were to run the ragel command on this file it would compile, but you would basically end up with a blank Ruby file. We have only specified the machine, we also have to tell Ragel to actually translate this machine into Ruby code using Ragel’s write statements. The first write statement we need to add is

%% write data;

If you add this line after the state machine definition block, it will compile, as long as you remember to add a blank line afterwards. (After you’ve worked with parsers for a while you come to appreciate newlines in a whole new way.) After adding this line and compiling, you should have a rather significant Ruby file with lots of class << self statements all generated by Ragel. You don't need to study this code, at least not right now. It's pretty dull and ugly. And, if you run the ruby file at this point, you won't see any output.

There are 2 more write statements to add, and for convenience we're going to place them within a ruby method. The argument to this method is going to be the string we want to parse. Ragel expects to find a variable named "data" containing an array of ASCII codes, so we will need to convert our string to an array. This is done very easily in Ruby using the unpack method.

def run\_machine(data)  
 data = data.unpack("c\*") if data.is\_a?(String)  
 %% write init;  
 %% write exec;  
end

write init tells Ragel that we want to generate initialization code for the state machine. The code Ragel generates here is:

begin  
 p ||= 0  
 pe ||= data.length  
 cs = hello\_start  
end

The variable p keeps track of which character in the data string we are currently parsing, starting at 0. pe is an upper limit for p. cs stores the current state of the state machine, and here it is initialized to the starting state of the state machine. These variables are discussed in the User Guide.

write exec tells Ragel to write the meat of the parser (finally!). The code generated here will actually take an input (the data argument) and determine what the state of the system should be based on that input, executing any actions which might be triggered along the way. Let's add some puts statements so we can follow the code execution.

def run\_machine(data)  
 data = data.unpack("c\*") if data.is\_a?(String)  
 puts "Running the state machine with input #{data}..."  
  
 %% write init;  
 %% write exec;  
  
 puts "Finished. The state of the machine is: #{cs}"  
 puts "p: #{p} pe: #{pe}"  
end

Just add 2 more lines at the end to call run\_machine with various arguments and then we can actually compile and run our state machine.

run\_machine "h"  
run\_machine "x"

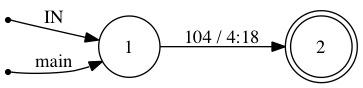
And here we go...

Running the state machine with input 104...  
hello world!  
Finished. The state of the machine is: 2  
p: 1 pe: 1  
Running the state machine with input 120...  
Finished. The state of the machine is: 0  
p: 0 pe: 1

It worked! Now, to help us interpret the values of p, pe and cs let's take a look at the state chart of this state machine. Ragel has built-in Graphviz support to create state charts. We need to use the -V flag instead of -R.

ragel -V simple\_state\_machine.rl > simple\_state\_machine.dot

If you render the resulting simple\_state\_machine.dot file in [Graphviz](http://www.graphviz.org/), you should get something like this:



We can see that the state machine has only one possible transition, from state 1 to state 2. When we passed "h" as the parameter to run\_machine we did indeed end up with the variable cs (current state) equal to 2 at the end of our run. When "x" was passed, we ended up with cs = 0. 0 is the error state, indicating that an error occurred in the state machine. (You can tell that 0 is the error state by reading some of the variable assignments generated by write data, the code I said was dull and ugly.)

In the label 104/4:18 over the arrow transitioning from state 1 to state 2, the 104 corresponds to the ASCII code for the letter "h". (Type "?h" in irb.) The / indicates that an action is being performed, and 4:18 tells us that the action starts at line 4, column 18 of the .rl file. Had we given our action a name, that would have appeared here instead of the file position.

By the way, here's the (textmate-specific) shell script I use to run all these steps quickly:

ragel -R simple\_state\_machine.rl  
ragel -V simple\_state\_machine.rl > simple\_state\_machine.dot  
dot -Tpng simple\_state\_machine.dot > simple\_state\_machine.png  
open simple\_state\_machine.png  
ruby simple\_state\_machine.rb  
mate simple\_state\_machine.out

Now, try running this code:

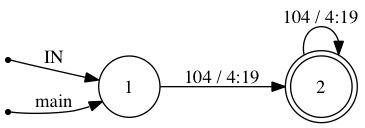
run\_machine "hh"

You should get:

Running the state machine with input 104104...  
hello world!  
Finished. The state of the machine is: 0  
p: 1 pe: 2

You don't get "hello world!" twice. Sorry. Our state machine is only looking at a the first character we pass. It knows we gave it two characters, the variable pe = 2, but after it evaluates the first character it's in a final state. There's no arrow coming out of the state 2 circle. So, passing additional input results in the system entering the error state. If we want the entire data string to be evaluated, we need to make a small change to our machine specification.

main := expr+ @ { puts "hello world!" } ;



(Try expr\* instead of expr+ and see how the state chart is different.)

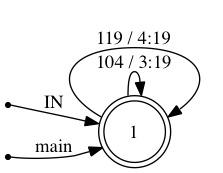
Now, try running this new state machine with inputs "hhh" and "hxh":

Running the state machine with input 104104104...  
hello world!  
hello world!  
hello world!  
Finished. The state of the machine is: 2  
p: 3 pe: 3  
Running the state machine with input 104120104...  
hello world!  
Finished. The state of the machine is: 0  
p: 1 pe: 3

When we pass "hhh", we get a "hello world!" for each "h". When we pass "hxh", we get the first "hello world!", but when we hit the "x" we enter the error state, so the last "h" doesn't get evaluated.

Here's one more [example](http://ananelson.com/blog/2008/01/multiple_state_machine.rl), this time without defining a run\_machine method:

%%{  
 machine hello\_and\_welcome;  
 main := ( 'h' @ { puts "hello world!" }  
 | 'w' @ { puts "welcome" }  
 )\*;  
 }%%  
 data = 'whwwwwhw'  
 %% write data;  
 %% write init;  
 %% write exec;



welcome  
hello world!  
welcome  
welcome  
welcome  
welcome  
hello world!  
welcome

So, there you go. Hours of entertainment await you. We've only scratched the surface of Ragel's features here, but you should now be able to navigate through the User Guide without too much trouble. If you need a better reason than "fun" to play with Ragel, then bear in mind that parsers are a great tool for constructing Domain Specific Languages (DSLs), and state machines are magic code shrinking machines for situations where you need to keep track of the, er, state of something and control the transitions between states (i.e. business logic). I would highly recommend everyone to read[this article about Ragel](http://zedshaw.com/essays/ragel_state_charts.html) which inspired me to check it out. If you're into Rails, then take a look at the acts\_as\_state\_machine plugin which might be more intuitive than Ragel at first. If the DSL angle is more your cup of tea then you might want to look at [ANTLR](http://www.antlr.org/) instead, which has a different focus and feature set than Ragel.

# ===== Ragel State Charts =====

<http://zedshaw.com/archive/ragel-state-charts/>

The [Ragel State Machine Compiler](http://www.cs.queensu.ca/~thurston/ragel/) is one kick-ass piece of software by [Adrian Thurston](http://www.cs.queensu.ca/~thurston/) that I’ve been using for about a year now. What Ragel does is use a mixture of C/C++/D/Java code and a state machine specification to produce a functioning state machine. Using Ragel you can process text for network protocols, program logic, programming languages, parsers, etc.

*NOTE: This essay is old and just here for historical purposes.*

# **Specifying Servers With Ragel**

What I’m going to cover in this essay is some of the wicked weird ways I’ve been using Ragel to create rock solid network servers. I’m going slightly outside the bounds of Adrian’s original intent, but he designed it to be flexible enough that Ragel just handles the new work easily. I’d say using Ragel has significantly cut down on the amount of work I do to design new protocols, and I’m dying to revisit old servers I’ve written.

The best thing you’ll get from this essay is how state machines can make your networking code very bullet proof. Specifying your operations in a state machine has always been a great way to validate correct operation, but in the past it’s been a major pain in the ass. Using Ragel and a few little tricks I’ve been able to easily specify some very stable software with minimal development effort.

## Why Ragel Is Cool

This isn’t anything new as there’s been many software packages that do this, but what makes Ragel different is you can:

1. inject actions at any point in the machine’s transitions
2. use code to alter the machine’s state on the fly
3. flexibly mix between regex specification and state chart
4. specifications
5. produce a variety of machine styles like table, flat, or goto
6. machines
7. get varying degrees of control over the state minimization
8. easily mesh it with whatever IO or character access code you have
9. and it’s **fast** as hell

All of this sounds super fancy, but it’s really based on solid research done on state machines over the years. Adrian simply took all the best research and built a sweet little domain language for state machines. My favorite feature is how he cleanly merged the two styles of specification: regex and state chart. Regex style machines are common in text parsing, but state charts are better for specifying program logic. Having the power to use both makes for some damn good results.

What many people don’t realize when they first look at Ragel is that its flexibility at specifying regular state machines means that it breaks a cardinal rule of lexical analysis: A regex cannot parse nested structures.

I won’t get into why, but try it yourself sometime on an XML document. Use some fancy regex to pull out certain tags in the document then mix up the structure. You’ll find yourself able to do it, but every little change requires a tweak. Eventually you get this monstrosity that can’t be maintained. It’s just better to realize a regex is bad at this and mix regex analysis with parsing.

Ragel however gives you an insane amount of flexibility at**combining** various complex machines so that you can get damn close to processing hierarchical structures. It’s not complete enough to fully replace a LALR grammar generator, but it’s advanced enough to handle almost all the nasty hand coded lexers out there. Ragel produces efficient machines from fairly complex specifications and outputs fast as hell code of varying styles.

## Ragel In Mongrel

Ragel is the software behind [Mongrel’s](http://mongrel.rubyforge.org/) speed and correct HTTP processing. With very little effort I was able to use Ragel to create a full HTTP processor that can process the protocol at insane speeds. From a developer point of view Ragel let me crank out a fully functioning web server in about a week. In fact, the number of changes to the parser in Mongrel only accounts for 4.2% of the total revisions.

Specifying the protocol with a clear state machine also made Mongrel more secure than other servers. Simply being more explicit about what is valid HTTP means that most of the security attacks that worked on Apache were rejected outright when tried on Mongrel. In fact, we’d put Mongrel behind Apache and watch Mongrel reject about 80% of the attacks while Apache let them right on through.

I thought, if using Ragel on the protocol parsing made it more secure by default, then what would happen if I used it on other parts of the server? What if I created a similar state machine for the socket connection state or the server’s internal logic? Would that make things more secure as well? How hard is it to specify things this way?

This document is my first cut at answering these questions so other people can try them out and see if it works for them.

# **Enter Utu**

Very quickly, [Utu](http://savingtheinternetwithhate.com/) is my latest freaky experiment in writing better client/server software. It’s the culmination of years of experience implementing various protocols and inventing my own. What I’m trying to do is simultaneously try out a weird ass idea and use that idea as a catalyst for doing an alternative protocol design and implementation.

You don’t need to know too much about Utu really, but you need to understand a small bit to see why using a state machine is a great way to approach the problem.

Utu is a messaging server that has built-in cryptography and sender pays enforcement. Imagine an SMTP server that was always encrypted, could identify all senders and receivers by public keys, and could force senders to “pay” to send based on the opinion of receivers. In Utu this is called “hate calculations”. However, since Utu is just a generic messaging server with a simple hub/spoke design, it can do SMTP, HTTP, or IM style communications.

The goal for Utu is to fight the griefers of the Internet with hate. As you communicate with people over Utu you can tell the Hub you hate them. The amount you hate them turns into a throttling hate calculation they have to do before they can continue talking to anyone.

Let’s take an IRC channel as the best example of where Utu will help. Typically IRC channels are either chaotic free-for-alls or Nazi controlled kingdoms. It’s also common that the operators of a heavily controlled IRC channel don’t follow their own rules.

In Utu, a channel wouldn’t have operators. Instead, when a griefer showed up to cause mayhem, all the participants would be able to “pay the hate tax”. Once members pay to hate the griefer the Hub then starts throttling that person at the average paid level. The more people hate them the more they cook their CPU.

A fictional Utu conversation in this case might look like this:

1:00 < griefer > Ruby on Rails sucks!  
1:01 \* joe452 hates griefer at level 24  
1:01 \* frankyboy hates griefer at level 26  
1:01 \* argc12 hates griefer at level 32  
1:10 \* griefer's hate is now 27.3  
1:11 < joe452 > that should shut him up  
1:20 < griefer > Damn guys, it takes me 10 minutes to send a message. You suck!  
1:21 \* griefer leaves #rubyonrails

The idea is that *joe452*, *frankyboy*, and *argc12* all payed to hate*griefer* by performing the same calculation he’d be penalized with. This payment is a one time thing, and after they do that the Hub turns around and throttles griefer permanently at 28 (rounded up) whenever he talks to that room.

## State Machines In Utu

Utu already uses Ragel to parse its simple wire protocol (another document on that coming soon). The problem was if the hate calculations are to work they have to be maintained across socket connections. If *griefer* above is able to say something, disconnect, and then reconnect to avoid doing the hate calculation then it won’t work.

What’s needed is to maintain the state of *griefer’s* connection between socket reconnects. If he was forced to pay some hate, and he disconnects, then when he comes back his socket is immediately put into the “Hated” state.

I originally coded this by hand and it was nasty. On a hunch I redid the same logic using a Ragel state chart and simplified it tremendously. The new state machine is just starting out but already I’ve got the following advantages:

\* I can design and test the logic of a Hub Connection without actually having any network running.

\* The new machine is explicitly specified so people trying to get around it can be booted.

\* It’s a hell of a lot easier to maintain and understand.

\* I can actually use this as the specification for an Utu server. A single state machine diagram and a Ragel specification works much better than pages and pages of RFC documentation.

The rest of this document describes how I did it so that you can try it in your own servers.

# **State Machines**

When I show people this stuff they look at me like I’m crazy. Here’s a very recent chat with a bunch of high speed Ruby coders to demonstrate:

21:14 < zedas> http://pastie.caboo.se/29404 check this out  
 21:14 < zedas> that's the state machine for controlling an utu connection, with the hate   
 calculating process included  
 21:14 < zedas> then there's a unit test that basically just feeds various combos of events   
 (UEv\_\*) to test that the logic all works  
 21:15 < zedas> and then i can validate it with the graph ragel produces at the top  
 21:15 < zedas> and what's really cool is this syntax is very close to CSP syntax  
 21:16 < technoweenie> zooom  
 21:16 < technoweenie> right over my head :)

Dammit this stuff is not hard, it’s just that nobody learns about state machines unless they take an obscure compiler design class. Very few universities teach state machines as a method of specifying the logic for a program. Even fewer cover Hierachical State Machines or State Charts.

To rectify this I’m gonna give you a crash course in state machines that will hopefully help you understand enough to get what’s going on. **Skip this if you know about FSM already.**

## State Machines Are Classes (kinda)

I think the best way to understand state machines is to see how they might map to an Object Oriented language like Ruby. Fundamentally all state machines deal with State, Events, Transitions, and Actions. These can be mapped to OO code like this:

* State – instance variables or data.
* Events – public methods or messages other people can send.
* Actions – private/protected methods or internal logic when something
* happens.
* Transitions – changes to instance variables to move to a new state.

Take a look at this simple Ruby class below as an example:

001 class HelloMachine  
002 def initialize  
003 @message = "Hello World"  
004 end  
005   
006 def say\_hello  
007 if @message then write\_message else write\_error end  
008 end  
009   
010 def change\_message(msg)  
011 @message = msg  
012 if !@message then write\_error end  
013 end  
014   
015 private  
016 def write\_message; puts @message; end  
017   
018 def write\_error; puts "No hello available."; end  
019 end

Each part of this class loosely matches what is in a state machine:

1. message is the class’s current state. It can be something to print or nil.
2. Changes to message amounts to a transition, that’s what change message does.
3. say\_hello and change\_message are both events that cause something to happen.
4. change\_message changes the internal state (message) while say\_hello changes the *screen* state by printing something.
5. The if-statement in say\_hello handles the two states of nil or not-nil.
6. The private methods write\_message and write\_error are actions for the machine. They’re called on “transitions” or to perform activity when an event happens.

This is how most code these days is written in the OO world, but even this little class has a problem: it’s not *explicit* what message can be. It's just assumed that people won't be stupid and set it to anything other than nil or a String. Ruby's puts is pretty powerful, but if you want to really make sure that it's used properly you'd have to add extra if-statements to change\\_message so thatmessage can’t be in an invalid state.

# **State Machines Are Explicit**

What a state machine formalizes is mathematically how these four pieces (State, Events, Actions, and Transitions) fit together and then defines explicitly what is allowed. Rather than just letting `message be anything, a state machine defines a set of allowed values. That’s it. Rather than allowing any event a state machine defines a limited set of events. Transitions are also restricted to what’s defined in the state machine so that it’s clear how a machine moves from one state to another and what causes that move. Actions are quite as well formalized since they’re more of a practical addition to make state machines useful for processing.

What annoys most programmers about state machines (apart from having to learn tons of math) is that they require this explicit thinking and covering all the possibilities. When you code up some OO you just code. As you build it up you start adding more restrictions, cover the code in unit tests, and then assume it works. With a state machine you sit down and define all the possibilities, and then the state machine compiler you use makes sure it is always in this domain.

As a quick taste, I can take the above HelloMachine and rewrite it like this in Ragel:

001 Hello = (  
002 start: (   
003 say\_hello @write\_message -> start |  
004 set\_message\_string @change\_message -> start |  
005 set\_message\_nil @write\_error -> no\_message   
006 ),  
007   
008 no\_message: (   
009 set\_message\_string @change\_message -> start |  
010 set\_message\_nil -> no\_message |  
011 say\_hello @write\_error -> no\_message  
012 )  
013 ) >initialize;

In this example we setup a new kind of event set\_message\_string and set\_message\_nil to indicate that you’re only allowed to set it in two ways. There’s also two basic states *start* and no\_message. The > symbols mean transition to that state, and `write\_error means run the *write\_error* action before you transition.

Now, obviously this isn’t much smaller than your original code, and you’ve gotta understand the new syntax to get it, but take a look at the following diagram generated by Ragel from this machine (click for larger image):

HelloMachine

That’s a fairly nice concise diagram for what is going on. The terminology might be a little foreign, but right away you can see when each action is called, what causes the transitions to the different states (states are in the circles), and how loops are handled. Trying to get a similar description from a larger set of source code would be impossible without some serious tools and it wouldn’t even be close to as accurate.

# **Using State Machines In Protocols**

When you design a server there’s several places where you have to control determinism to make sure things stay sane:

1. lexical elements (lexemes).
2. syntactic structure (grammar)
3. semantic meaning and analysis (logic)

Typically you see these terms used in the design of compilers but not really in designing servers. The thing is clients and servers do much of the same work that compilers do, it just has to do it dynamically and with malicious user inputs from untrusted sources. Why not leverage the decades of theory and practice in compiler design to make servers more robust?

With this in mind we can see using a typical lexer for lexical elements and a parser generator for the grammar should be easy. If you design your protocol right you can get away with just using Ragel for both of those as well, but some protocols will need a real parser generator. I recommend “Lemon”:<http://www.hwaci.com/sw/lemon/> if you need one that’s good with memory and error handling.

For semantic meaning we’re going to use the state chart style of machine specification for Ragel. This way of specifying a state machine uses labels and transitions inside nested state machines to make processing happen. You’ve already seen this style with the Ragel version of the HelloMachine we had above.

# **Utu Connections and Hubs**

Our server will use two state machines to manage its operations. One will be for the Hub and will be very simple for now. Its job is simply dealing with new network connections, handling the start-up process, and stopping connections that are dead.

The Hub manages a set of Connection machines that manage the state of each connected socket. The Connection receives events from the socket and other input points and, depending on what needs to be done, makes transitions around the machine. The Connection does the majority of the processing and is only alive while the socket is connected.

# **Utu Hub State**

When the Utu server starts up it needs to create a Hub that’s listening on a socket so it can service connections. Part of this process involves loading a key either from a file or dynamically generating it. Once the cryptography is ready to go and the Hub is listening on the socket the Hub just has to keep processing clients as they open and close.

First important thing I do in “the hub.rl”:hub.rl file (after some C boilerplate and action definitions) is to list the events we’re interested in:

001 listen='L';  
002 gen\_key='K';  
003 gen\_key\_done='k';  
004 key\_file\_load='F';  
005 conn\_open='O';  
006 conn\_close='C';  
007 done='D';

Normally Ragel operates on characters, but I can just as easily use these as events to the machine by defining similar *enum*definition in a C .h file. For now we’ve just got a bunch of characters that stand for each event that’s allowed.

After we say what events are possible, I specify the state machine, transitions, and actions to call:

001 Hub = (  
002 start: (  
003 gen\_key -> GenKeys  
004 ),  
005  
006 GenKeys: (  
007 gen\_key -> GenKeys |  
008 gen\_key\_done -> CryptoReady |  
009 key\_file\_load -> CryptoReady  
010 ),  
011  
012 CryptoReady: (  
013 listen -> Listening  
014 ),  
015  
016 Listening: (  
017 conn\_open `open\> Listening |  
018 conn\*close `close -> Listening |  
019 done -> final  
020 )  
021  
022 ) >begin %finish `!error;  
023  
024 main := ( Hub %{ printf ("\\n"); } )\\*;

This is not really enough logic to warrant a full state machine approach, but the Hub will start to pick up more complex decisions and logic as Utu progresses. Remember I already tried to do just the above in hand written form and it was disgusting. Hopefully this will be much better.

Let’s walk through this line by line so that you can start to understand how a Ragel state chart is specified and what each bit of syntax means. The Hub’s machine is simpler than the Connection’s so we’ll be able to cover the various features and understand it. First up, here’s the diagram from the above machine for you to take a quick look at (click for a larger version):

image

* 001: This starts off the Hub machine. The closing parenthesis is on 020.
* 002: The *start:* is a label. We’re basically telling Ragel where the Hub machine should start its life.
* 003: This is the first event > state transition we’re making. We’re saying that when the *gen*key\_ event happens, transition to the *GenKeys* state. In this case *GenKeys:* is on line 006.
* 004: A **very** important point is that each of these labeled states must have a ‘,’ character separating them. Ragel won’t report an error but the machine won’t work right without it.
* 006: Here’s our state for handling the key generation process. There’s three events that this state can accept.
* 007: The first event we handle is another *gen*key\_ so that we can report status during the lengthy key generation process. That’s why this transitions> *GenKeys* again.
* 008: At the end of 007 there was a | (pipe, OR symbol) that separated *gen*key\_ event transitions from this one,*gen*key*done*.
* 009: Just like *gen*key*done* event the *key*file*loaded event causes the machine to transition to*CryptoReady\_ indicating it’s done.
* 012014: These lines are just like *start:* except that we’re waiting for the server’s socket to be ready for listening. Once we get the *listen* event we transition> *Listening*.
* 016: Our Listening state that we’ll now loop continuously inside until we get a *done* event.
* 017-019: Here we process the Connection open/close events, and the done event for when the hub shuts down. The new thing to notice here are the open\_ and \_*close*statements. These say to run the actions named “open” and “close” and are how you make the machine do things for the program.
* 022: This line ends the *Hub* machine and then appends a few new actions. Ragel uses a funky but very cool syntax for actions that we’ll cover in a second.
* 024: The last line simply tells Ragel that the main machine to run is Hub and a small action with the %{ .. } syntax prints out a newline when the Hub exits.

Take a look at the machine, [the hub.rl file](http://zedshaw.com/archive/ragel-state-charts/hub.rl) and the [hub state diagram](http://zedshaw.com/archive/ragel-state-charts/HubState.png) and bounce between them trying to trace how the actions move when different character events are handed to the machine. Using the above description of the syntax you should be able to use the diagram to sort out how the Hub works. If not, try picking an event and then matching it to lines in the diagram.

### Actions In Ragel

Ragel’s unique feature is that you can embed an “action” anywhere in the state machine, and at the same time control what stage of the machine should run the action. Ragel uses a wide array of symbols prefixed to the action name to indicate when it should be run. Let’s take our machine above and some of the actions:

* >begin on line 022 means run the *begin* action on entry to the Hub machine.
* %finish on line 022 means run the *finish* action when the Hub is finally exited.
* \*`!error means run the *error* action at any point where there’s an error.
* The ‘' symbol also means on each transition that's not an entry or exit. So on 017 and 018 where we doopen and `close we’re basically saying to run those for that event and then transition.

The Ragel manual has a ton more of these kinds of specifiers and the syntax can get very advanced, especially once you start adding error terms and such. For now just know in a state chart machine when you want to run an action put it after the event and prefixed with an @ just like on lines 017 and 018 above.

## Generating The Code

If you take a look [at the hub.rl file](http://zedshaw.com/archive/ragel-state-charts/hub.rl) you’ll see how this is all worked into the middle of some C code. When we want to generate this file into C code we run this command:

$ ragel hub.rl | rlcodegen -o hub.c

The resulting hub.c file isn’t much to look at but it compiles and then you can start running it. We’ll show how it’s run and unit tested in the next section.

When you want to get a diagram like the ones I’m showing you run this command:

$ ragel hub.rl | rlcodegen -V -p -o hub.dot  
 $ dot -Tpng hub.dot -o hub.png

To make the above work you have to install the [Graphviz](http://www.graphviz.org/)diagram system and make sure *dot* is in your path.

How the *ragel* and *rlcodegen* tools work is that *ragel* reads your state machine specification and then outputs XML on stdout (you can save it to a file). This gets passed to *rlcodegen* which reads it and produces code of different styles and for different languages. For example I can have *rlcodegen* produce a goto style machine (they’re fast) for Java with this:

$ ragel -J ajavathing.rl | rlcodegen -o -G2 ajavathing.java

You can also control other options so read the [Ragel PDF](http://www.cs.queensu.ca/~thurston/ragel/) for your version (down about mid page).

## Unit Testing The Hub

A state machine like this has some interesting testing properties. First we can actually unit test the logic for the hub without connecting it to any other part of the system until we’re ready. Second we can actually [fuzz](http://en.wikipedia.org/wiki/Fuzz_testing) the machine to see how it reacts to invalid inputs. Finally, testing the Hub involves feeding it varying streams of events, so we can try out different usage scenarios really easily.

Here’s a sample using the [CUT](http://www.falvotech.com/content/cut/) test suite where we just test a simple run:

001 void \_\_CUT\_\_UtuHub\_Hub\_state()  
002 {  
003 HubState fsm;  
004   
005 const char simple\_events[] = {UEv\_GEN\_KEY, UEv\_GEN\_KEY\_DONE,  
006 UEv\_KEY\_FILE\_LOAD, UEv\_LISTEN, UEv\_DONE, 0};  
007   
008 hub\_exec(&fsm, simple\_events, 1, 1);  
009 }

Notice on line 005 we setup an array of a bunch of C constants. Each of those is set to the characters in our events section above, so we’re basically feeding characters to the *HubState* fsm. The *hub*exec\_ function used on 008 above looks like this:

001 int hub\_exec(HubState \*fsm, const char \*events, int initialize, int finalize)  
002 {  
003 if(initialize) ASSERT(HubState\_init(fsm), "failed to init HubState");  
004   
005 while(\*events) {  
006 if(HubState\_exec(fsm, \*events) == -1) return 0;  
007 events++;  
008 }  
009   
010 if(finalize) ASSERT\_EQUALS(HubState\_finish(fsm), 1, "failed to finish machine");  
011   
012 return 1;  
013 }

The *hub*exec\_ method cuts down on repetition in the test cases (which we’ll demonstrate more with the Connection machine later). What it does is:

* 003: Initialize the Hub with HubState*init but only if asked.*
* *\* 005-006: Process each event until it hits a 0 indicating the last*
* *event. It uses HubState*exec to do this.
* 010: Finishes the HubState fsm if asked to using HubState\_finish.

Using this setup we can test out fairly complex usage scenarios with just arrays of events and some function calls to *hub*exec\_ to make them happen.

## Utu Connection State

The Connection machine (in the [connection.rl](http://zedshaw.com/archive/ragel-state-charts/connection.rl) file) is setup almost identically but is more extensive. There’s no new syntax in the file, so we’ll just show the events it handles and the machine and cover the new stuff.

001 open='O';  
002 key\_present='P';  
003 key\_reject='p';  
004 key\_accept='a';  
005 peer\_fail='T';  
006 member\_fail='M';  
007 member\_register='m';  
008 close='C';  
009 svc\_recv='s';  
010 msg\_sent='e';  
011 msg\_queued='Q';  
012 msg\_recv='R';  
013 hate\_apply='H';  
014 hate\_challenge='h';  
015 hate\_paid='b';  
016 hate\_valid='V';  
017 hate\_invalid='v';

Nothing new here other than a larger number of events to deal with. Notice that I did reuse some of the events from the [Hub machine](http://zedshaw.com/archive/ragel-state-charts/hub.rl) so that the C code is easier to maintain. In an ideal situation the Ragel code wouldn’t even have these specified and instead they’d just come from C.

Here’s the Connection’s state machine specification, which is twice the size of the Hub’s specification, but doesn’t have anything new except for a single nested machine used to handle hatred:

001 Connection = (  
002 start: ( open @open -> Accepting ),  
003   
004 Accepting: ( key\_present @key\_check -> KeyCheck ),  
005   
006 KeyCheck: (  
007 key\_reject @establish\_failed -> Aborting |  
008 key\_accept @tune -> Tuning  
009 ),  
010   
011 Tuning: (  
012 peer\_fail @establish\_failed -> Aborting |  
013 member\_fail @establish\_failed -> Aborting |  
014 member\_register @established -> Idle  
015 ),  
016   
017 Idle: (  
018 msg\_recv @recv -> Delivering |  
019 msg\_queued @queued -> Sending |  
020 svc\_recv @service -> Servicing |  
021 hate\_apply @hate\_apply -> Hated |  
022 close @close -> final  
023 ),  
024   
025 Hated: (  
026 start: ( hate\_challenge @hate\_challenge -> Awaiting),  
027 Awaiting: ( hate\_paid @hate\_paid -> Validating ),  
028 Validating: (   
029 hate\_valid @hate\_valid | hate\_invalid @hate\_invalid   
030 ) -> final  
031 ) -> Idle,  
032   
033 Aborting: ( close @aborted -> final ),  
034   
035 Delivering: ( msg\_queued @delivered -> Idle ),  
036   
037 Sending: ( msg\_sent @sent -> Idle ),  
038   
039 Servicing: ( msg\_queued @delivered -> Idle )  
040   
041 ) >begin %finish;  
042   
043 main := ( Connection %{ printf("\n"); } )\*;

Which is diagrammed by Ragel as (click to enlarge):

image:ConnectionState.png

There’s nothing new in the Connection state, it’s just more complex. Each event triggers an action and then transitions to a new state in the *Connection* machine.

The main difference is we use a nested machine for the *Hated*label on line 025. Just like Connection is a machine with a bunch of labels in it, each label can also point at a machine with labels in it. *Hated* is such a nested machine and it handles the process of starting a hate calculation (012), paying it (027), validating the hate (029), and then going back to normal processing (031).

The big thing with nested machines is that you can only transition to labels in that machine or to final. This is why the whole *Hated* state machine has a transition to -> *Idle* on line 031.

Another thing you should see is I didn’t include the !error on the Connection machine like in the Hub machine previously. The reason is this makes the diagram more difficult to understand since it produces those "DEF/error" lines in "the Hub Diagram":HubState.png which are really annoying initially. What I do is add the!error action after I’ve worked out the logic more completely.

# **Unit Testing Connection State**

Testing the Connection machine is just like the testing we did with the Hub machine:

001 void \_\_CUT\_\_UtuHub\_Connection\_state()  
002 {  
003 ConnectionState fsm;  
004   
005 const char simple\_events[] = {UEv\_OPEN,   
006 UEv\_KEY\_PRESENT, UEv\_KEY\_ACCEPT,  
007 UEv\_MEMBER\_REGISTER, UEv\_CLOSE, 0};  
008   
009 ASSERT(conn\_exec(&fsm, simple\_events, 1, 1), "simple events failed");  
010   
011 const char default\_setup[] = {UEv\_OPEN,   
012 UEv\_KEY\_PRESENT, UEv\_KEY\_ACCEPT,  
013 UEv\_MEMBER\_REGISTER, 0};  
014   
015 const char hate\_paid\_valid[] = {UEv\_HATE\_APPLY, UEv\_HATE\_CHALLENGE,   
016 UEv\_HATE\_PAID, UEv\_HATE\_VALID,   
017 UEv\_CLOSE, 0};  
018   
019 const char hate\_paid\_invalid[] = {UEv\_HATE\_APPLY, UEv\_HATE\_CHALLENGE,   
020 UEv\_HATE\_PAID, UEv\_HATE\_VALID,   
021 UEv\_CLOSE, 0};  
022   
023 ASSERT(conn\_exec(&fsm, default\_setup, 1, 0), "default setup failed");  
024 ASSERT(conn\_exec(&fsm, hate\_paid\_valid, 0, 1), "hate paid valid failed");  
025 ASSERT(conn\_exec(&fsm, default\_setup, 1, 0), "default setup failed");  
026 ASSERT(conn\_exec(&fsm, hate\_paid\_invalid, 0, 1), "hate paid invalid failed");  
027 }

The *conn*exec\_ function is almost identical to the *hub*exec\_ function and it’s just used to reduce repetition in the test code. In this test we’re trying out different interactions like a simple test then a few tests based on hate.

Otherwise there’s nothing more to the whole thing than just running different lists of events through the machines and making sure they are still operating.

When you start to compare the specification with [the diagram](http://zedshaw.com/archive/ragel-state-charts/ConnectionState.png)you should be able to tell where the *Hated* machine starts working, how it’s transitioned into, and follow the path of other events. Ragel uses numbers for the states, but with a machine this small it’s not hard to figure out what’s what.

# **Next Steps**

With my new found knowledge of Ragel state charts I’m starting to work out the Utu machines and testing their logic more extensively. The next step is to obviously make them do something useful. What will happen is based on this diagram you saw previously:

You can see the Connection gets events from the Hub and the sockets it’s managing. The events from the Hub are combinations of *hate*apply\_ for transferring hate between people over the connections and open/close events for when the Hub needs to stop a Connection.

Events from the socket will come from two little threads that parse data off the wire into messages for the machine, or send out messages that are generated from the machine.

The follow-up to this article explain how these state machines are actually used in practice to make the server operate, and how well they do when compared with hand coding. I’ll explore things like whether it reduced the code compared to the previous version, how changes were handled, how robust the new version is, and probably see what fuzzing the machine does to it.

# ===== Building tokenizers with Ragel =====

<https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel>

April 10, 2013,

By James Dabbs



One of the problems we've been wrestling with lately here at Emcien is finding patterns in network traffic (with an eye towards detecting intrusion attempts). We've exerted considerable effort to make our pattern detection engine run as fast as possible, but recently did some profiling and found that significantly more time was being spent tokenizing the input before handing it off for analysis. I've been building a new tokenization tool chain using Ragel and wanted to share a little of my experience by walking through the design of a simple Ragel parser for converting Apache log lines to JSON.

This article assumes a passing familiarity with [finite state machines (FSMs)](http://blog.markshead.com/869/state-machines-computer-science/) and regular expressions. Those of you who prefer code to text can check out the finished product [on Github](https://github.com/emcien/ragel-apache-logs).

# Why Ragel?

I had two main design constraints for this tokenizer project:

* Make it fast
* Make it work consistently in the engine (C), app (Rails) and on the client-side (Backbone)

Ragel works by defining a state machine directly and then using that definition to generate code in any of several host languages - C, D, Go and Java are supported by default, but there are projects to allow generation in other languages, including [javascript](https://github.com/dominicmarks/ragel-js). That nicely covers point 2, with the incidental benefit that each tokenizer is essentially a [declarative](http://latentflip.com/imperative-vs-declarative/) specification of the input it expects to see.

As for the first point - speed - Ragel exposes code generation strategies that can produce some *really* fast C code, as we'll see shortly.

# How it works

Broadly, using Ragel to write a program consists of an abstract machine specification and code to integrate that machine into a host language:

## The input specification

We'll want to generate a machine that can read in lines like this (clearly old) log entry

**71.109.86.22** **-** **-** [19/Apr/2006:02:27:44 -0700] "GET /bg/henna-take1.jpg HTTP/1.1" **200** **133028** "http://profile.myspace.com/index.cfm?fuseaction=user.viewprofile&friendid=6980420" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"

Here's the specification I ended up writing:

machine clf;  
 action mark { mark(); }  
  
 host = [0-9\.]+ >mark *%{ emit("host"); };*  
 user = alpha+ >mark *%{ emit("user"); };*  
 date = [^\]]+ >mark *%{ emit("date"); };*  
 request = [^"]+ >mark *%{ emit("request"); };*  
 status = digit+ >mark *%{ emit("status"); };*  
 size = (digit+ | '-') >mark *%{ emit("size"); };*  
 url = [^"]\* >mark *%{ emit("url"); };*  
 agent = [^"]\* >mark *%{ emit("agent"); };*  
  
 line = (  
 host space  
 '-' space  
 '-' space  
 '[' date ']' space  
 '"' request '"' space  
 status space  
 size space  
 '"' url '"' space  
 '"' agent '"' '\n'  
 );

Let's step through that -

machine clf declares the clf (common log format) machine and allows us to include it in other machines later.

Code in a {} block is embedded code in the host language, so action mark { mark(); } declares the mark action, but delegates its actual implementation to whatever code we end up embedding this machine in.

Ragel features several simple built-in machines and knows how to generate a machine from a basic regex.host = [0-9\.]+ defines a machine named host that accepts a string of one or more digits or periods (you could, of course, specify this more carefully). The other named machines are similar (recall that e.g. [^"]matches anything except a ").

The interesting part of the simple machine definitions is the >mark %{ emit("type"); }. > and % define actions to take on certain types of state transitions - in this case, marking the parser's position when entering (>) one of these machines and emitting the parsed token upon leaving (%). Again, the { emit("type"); } block is just code in the host language; we'll have to define an emit function in every program we write using this machine - but fortunately, that syntax works for calling functions in C, Ruby and Javascript.

line is a compound state machine built up from the simpler ones we've just defined. The default operation is concatenation, so line just looks for these parts one after the other.

Fun aside: Ragel integrates nicely with Graphviz. [Here's](http://i.imgur.com/QYoH0gh.png) the machine we just built (as generated by ragel -Vp).

Now that we've declared a machine, we should write a program to take it for a spin:

## A Ruby debugger

In fairness, nailing down your input format can be a delicate and error-prone process. I found it helpful to write a Ruby tokenizer first so that I could stop and introspect the parser's state at the point of failure. Here's what that might look like:

*%%{*  
 machine debugger;  
 include clf "clf.rl";  
  
 variable data @data;  
 variable p @p;  
  
 main := line $!{ error };  
 }*%%*

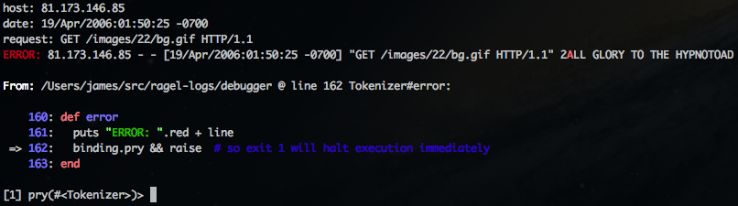
%% deliniates Ragel directives. Here we're defining a new main machine which includes our previously defined line and adds a call to the error function if the parser hits an error in any state ($!). Internally, Ragel uses the variables data for the text to parse and p for the symbol currently being read. We want these to be instance variables for a Tokenizer class, but fortunately Ragel allows us to specify what names those variables actually use. Here's the class definition:

**class** **Tokenizer**  
 **def** initialize path  
 @path = path  
 *# This writes the actual state transitions into the generated .rb file*  
 %% write data;  
 #% *# Fix syntax highlighting*  
 **end**  
  
 **def** run  
 File.foreach @path **do** |line|  
 *# Store the current line for later reference*  
 @line = line  
 *# Ragel expects data to be an array of integers and needs a little*   
 *# extra bookkeeping to know where to start and stop parsing*  
 @data = line.unpack 'c\*'  
 @p, pe = 0, line.length  
 eof = pe  
  
 *# This writes the actual machine instructions into the file - this*  
 *# code will consume input and and call `mark` and `emit` when applicable*  
 %% write init;  
 %% write exec;  
 **end**  
 **end**  
  
 **def** mark  
 *# Simply store the current position*  
 @ts = @p  
 **end**  
  
 **def** emit type  
 *# Print the type and text of the last read token*  
 puts "**#{**type**}**: **#{**@data[@ts...@p].pack('c\*')**}**"  
 **end**  
  
 **def** line  
 *# Print the current line, highlighting the current character*  
 pre, curr, post = @line[0 ... @p], @line[@p], @line[@p+1 .. -1]  
 highlight = curr =~ /\s/ ? curr.on\_light\_red : curr.light\_red  
 pre + highlight + post  
 **end**  
  
 **def** error  
 *# Print out the erroneous line, and halt so we can inspect the state of*  
 *# the machine*  
 puts "ERROR: ".red + line  
 binding.pry && **raise** *# so exit 1 will halt execution immediately*  
 **end**  
 **end**  
  
 Tokenizer.new(ARGV.first).run

Run

ragel -R **<filename>** -o debugger

to interpolate these directives into actual Ruby source code. You'll be doing this a lot, so I'd recommend making a Makefile for it - something like [this](https://github.com/emcien/ragel-apache-logs/blob/master/Makefile). Once the code is generated, you can run it with ruby debugger /path/to/infile. Now if we try to parse some unexpected input, we'll get something like:



and be dropped into a [pry](http://pryrepl.org/) terminal to poke around. Very convenient.

**How are we doing on speed?**

$ time ruby debugger log.100k > out  
  
 real 0m47.341s  
 user 0m47.098s  
 sys 0m0.201s

About 2100 lines/s. Tolerable for small files. Ragel does have the -T1 optimization option for Ruby code, but we can do better.

## A C binary

The C program is fairly similar to the Ruby one, just with more fiddly string details and less robust error handling. In this case, we'll be outputting the results as a JSON-formatted array of objects:

#define MAX\_LINE\_LENGTH 4096  
 char inbuffer[MAX\_LINE\_LENGTH], outbuffer[MAX\_LINE\_LENGTH];  
  
 %%{  
 machine parser;  
 include clf "clf.rl";  
 main := line;  
 write data;  
 }%%  
  
 *// Declare these in global scope so that the machine and our action*  
 *// definitions can access them.*  
 int cs;  
 char \*p, \*pe, \*ts;  
  
 *// Mark is similar, but C has no notion of instance variables*  
 *// To emit, we write this key:value pair to our output buffer*  
 **static** **inline** void mark() { ts = p; }  
 **static** **inline** void emit(char \*type) {  
 \*p = '\0';  
 char fmtbuffer[MAX\_LINE\_LENGTH];  
 sprintf(fmtbuffer, "**\"**%s**\"**:**\"**%s**\"**,", type, ts);  
 strcat(outbuffer, fmtbuffer);  
 }  
  
 int main() {  
 int lines = 0;  
  
 fputc('[', stdout); *// Start the list*  
 **while**(fgets(inbuffer, MAX\_LINE\_LENGTH, stdin) != NULL) {  
 *// Start writing the JSON object to the buffer*   
 *// Include a leading comma if we need to separate it from its predecessor*  
 **if** (lines) {   
 strcpy(outbuffer, ",**\n**{");  
 } **else** {  
 strcpy(outbuffer, "{");  
 }  
  
 *// Start the machine running on the input buffer*  
 p = inbuffer;  
 pe = inbuffer + strlen(inbuffer);  
 ts = p;  
 %% write init;   
 %% write exec;  
  
 *// Finalize the object, chomping off the last comma*  
 int len = strnlen(outbuffer, MAX\_LINE\_LENGTH);  
 outbuffer[len - 1] = '\0';  
 fprintf(stdout, "%s}", outbuffer);  
 lines ++;  
 }  
 fprintf(stdout, "]**\n**");  
 **return** 0;  
 }

Easy, modulo C's usual string-handling headaches. Compilation is similar to the Ruby case, only without the -R flag. There are several optimization options available - I'm using -G2 which generates a "really fast goto driven FSM".

**And just how fast is the finished product?** Compiled with -O3:

$ time ./parser < log.100k > out  
  
 real 0m0.418s  
 user 0m0.318s  
 sys 0m0.045s

~240k lines/s and two orders of magnitude faster than the Ruby version. Not bad[.](http://i2.kym-cdn.com/photos/images/original/000/138/246/tumblr_lltzgnHi5F1qzib3wo1_400.jpg)

## Parting thoughts

This machine barely scratches the surface of what Ragel can do - it also supports really fine-grained action specifications and tools for writing scanners, backtracking and controlling non-determinism. Full details are available in the [Ragel user guide](http://www.complang.org/ragel/ragel-guide-6.8.pdf).

My favorite thing about using Ragel is the "write once, run anywhere" workflow. With this machinery in place, tokenizing a new input format is just a matter of declaring a new grammar and running any of your tokenizers in a new language is just a matter of writing a new adapter. I'll leave writing a Javascript version of this machine as an exercise for the reader.

Happy parsing!