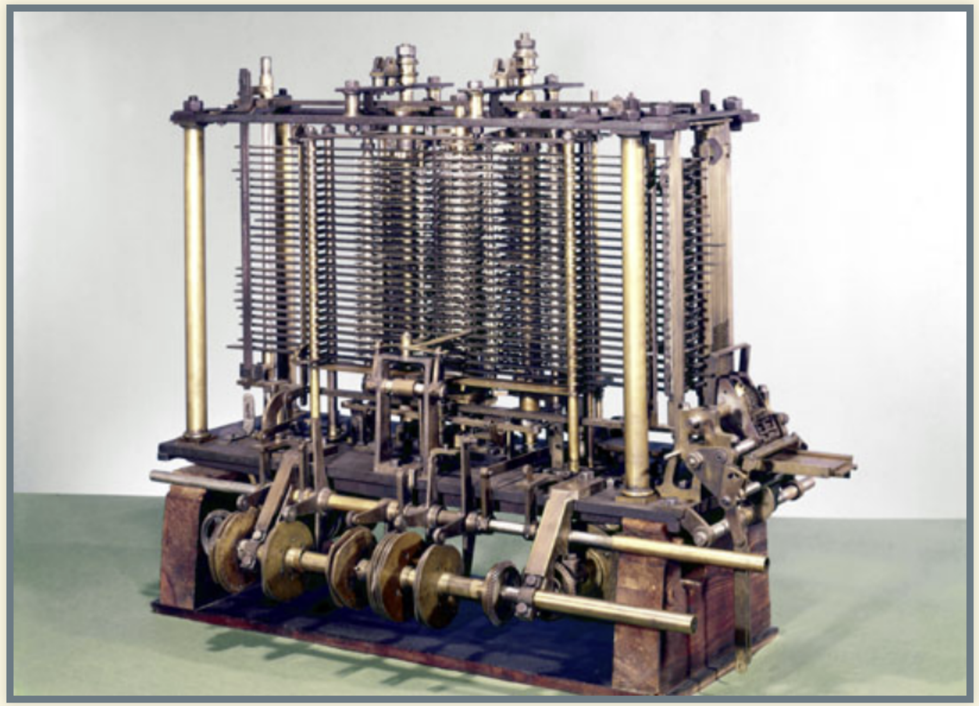
STATELY STATE MACHINES WITH RAGEL



RubyConf 2015

Ian Duggan

GOALS FOR THIS TALK

1. Convince you that Ragel is worth trying.
2. Give you some intuition about how it works.
3. Show you how to setup a basic parser.

HELLO

My name is Ian Duggan

I PLAY HOCKEY

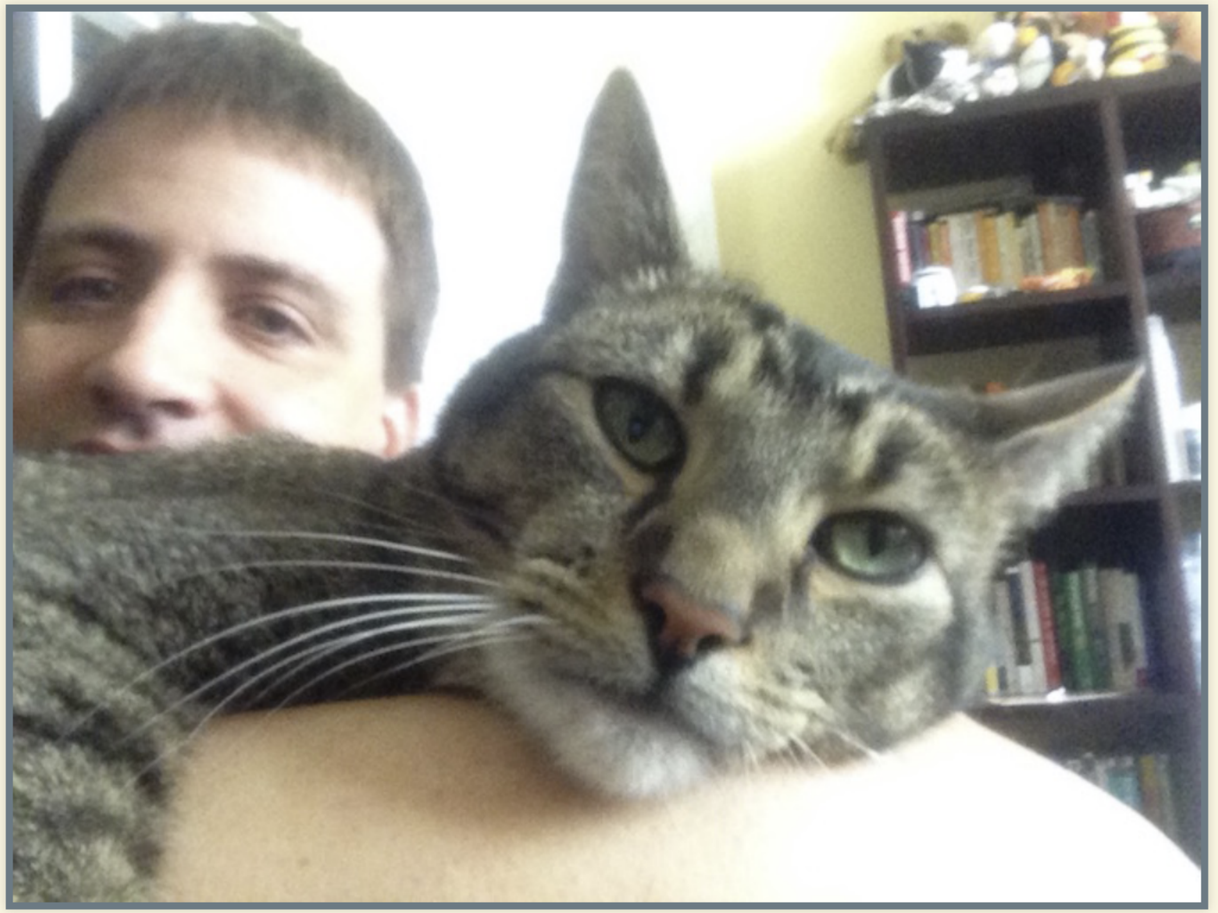
Several times a week.

I PLAY GUITAR

And banjo. And mandolin. And ukulele. Poorly. I have a fiddle that's gathering dust.

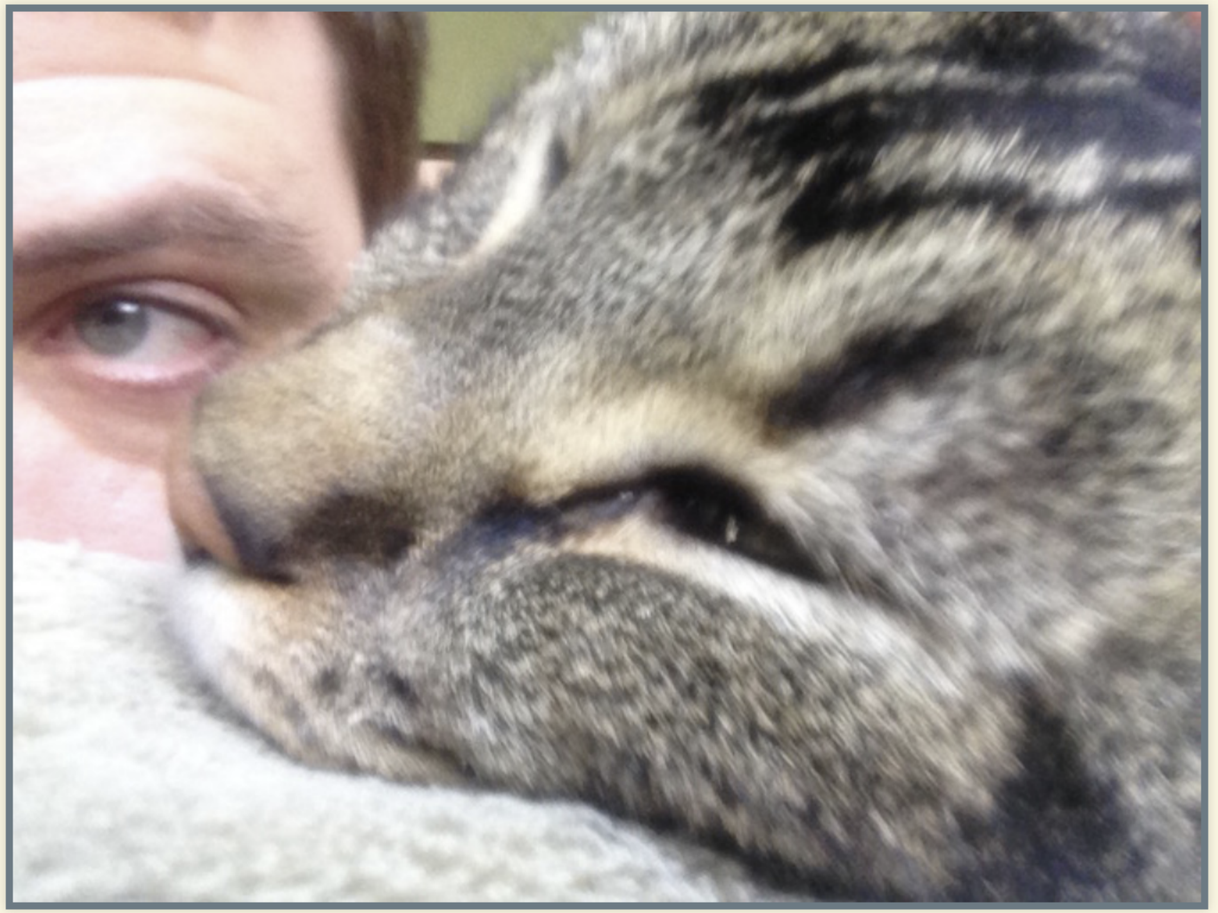
SOMETIMES I FLY

I LOVE MY CATS



THEY ARE GOOFBALLS

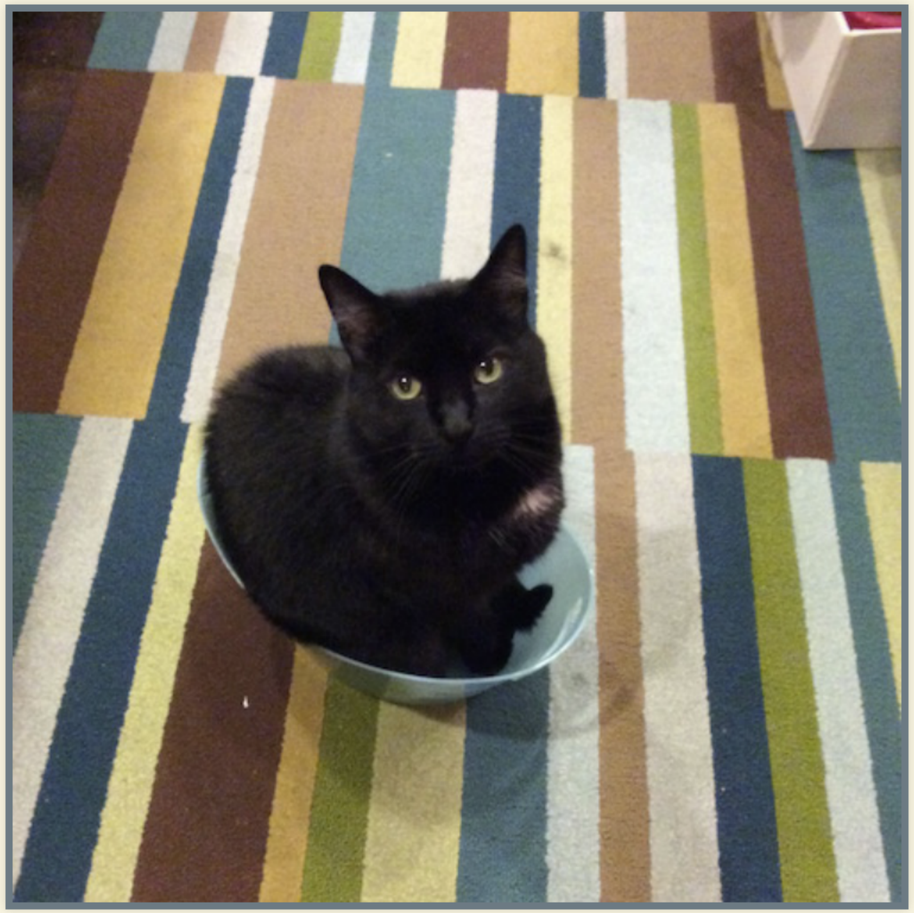
BUT VERY FURRY



AND DOPEY

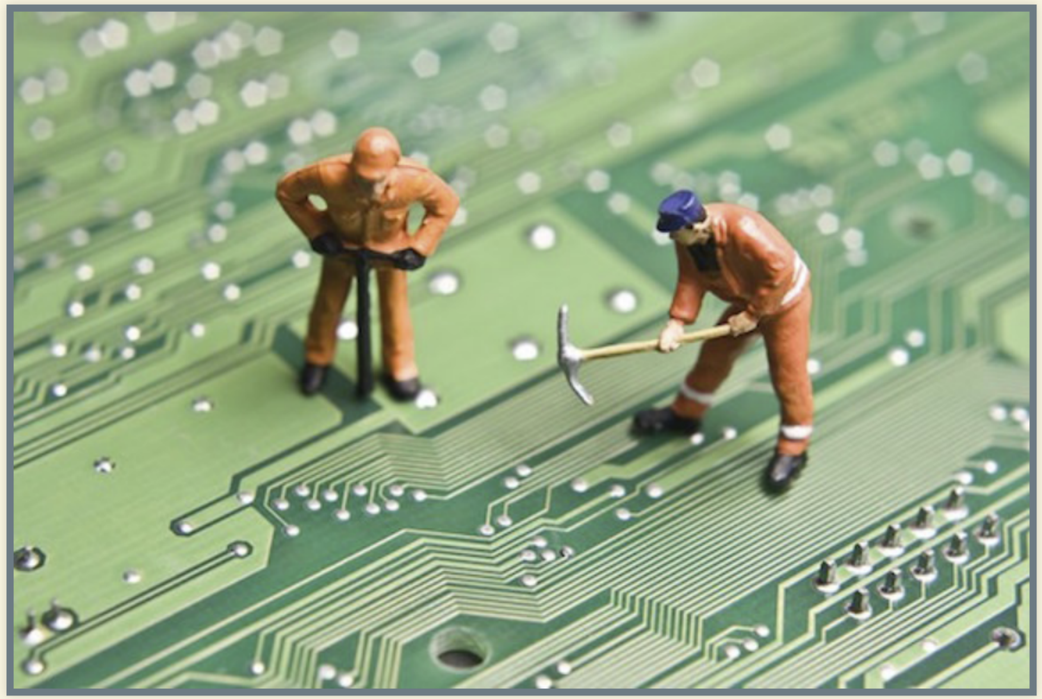
AND RELAXED

AND FUN



SOFTWARE! FTW!

I'm a software dude. I code things. I code the internets and the googles. I'm also a recovering technology entrepreneur. I've been in and out of startup institutions my entire life.



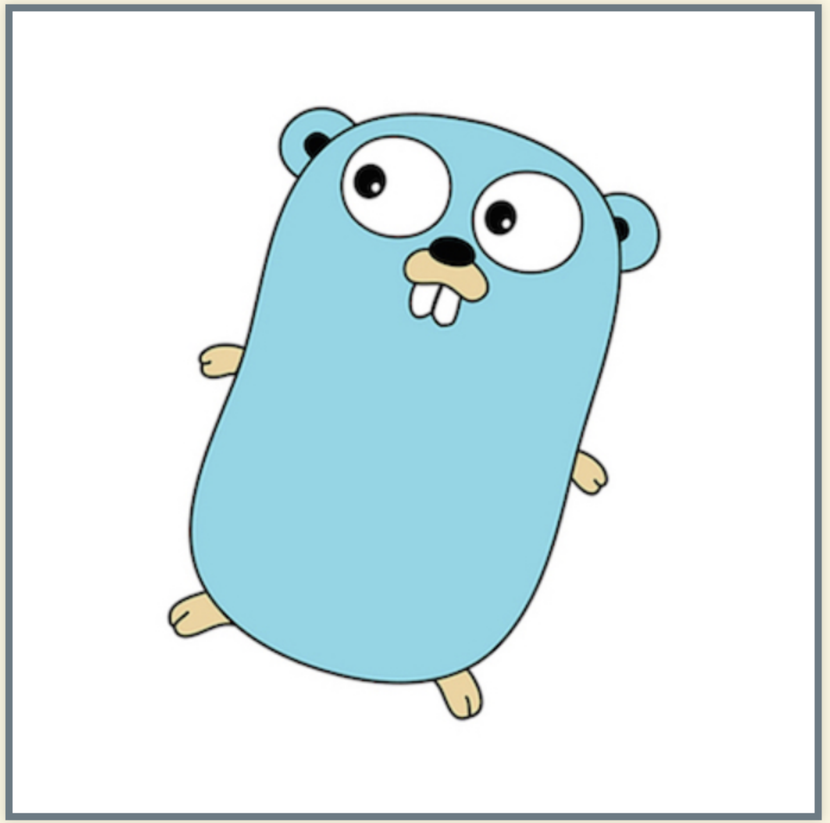
CURRENT STATUS

WE'RE HIRING (OF COURSE)

Lots of Ruby.

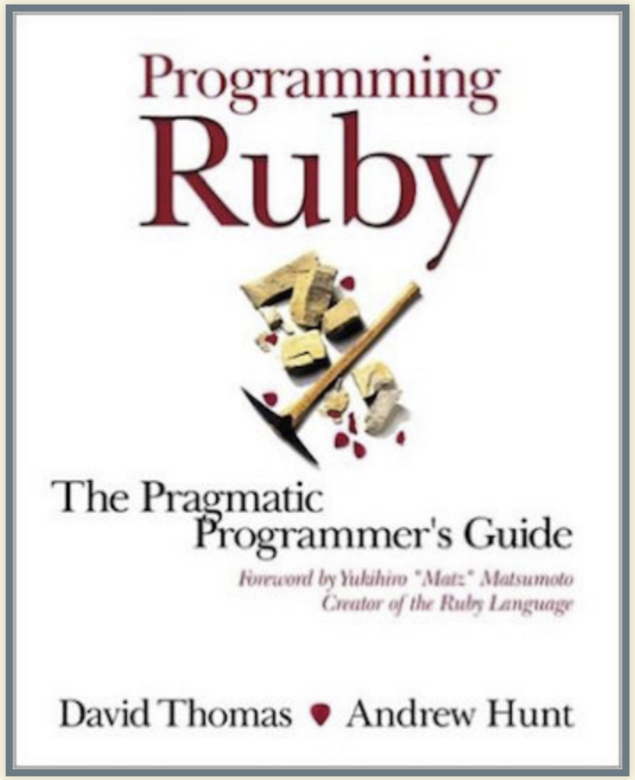
WE'RE HIRING (OF COURSE)

Lots of Go.

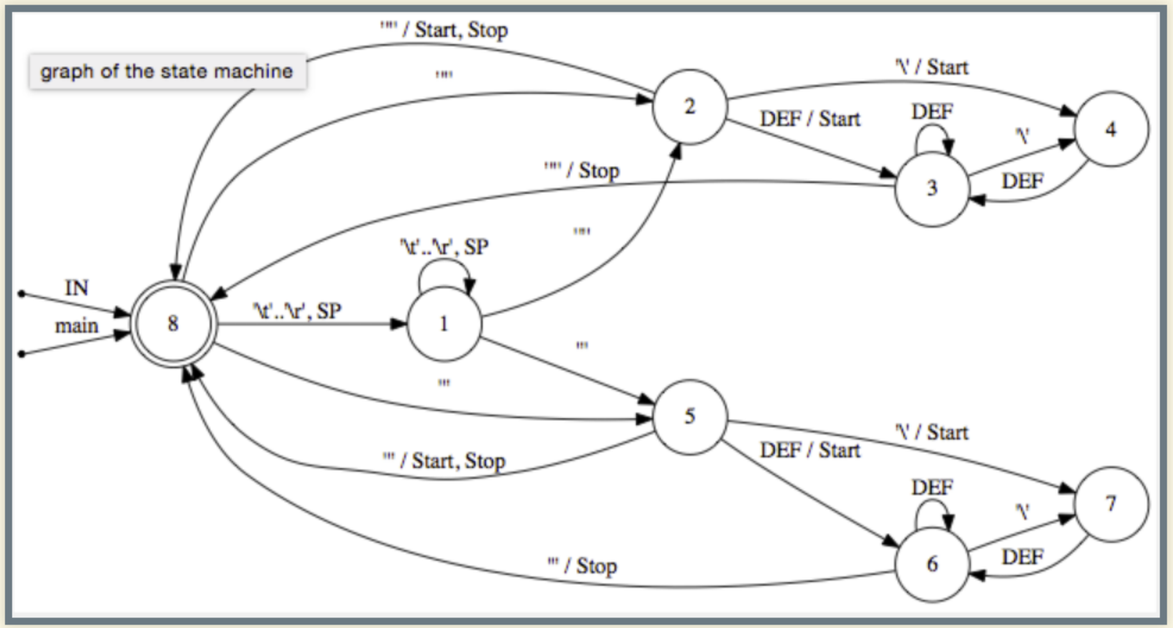


I'M A RUBYIST, SINCE 1.6

I've been using Ruby casually since the 1.6 days, and professionally for more than a decade.



TODAY IS ABOUT RAGEL



RAGEL IS REALLY COOL

If you don't have it in your bat-belt yet, you need to add it.
Today!

RUBY PROJECTS USING RAGEL

- Mongrel, Unicorn, Puma
- Whitequark
- Mail
- RedCloth
- Hpricot
- Gherkin

https://github.com/whitequark/parser/blob/master/lib/parser/lexer.rl

WHAT DOES RAGEL LOOK LIKE?

```
%%{
action dgt      { printf("DGT: %c\n", fc); }
action dec      { printf("DEC: .\n"); }
action exp      { printf("EXP: %c\n", fc); }
action exp_sign { printf("SGN: %c\n", fc); }
action number   { /*NUMBER*/ }

number = (
    [0-9]+ $dgt ( '.' @dec [0-9]+ $dgt )?
    ( [eE] ( [+\-] $exp_sign )? [0-9]+ $exp )?
) %number;

main := ( number '\n' )*;
}%%
```

BUT REGULAR EXPRESSIONS ARE EASY!

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. (from Wikipedia)

```
/a/          # match a
/abc/        # match "a", then "b", then "c" (concatenation)
/a|b/        # match "a" or "b" (alteration)
/gr(a|e)y/   # match "gr", then "a" or "e", then "y" (grouping)
/a?/         # match zero or one "a"
/a*/         # match zero or more "a"
/a+/         # match one or more "a"
/a{18}/      # match "a" 18 times
/a{2,}/      # match "a" 2 or more times
/a{2,10}/    # match "a" between 2 and 10 more times
```

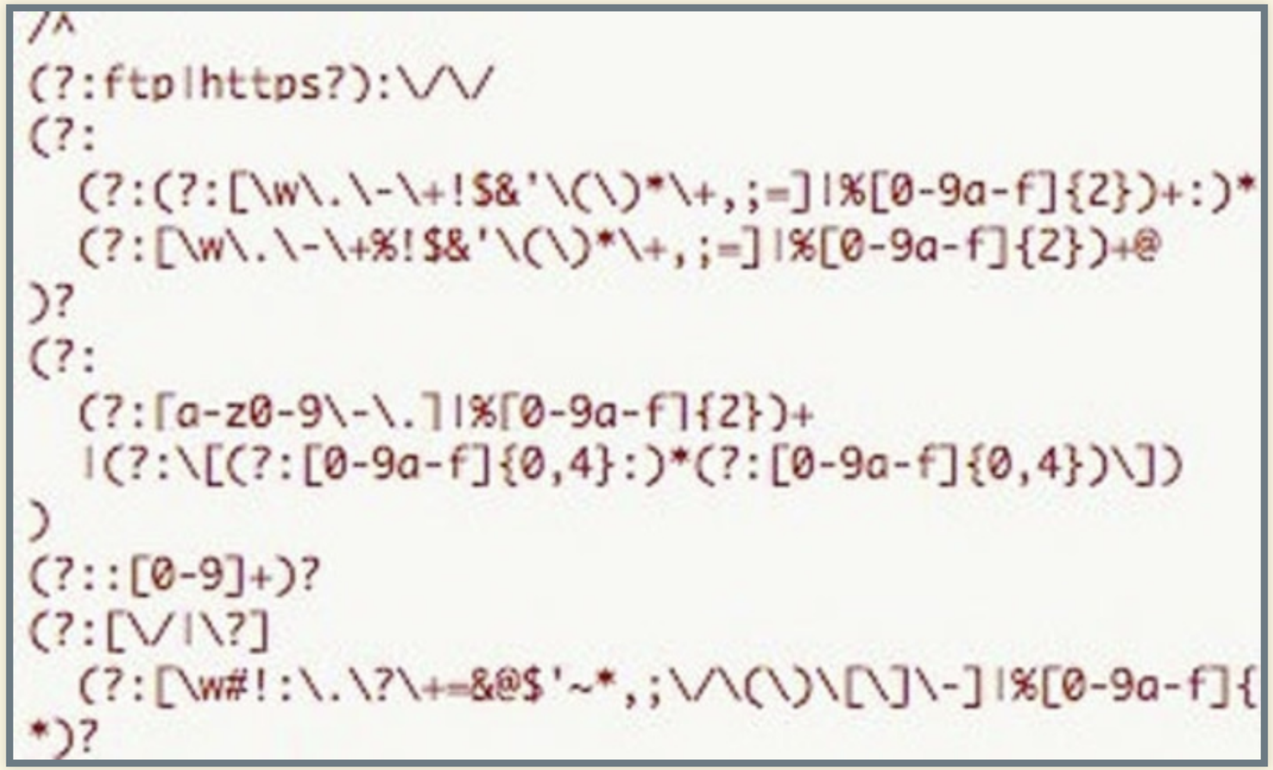
RUBY HAS GREAT TOOLS FOR REGULAR EXPRESSIONS

You can get by with them. You can especially get by with them in Ruby which draws its heritage from Perl, Sed, and Awk which made wonderful use of regexps.

```
@dot = @dot.gsub(/^.*->.*$/) do |line|
  line.gsub(/label = ".*"/) do |labels|
    labels.gsub(/\b\d+/) { |num| ASCII_MAP[num] || num }
  end
end
```

IRREGULAR EXPRESSIONS

zarro boogs found



```
/^
(?:ftp|https?):\/\/
(?:
  (?:(?:[\w\.\-\+!$&'\(\)*\+,;=]|%[0-9a-f]{2})+:)*
  (?:[\w\.\-\+%!$&'\(\)*\+,;=]|%[0-9a-f]{2})+@
)?
(?:
  (?:[a-z0-9\-\.]|%[0-9a-f]{2})+
  |(?:\[(?:[0-9a-f]{0,4}:)*(?:[0-9a-f]{0,4})\])
)
(?::[0-9]+)?
(?:[\/|\?]
  (?:[\w#!:\.\?\+=&@$'~*,;\/\(\)\[\]\-]|%[0-9a-f]{
*)?
```

SOMETIMES YOU WANT MORE CONTROL

I posit that this might be some sort of automaton.

FINITE AUTOMATA

- Have states and transitions.
- Change state based on sequence of inputs.
- **DFA** can be in only one state at a time.
- **NFA** can be in more than one state at a time.

EQUIVALENCE OF REGULAR EXPRESSIONS, NFAS, AND DFAS

It is possible to convert freely between regular expressions, deterministic finite automata, and nondeterministic finite automata. Given one, we can convert it to any of the other forms.

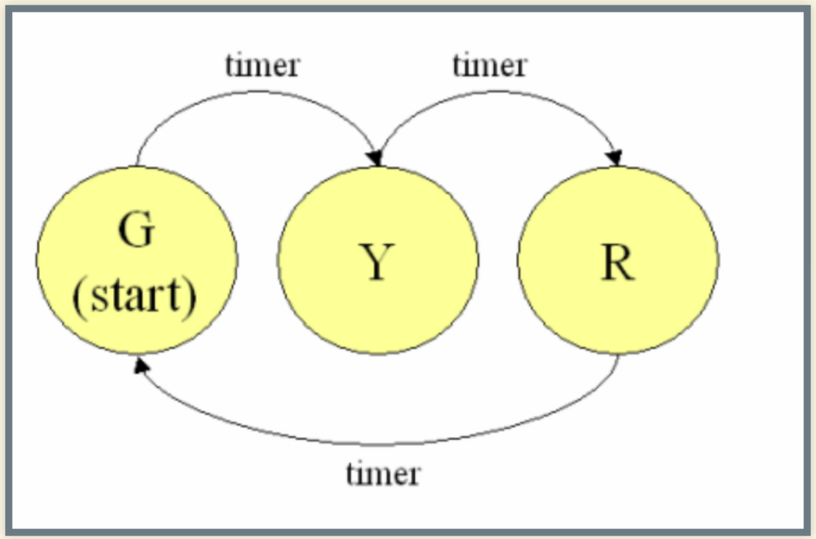
http://faculty.ycp.edu/~dhovemey/fall2008/cs340/notes/lecture3.html

THESE ARE ALL STATE MACHINES

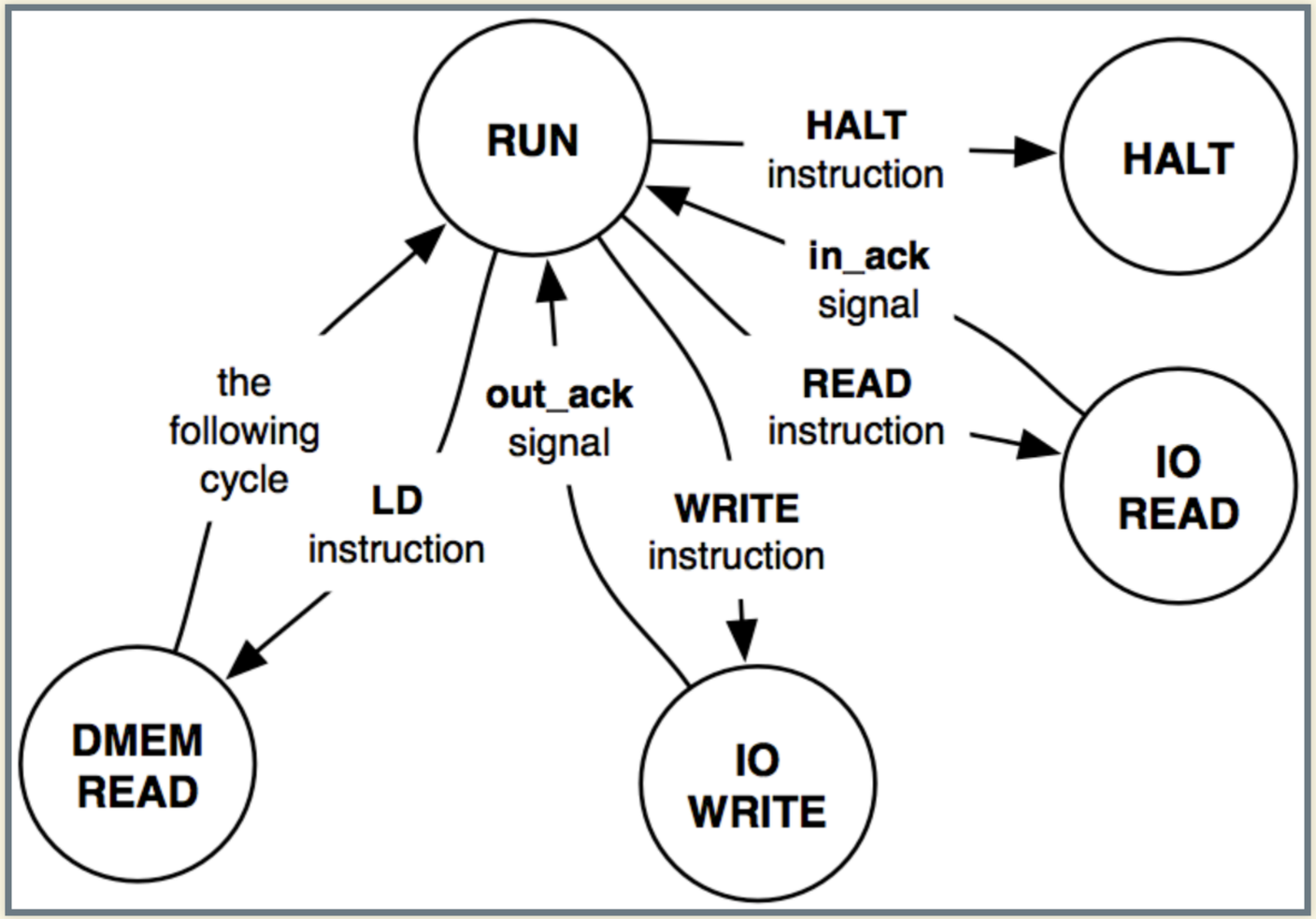
State machines are an important tool in computer programming, and Ragel is a wonderful tool for creating them.

STATE MACHINES ARE EVERYWHERE

THEY'RE IN YOUR STOPLIGHT



THEY RUN YOUR CPU



THERE ARE EXAMPLES EVERYWHERE

- watch with timer
- vending machine
- traffic light
- bar code scanner
- gas pumps
- number classification

THE CAT'S MEOW?

State machines are great for many reasons. They are simple to understand, and there has been a great deal of research around finite automata and state machines. With the right approach they can also produce code that is faster, easier to maintain, and more correct and thus more secure.

STILL NOT CONVINCED?

Rather than me trying to convince you that they're useful,
let's just talk about them for a bit and see where we end up.

LET'S GO OVER SOME VOCABULARY

START STATE

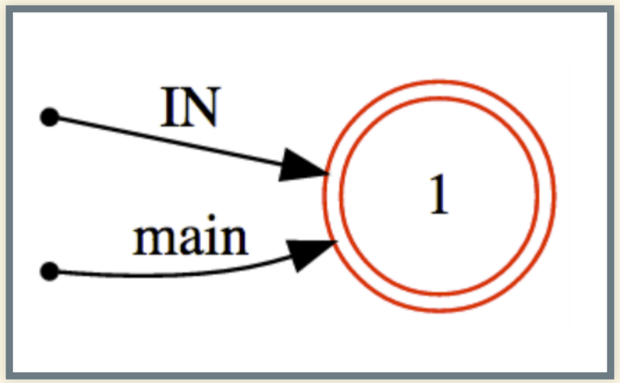
This is the initial state of a machine.

(S0)

ACCEPT STATE

In this state, the machine is said to have "accepted" the input.

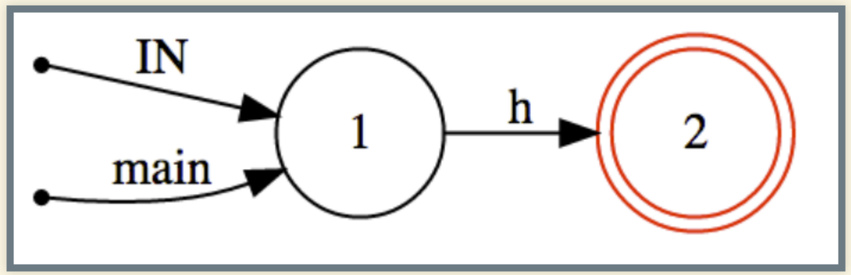
(double circle)



TRANSITION

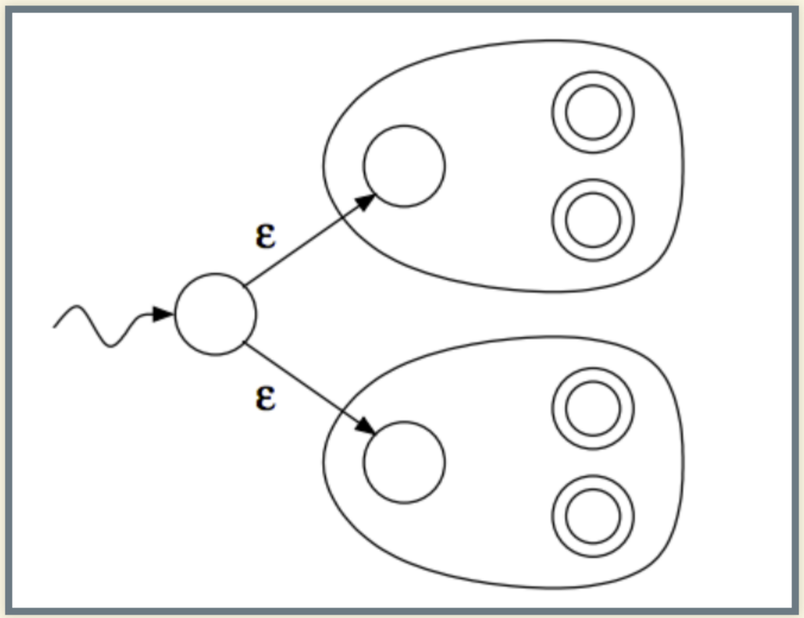
Upon consuming a single character, the machine can move from one state to another.

(labelled arrow)



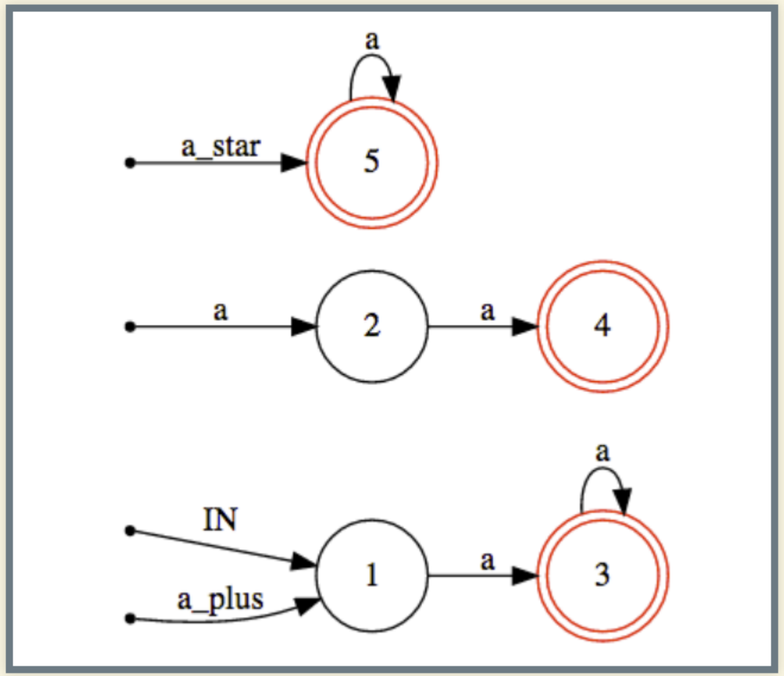
EPSILON TRANSITION

Allows an automaton to change its state spontaneously, i.e. without consuming an input symbol.



SIMPLE MACHINES

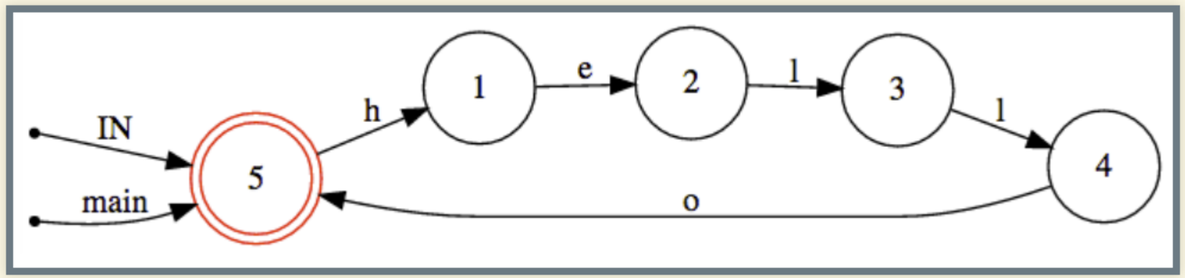
'a', 'a'*, 'a'+



MORE COMPLEX

'hello'*

Zero or more hellos.



WHAT'S THE BIG DEAL?

These just look like regular expressions.

WHAT IS RAGEL EXACTLY?

Ragel is a finite-state machine compiler with output support for C, C++, C#, Objective-C, D, Java, OCaml, Go, and Ruby source code. It supports the generation of table or control flow driven state machines from regular expressions and/or state charts and can also build lexical analysers via the longest-match method. Ragel specifically targets text parsing and input validation.

https://en.wikipedia.org/wiki/Ragel

STATE MACHINE GENERATION

Ragel supports the generation of table or control flow driven state machines from regular expressions and/or state charts and can also build lexical analysers via the longest-match method. A unique feature of Ragel is that user actions can be associated with arbitrary state machine transitions using operators that are integrated into the regular expressions. Ragel also supports visualization of the generated machine via graphviz.

https://en.wikipedia.org/wiki/Ragel

HOW DO YOU PRONOUNCE IT?

- RAY-gull?
- RAY-jul?
- RAH-gull?
- RAH-jul?

LET'S GET IT FROM THE HORSE'S MOUTH

ADRIAN D. THURSTON CREATED RAGEL

https://www.mail-archive.com/ragel-users@complang.org/msg00344.html

```
Re: [ragel-users] pronounciation
Adrian Thurston Sat, 10 Apr 2010 09:02:10 -0700

Hi Landon,

I usually say something like "rah-ghel." I had no
phonetic basis for it when I picked it. I just took my nickname
"Age" and wrapped it in the R and L of regular languages. I've
since learned that it means "man" in Arabic.

Adrian

Landon Cox wrote:
> Hi Adrian and others,
>
> Simple question:  What is the pronunciation of "Ragel"?
>
```

WELL, DARN. I'VE BEEN PRONOUNCING IT WRONG FOR QUITE SOME TIME!

RAGEL IS A DSL FOR CREATING STATE MACHINES

It is espcially useful for parsing protocols and data formats.
(HTTP, XML, JSON, CSS, etc...)

LET'S LOOK AT THE DSL

GENERAL STRUCTURE OF A RAGEL FILE

- Mostly in the host language
- has a .rl extension (simple.rl)
- %% is used for inline statements
- %%{ is used for multiline statements }%%

```
%%{
  machine foobar;
  main := 'foobar';
}%%

%%write init;
%%write exec;
```

NAMING A MACHINE

With named machines, you can spread a machine's statements across several files or include common sections.

```
machine phone_parser;
```

MACHINE DEFINITION

You define a machine using the equals operator.

```
<name> = <expression>;
```

This allows it to be referenced later.

MACHINE INSTANTIATION

This causes the actual generation of the referenced set of states.

```
<name> := <expression>;
```

Each instantiation generates a distinct set of states.

FILE INCLUSION AND IMPORT

You can include and import definitions from other files.

```
include FsmName "inputfile.rl";

import "inputfile.h";
```

These can help you keep things organized. See the manual for the specific semantics of each.

WHITESPACE

Any amount of whitespace can separate tokens.

COMMENTS

```
# is used for single line comments
```

LITERALS

Literals are contained in quotes, regexp slashes, or brackets for groupings.

```
""    # string
''    # string
//    # regexp
[]    # union
```

ESCAPE CHARACTERS

```
\0    null
\a
\b    backspace
\t    tab
\n    newline
\v    vtab
\f    formfeed
\r    carriage return
```

You can escape the end of a line with a \ (as in shell scripting)

HOST LANGUAGE CODE

Braces are used to delimit host language code

```
%%{
   { puts "I am ruby" }
}%%
```

NUMBERS

Integers and hexadecimals can be used to refer to numbers.

```
[+-]?[0-9]+             # integers    (-23432, +23423, 23423)
0x[0-9A-fa-f]           # hexadecimal (0xABD)
```

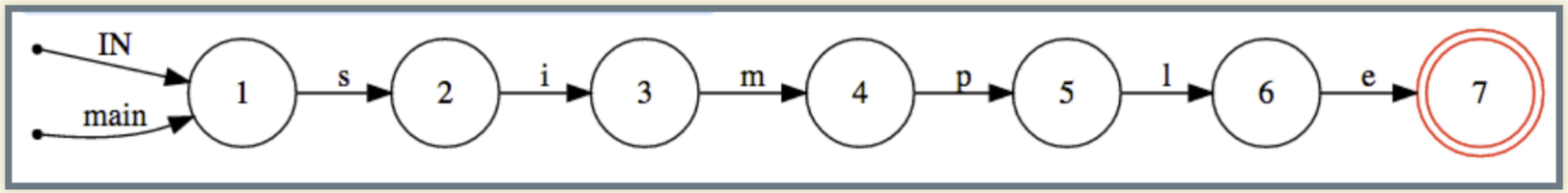
KEYWORDS

```
access
action
alphtype
getkey
write
machine
include
```

CONCATENATION LITERAL

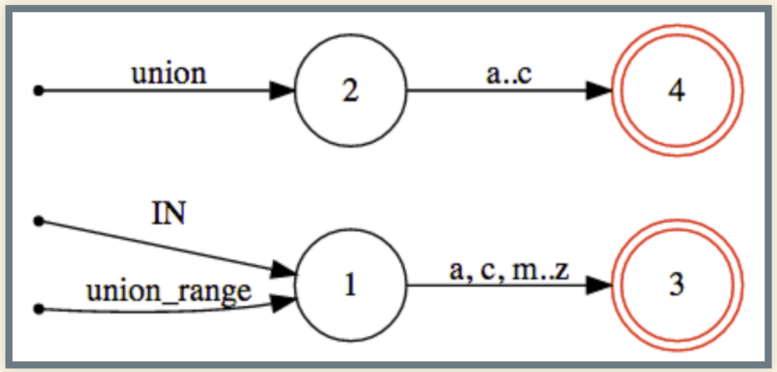
Match on a sequence of letters.

```
%%{
  machine simple;
  main := 'simple';
}%%
```



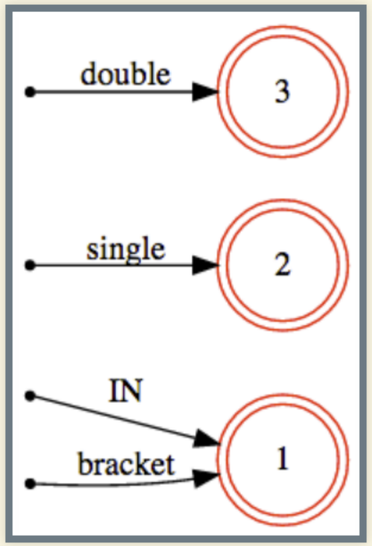
UNION EXPRESSION

```
%%{
  machine union_range;
  union := [abc];
  union_range := [acm-z];
}%%
```



ZERO LENGTH MACHINES

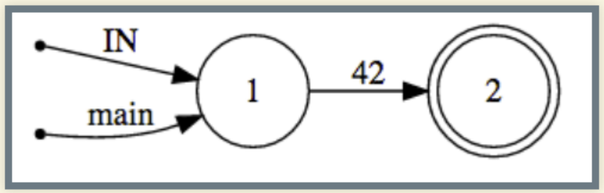
```
%%{
  machine zero_length_machines;
  single := '';
  double := "";
  bracket := [];
}%%
```



NUMERICAL LITERAL

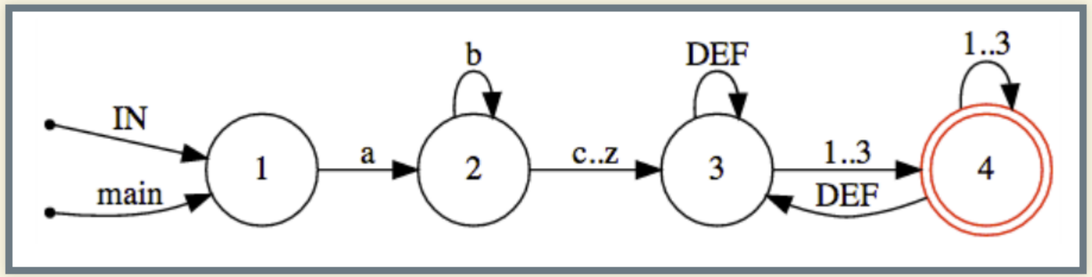
Produces a two-state machine with one transition on the given value, which can be given in decimal or hexadecimal.

```
%%{
  machine numerical_literal;
  main := 42;
}%%
```



REGULAR EXPRESSION

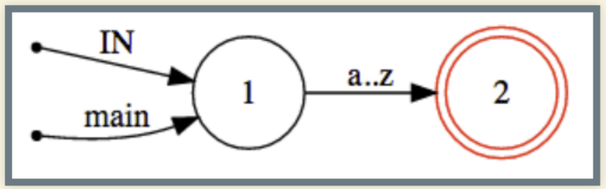
```
%%{
  machine regexp2;
  main := /ab*[c-z].*[123]/;
}%%
```



RANGE EXPRESSION

Matches any character between 'a' and 'z' inclusive.

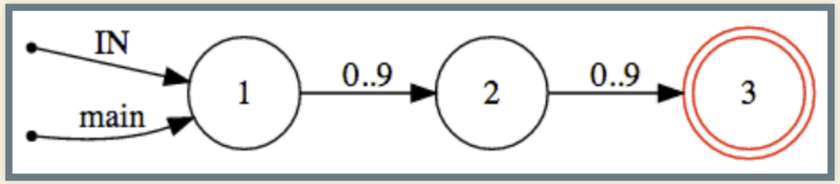
```
%%{
  machine range;
  main := 'a'..'z';
}%%
```



VARIABLE NAME

Inserts the machine referenced by this name.

```
%%{
  machine variable_name;
  secret_code = [0-9]{2};
  main := secret_code;
}%%
```



BUILTIN MACHINES

```
any    — Any character in the alphabet.
ascii  — Ascii characters. 0..127
extend — Ascii extended characters.
alpha  — Alphabetic characters. [A-Za-z]
digit  - Digits. [0-9]
alnum  — Alpha numerics. [0-9A-Za-z]
lower  — Lowercase characters. [a-z]
upper  — Uppercase characters. [A-Z]
xdigit - Hexadecimal digits. [0-9A-Fa-f]
cntrl  — Control characters. 0..31
graph  — Graphical characters. [!-~]
print  — Printable characters. [ -~]
punct  — Punctuation.
space  - Whitespace. [\t\v\f\n\r ]
zlen   - Zero length string. ""
empty  - Empty set. Matches nothing. ^any
```

BUILDING BLOCKS

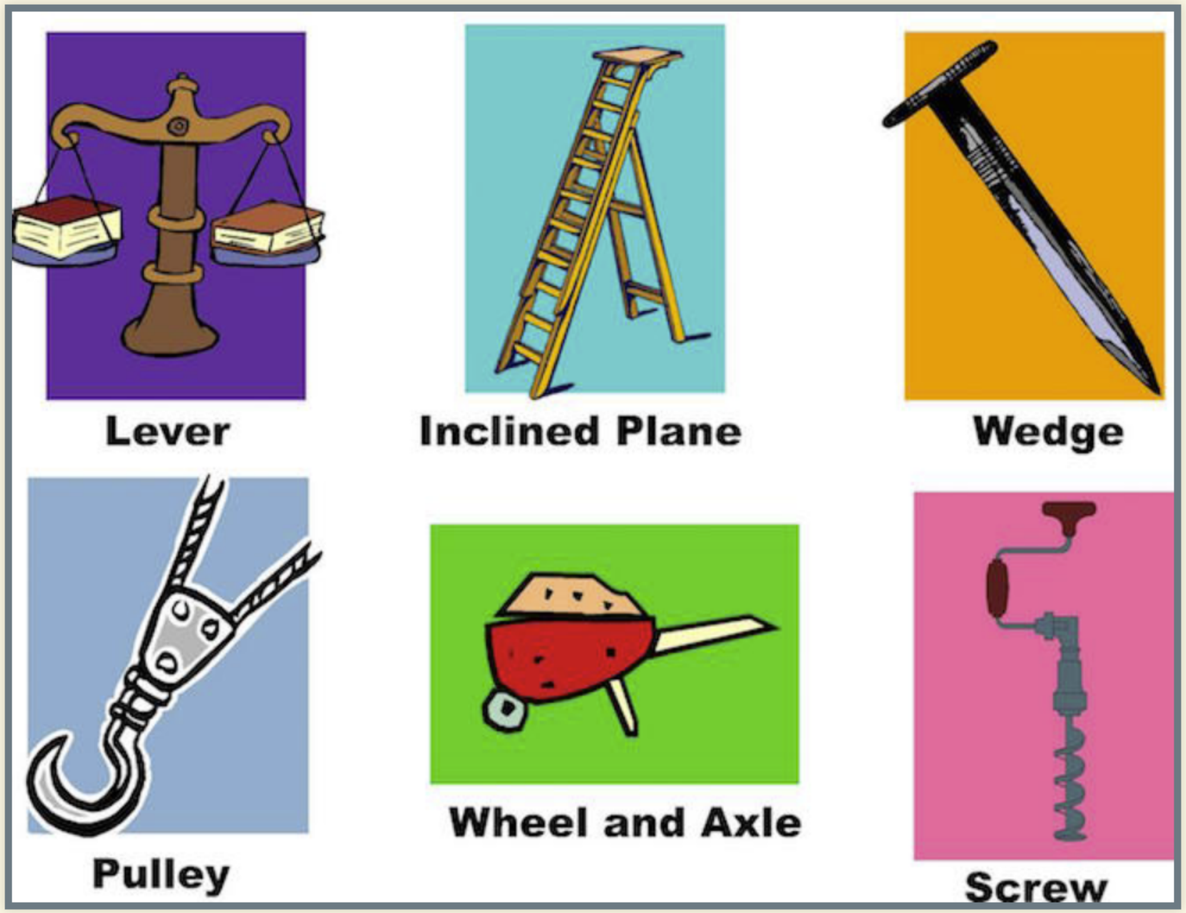
We have simple machines now.

Like levers, wedges, wheels, and pulleys.

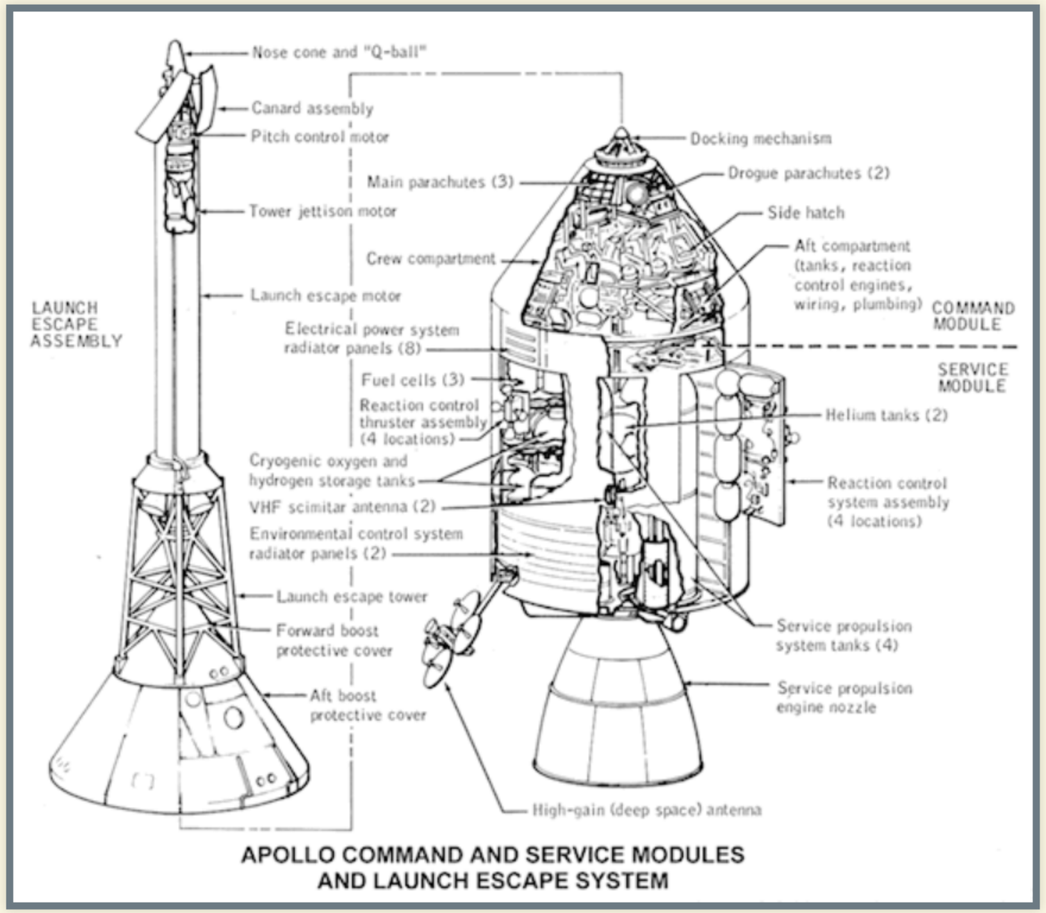
But let's not stop here.

From simple machines we can make complex machines.

SIMPLE



COMPLEX



APOLLO COMMAND AND SERVICE MODULES AND LAUNCH ESCAPE SYSTEM

ISN'T SHE CUTE?

COMPOSITION

Ragel's DSL allows you to take these simple machines, and through some basic operators, combine those into bigger machines, and then combine those into BIGGER machines.

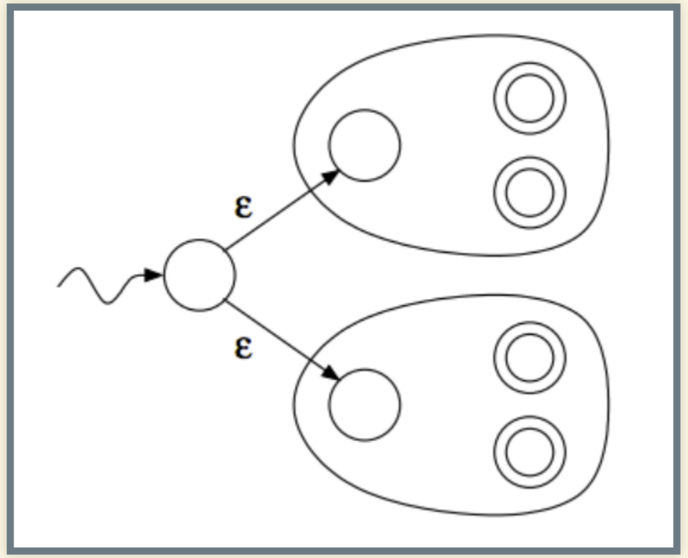
COMPOSITIONAL OPERATORS

```
expr | expr    - union
expr & expr    - intersection
expr - expr    - difference
expr -- expr   - strong difference
expr . expr    - concatenation
expr*          - kleene star
expr+          - one or more repetion
expr?          - optional
expr{n}        - exactly N copies of expr
expr{n,}       - Zero to N copies of expr
expr{,m}       - N or more copies of expr
expr{n,m}      - N to M copies of expr
!expr          - negation
^expr          - character-level negation
```

UNION

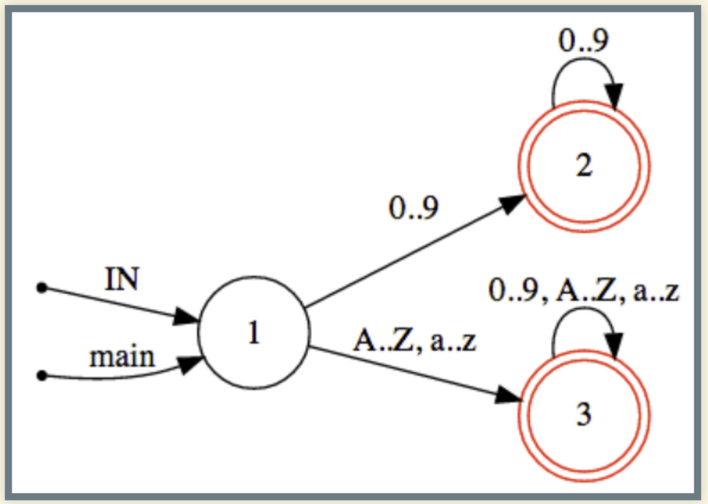
Matches any string in machine one or machine two

```
expr | expr
```



UNION EXAMPLE

```
%%{
  machine union2;
  # Hex digits, decimal digits, or identifiers
  main := digit+ | alpha alnum*;
}%%
```



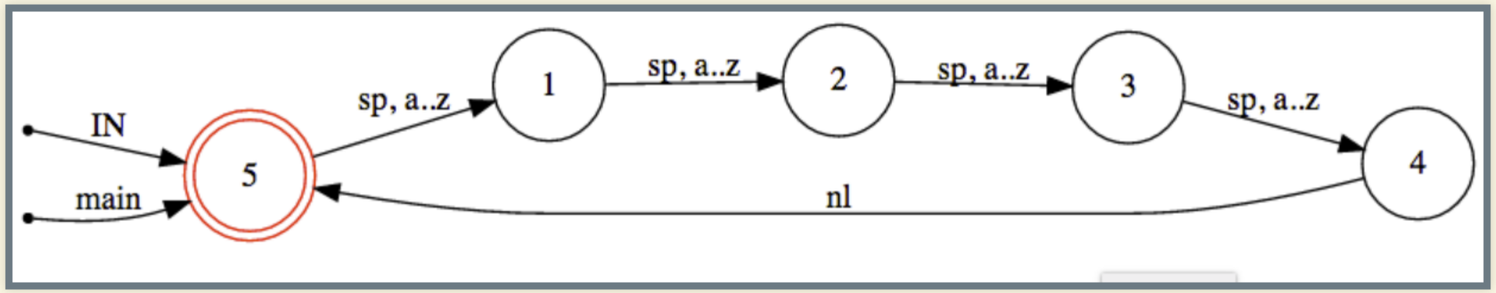
INTERSECTION

Matches any string that is in both machine one and two.

```
expr & expr
```

INTERSECTION EXAMPLE

```
%%{
  machine intersection;
  main := /[^\n][^\n][^\n][^\n]\n/* & (/[a-z][a-z]*/ | [ \n])**;
}%%
```



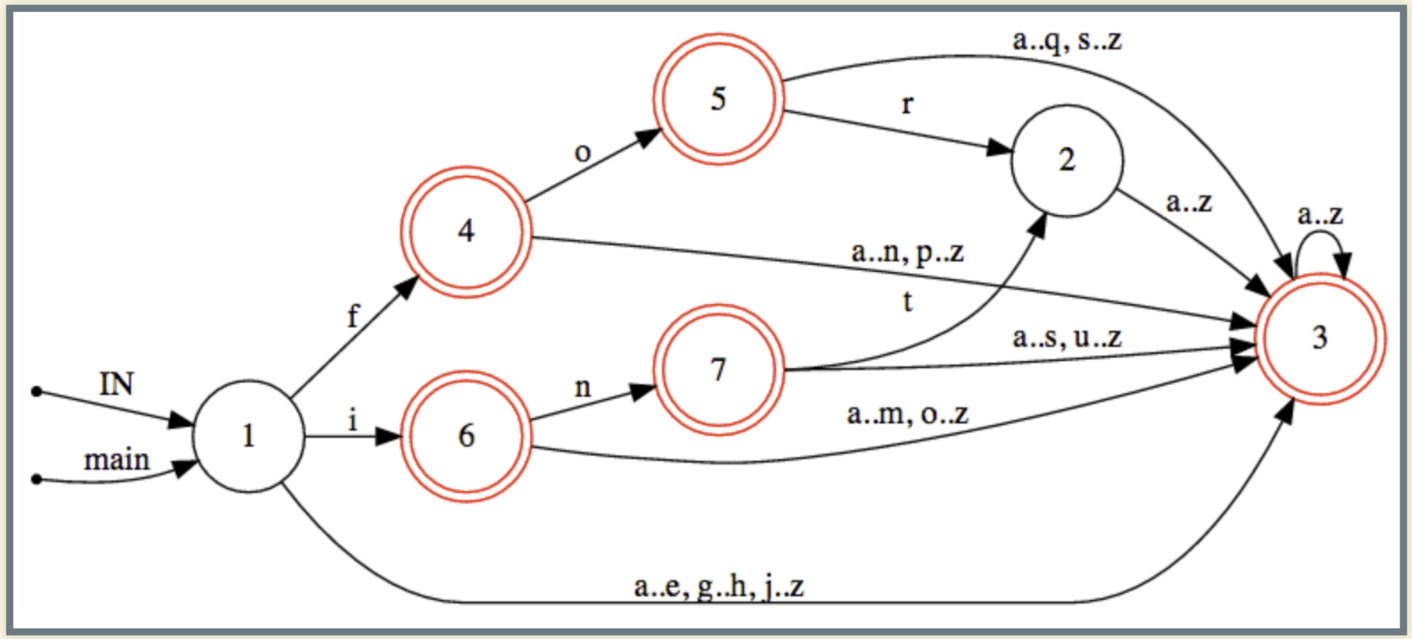
DIFFERENCE

Matches strings in machine one but not in machine two

```
expr - expr
```

DIFFERENCE EXAMPLE

```
%%{
  machine difference;
  # Subtract keywords from identifiers.
  main := /[a-z][a-z]*/ - ( 'for' | 'int' );
}%%
```



STRONG DIFFERENCE

Matches any string of the first machine that does not have any string of the second machine as a substring.

```
expr -- expr
```

Equivalent to:

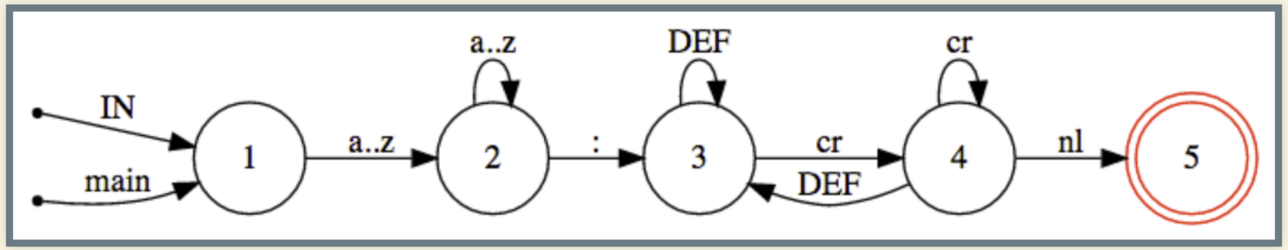
```
expr - ( any* expr any* )
```

STRONG DIFFERENCE EXAMPLE

Used to excluded CRLF from a sequence.

```
%%{
  machine strong_difference;
  crlf = '\r\n';
  main := [a-z]+ ':' ( any* -- crlf ) crlf;
}%%
```

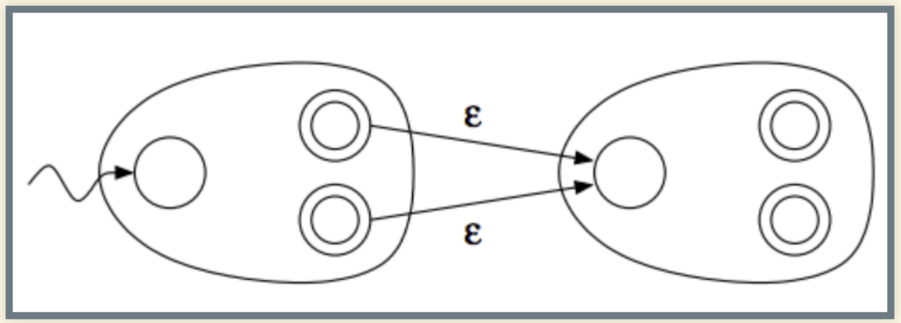
The DEF transition is taken if no other transition can be taken.



CONCATENATION

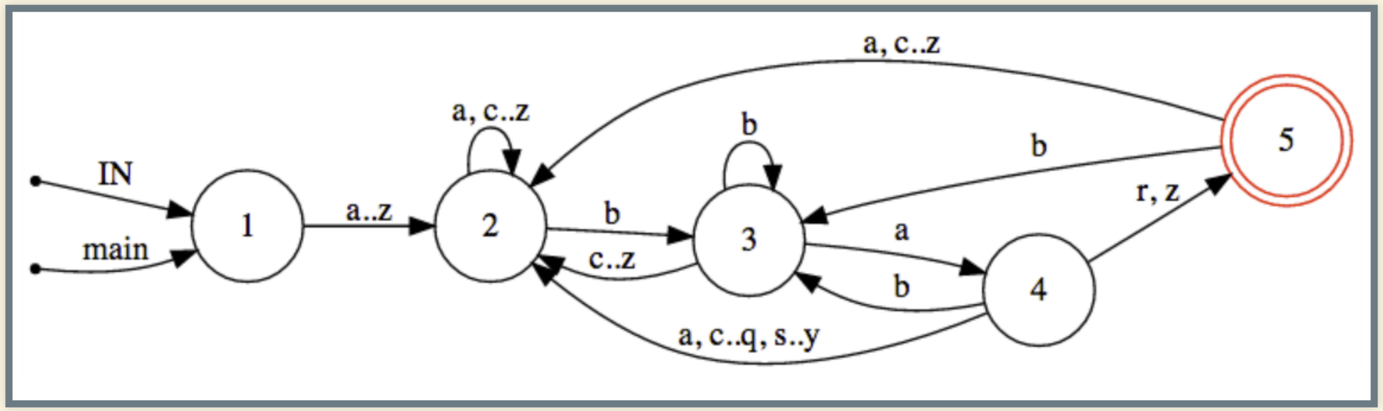
Matches all the strings in machine one followed by all the strings in machine two.

```
expr . expr
```



CONCATENATION EXAMPLE

```
%%{
  machine concatenation;
  #concatenation
  main := [a-z]+ . /ba[rz]/;
}%%
```



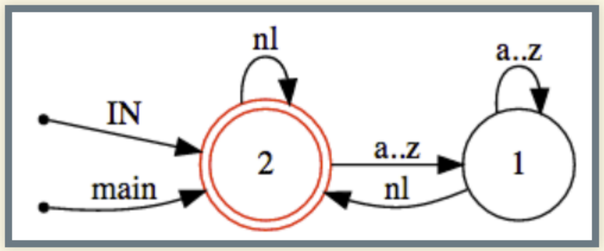
KLEENE STAR

Match zero or more repetitions of the machine it is applied to.

```
expr*
```

KLEENE STAR EXAMPLE

```
%%{
  machine kleene_star;
  # Match any number of lines with only lowercase letters.
  main := /[a-z]*\n/*;
}%%
```



ONE OR MORE REPETION

Produces the concatenation of the machine with the kleene star of itself. The result will match one or more repetitions of the machine.

```
expr+
```

Equivalent to:

```
expr . expr*
```

OPTIONAL

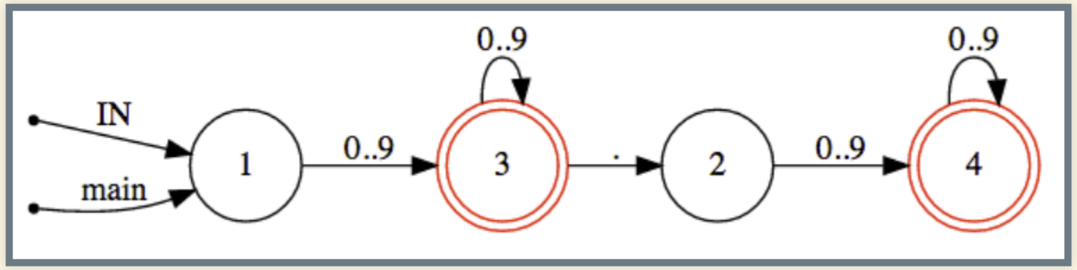
```
expr?
```

Equivlaent to:

```
expr | ''
```

OPTIONAL EXAMPLE

```
%%{
  machine optional;
  # Match integers or floats.
  main := digit+ ('.' digit+)?;
}%%
```



REPETITION

```
expr{n} - exactly N copies of expr
expr{n,} - Zero to N copies of expr
expr{,m} - N or more copies of expr
expr{n,m} - N to M copies of expr
```

NEGATION

Matches any string not matched by the given machine.

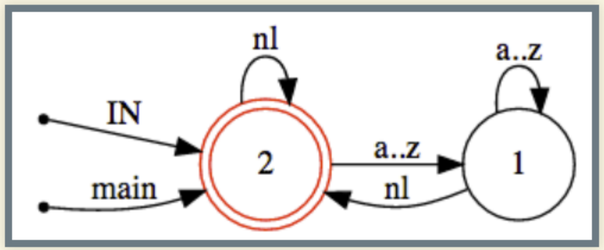
```
!expr
```

Equivalent to:

```
any* - expr
```

NEGATION EXAMPLE

```
%%{
  machine negation;
  # Accept anything but a string beginning with a digit.
  main := !( digit any* );
}%%
```



CHARACTER- LEVEL NEGATION

```
^expr
```

Equivalent to:

```
any - expr
```

STATE MACHINE MINIMIZATION

- Reduces the number of states through optimization
- Merges equivalent states
- On by default (can be disabled with -n)

USER ACTIONS

Composition is definitely cool and useful. But on top of that, Ragel gives you embedded actions. This is where you take all the composition and really make it sing, on key.

EMBEDDING ACTIONS

```
action <action_name> {
  # host code here
  count += 1
}
```

Actions can be referenced by name or embedded inline.

TRANSITIONS

Transitions come in four classes, and actions can be attached to any of them.

ENTERING TRANSITION

```
> operator

expr > action_name
expr >{ puts "entering" }
```

Embeds an action into all transitions leaving the "start state"

FINISHING TRANSITION

```
@ operator

expr @ action_name
expr @{ puts "finishing" }
```

Embeds an action into all transitions going into a "final state"

ALL TRANSITION

```
$ operator

expr $ action_name
expr ${ puts "transitioned" }
```

Embeds an action into all transitions, regardless of type (useful for debugging).

LEAVING TRANSITION

```
% operator
expr % action_name
expr %{ puts "leaving" }
```

Embeds an action into all transitions leaving the machine from a "final state"

EMBEDDING OPERATORS CAN GET FANCY

See the manual for more information on these.

- To-State Actions
- From-State Actions
- EOF Actions
- Global Error Actions (for error recovery)
- Local Error Actions (for error recovery)

NONDETERMINISM

One of the problems you will run into is when the trailing match of one machine is the same as the leading match of the next machine. In these cases, the state will be stuck in the first machine and never transition to the next machine.

NONEDETERMINISM EXAMPLE

The \n in ws will preven the final \n from matching.

```
ws = [\n\t ];
line = word $first ( ws word $tail )* '\n';
lines = line*;
```

The solution here is simple: exclude the newline character from the ws expression.

```
ws = [\t ];
line = word $first ( ws word $tail )* '\n';
lines = line*;
```

AMBIGUITY PROBLEMS

Here's an incorrect way to parse C language comments:

```
comment = '/*' ( any @comm )* '*/';
main := comment ' ';
```

The `any` will prevent the trailing */ from ever matching.

THIS WORKS BUT IT'S UGLY

```
comment = '/*' ( ( any @comm )* - ( any* '*/' any* ) ) '*/';
```

We have to carefully exclude things to get it to match.

THIS IS GETTING COMPLICATED!

But there's a solution.

Ragel lets you embed priorities into transitions to deal with ambiguity.

SETTING PRIORITIES MANUALLY

```
expr > int — Sets starting transitions to have priority int.

expr @ int — Sets transitions that go into a final state to have prior

expr $ int — Sets all transitions to have priority int.

expr % int — Sets leaving transitions to have priority int.
```

NAMESPACING PRIORITIES

When machines are combined, you can get odd interactions if you don't namespace the priorities.

```
expr > (name, int) — Starting transitions.
expr @ (name, int) — Finishing transitions (into a final state).
expr $ (name, int) — All transitions.
expr % (name, int) — Leaving transitions.
```

GUARDED OPERATIONS

Thinking in priorities is hard.

Fortunately Ragel provides some better mechanisms for us to use.

These are called "guarded concatenations"

FINISH-GUARDED CONCATENATION

A higher priority is then embedded into the transitions of the second machine that enter into a final state.

```
comment = '/*' ( any @comm )* :>> '*/';
```

This is much simpler to visualize and reason about.

ENTRY-GUARDED CONCATENATION

A higher priority is given to the second machine.

```
expr :> expr
```

```
# Leave the catch-all machine on the first character of FIN.
main := any* :> 'FIN';
```

Equivalent to:

```
expr $(unique_name,0) . expr >(unique_name,1)
```

LEFT-GUARDED CONCATENATION

The left hand machine has a higher priority.

```
expr <: expr
```

For stripping leading space:

```
main := ( ' '* >start %fin ) <: ( ' ' $ws | [a-z] $alpha )*;
```

LONGEST-MATCH KLEENE STAR

This has a higher priority for staying in the machine rather than wrapping around again.

```
expr**
```

```
# Repeat tokens, but make sure to get the longest match.
Main := (
  lower ( lower | digit )* %A | digit+ %B |
  ''
)**;
```

SCANNERS

Scanners are a common thing to build with Ragel, so it has special support for them.

```
<machine_name> := |*
  pattern1 => action1;
  pattern2 => action2;
  ...
*|;
```

SCANNER EXAMPLE

Tokenizing the contents of a header field.

```
%%{
  word = [a-z]+;
  head_name = 'Header';
  header := |*
    word;
    ' ';
    '\n' => { fret; };
  *|;
  main := ( head_name ':' @{ fcall header; } )*;
}%%
```

PROTOCOL PARSING

Ragel is well suited for protocol parsing.

Mapping an RFC onto a Ragel specification is pretty straight-forward.

Puma has a good example of this (heritage is the original mongrel parser by Zed Shaw)

https://github.com/puma/puma/blob/master/ext/puma_http11/http11_parser_common.rl

STATE CHARTS

Ragel allows you to specify states and transitions directly if you desire extreme customization.

This is like programming in the "assembly" of Ragel.

There are a few new operators for this.

STATE CHART EXAMPLE

Parsing XML CDATA.

```
action bchar { buff( fpc ); }
action bbrack1 { buff( "]" ); }
action bbrack2 { buff( "]]" ); }
CDATA_body =
start: (
']' -> one |
     (any-']') @bchar ->start
),
one: (
     ']' -> two |
     [^\]] @bbrack1 @bchar ->start
),
two: (
     '>' -> final |
     ']' @bbrack1 -> two |
     [^>\]] @bbrack2 @bchar ->start
);
```

PARSER MODULARIZATION

```
action return { fret; }
action call_date { fcall date; }
action call_name { fcall name; }

# A parser for date strings.
date := [0-9][0-9] '/'
        [0-9][0-9] '/'
        [0-9][0-9][0-9][0-9] '\n' @return;

# A parser for name strings.
name := ( [a-zA-Z]+ | ' ' )** '\n' @return;

# The main parser.
headers =
    ( 'from' | 'to' ) ':' @call_name |
    ( 'departed' | 'arrived' ) ':' @call_date;
```

PARSING RECURSIVE STRUCTURES

The general trick is to store some context about where you are in your recursive structure, say in a stack called @nestings, and push/pop to it as appropriate. When it comes time to call `fret`, you can examine your @nestings and steer the parser as deemed appropriate.

IMPLEMENTING LOOKAHEAD

This is possible. The trick here is to match deeper than you need, then use `fhold` to walk the parser back a few characters.

RAGEL INTERNALS

Ragel uses several variables for state. You can twiddle them in actions.

```
* data - the buffer where you should store the data
* p    - start index in data where Ragel is matching
* pe   - end index of data (Ragel should ignore anything past this)
* ts   - in a scanner, token start
* te   - in a scanner, token end
* act  - in a scanner, last matched action
```

Those are the major ones. See the manual for more details.

RAGEL OPERATION (ROUGHLY)

1. Starts in state 0
2. Feed it data, updating `p` and `pe` as appropriate
3. Run the `%%exec` loop
4. Characters move it through a state
5. It consumes `p -> pe` from `data`
6. If `cs` is `>= first_final_state` (final states are last)
   then you have “admitted” the string

RAGEL OPERATION (SCANNERS)

Scanners are a bit more involved, but not that much more.

1. Use a stack to track states
2. Use ts -> te to track where they are in a match
3. Use the stack to backtrack when necessary
4. Keep matching repeatedly until we are done
5. Longest match wins
6. It's useful to create helper methods (`emit`, `current_buffer`, `current_match(start, end)`)

RAGEL STRING EXTRACTION

To pull out the data you care about, while you are parsing, you will do something like this:

```
* >mark { puts "mark the beginning a pattern" ; @mark = @data[p] }
* %emit l{ puts "save the currently matched pattern" ; @things << data[@
```

HOST LANGUAGES

Several host languages are available.

```
host language:
   -C    The host language is C, C++, Obj-C or Obj-C++ (default)
   -D    The host language is D
   -Z    The host language is Go
   -J    The host language is Java
   -R    The host language is Ruby
   -A    The host language is C#
   -O    The host language is OCaml
```

CODE STYLES

Ragel uses your .rl code to compute the set of states and transitions. From that, it can generate code in a number of different styles.

```
code style: (C/D/Java/Ruby/C#/OCaml)
   -T0                  Table driven FSM (default)

code style: (C/D/Ruby/C#/OCaml)
   -T1                  Faster table driven FSM
   -F0                  Flat table driven FSM
   -F1                  Faster flat table-driven FSM

code style: (C/D/C#/OCaml)
   -G0                  Goto-driven FSM
   -G1                  Faster goto-driven FSM

code style: (C/D)
   -G2                  Really fast goto-driven FSM
   -P<N>                N-Way Split really fast goto-driven FSM
```

CODE STYLES PERFORMANCE

Each of these has different visual organization and performance characteristics. In languages like C, this can boil down to heavily-optimized GOTO statements in a single while loop. It's fast and cpu-cache friendly.

MULTI-LANGUAGE

It's possible to have a single Ragel definition that uses import semantics to allow implementing the actions in different languages using the same parent Ragel file. See the http11 parser in puma for details (C and Java)

https://github.com/puma/puma/tree/master/ext/puma_http11

RAGEL IN C

It's also possible to prototype in Ruby, then convert it to a C module for super speed. Ragel supports several output formats so you can do this port rather easily.

Again, see mongrel or puma for ideas.

RAGEL DIRECTIVES - INIT

Initializes the data buffer and sets the current state:

```
%write init;
```

RAGEL DIRECTIVES - DATA

Writes out definitions of the state and transition data:

```
%%write data;
```

RAGEL DIRECTIVES - EXEC

Writes out the code that processes the data buffer using the state and transition data

```
%%write exec;
```

%%WRITE DATA;

```
class << self
  attr_accessor :_hello_key_offsets
  private :_hello_key_offsets, :_hello_key_offsets=
end
self._hello_key_offsets = [
  0, 0, 1, 2, 3, 4
]

class << self
  attr_accessor :_hello_trans_keys
  private :_hello_trans_keys, :_hello_trans_keys=
end
self._hello_trans_keys = [
  101, 108, 108, 111, 104, 0
]
# LOTS MORE LIKE THIS
# ...
```

%%WRITE INIT;

```
begin
  p ||= 0
  pe ||= data.length
  cs = simple_start
  top = 0
end
```

%%WRITE EXEC;

```
 begin
  _klen, _trans, _keys = nil
  _goto_level = 0
  _resume = 10
  _eof_trans = 15
  _again = 20
  _test_eof = 30
  _out = 40
  while true
  _trigger_goto = false
  if _goto_level <= 0
  # LOTS MORE LIKE THIS
  # ...
  # ...
end
```

INSTALLATION

```
brew install ragel
```

GENERATING THE RUBY

simple.rl -> simple.rb

```
ragel -R simple.rl -o simple.rb
```

VISUALIZATION

You can get a dotviz graph.

```
ragel -V simple.rl > simple.dot
dot -Tsvg simple.dot -o simple.svg
```

CALLING FROM RUBY

To run on a single buffer of String data:

```
def ragel_parse(data)
  data = data.unpack("c*")
  eof = data.length
  tokens = []

  %% write init;
  %% write exec;

  puts tokens.inspect
end
```

RAGEL PLAYGROUND

I created a tool in Volt to do some basic visualization.

https://github.com/ijcd/ragel_playground

It's definitely a work in progress, but feel free to try it out.

DEMOS

1. hello parser
2. args parser
3. args state chart

TALK TO ME, BABY!

@ijcd

github.com/ijcd

https://github.com/ijcd/ragel_playground

https://github.com/ijcd/rubyconf-2015-ragel

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