Ruby Regular Expressions

AND FINITE AUTOMATA...

Why Learn Regular Expressions?

- RegEx are part of many programmer's tools
  - vi, grep, PHP, Perl
- They provide powerful search (via pattern matching) capabilities
- Simple regex are easy, but more advanced patterns can be created as needed (use with care, may not be efficient)
- ruby syntax closely follows Perl 5

Handy resource: rubular.com

From: http://www.websiterepairguy.com/articles/re/12_re.html

Outline

- Regular expression basics
  - how to create a pattern
  - how to match using =~
- Finite state automata
- Working with match data
- Working with named capture
- Regular expression objects
  - Regexp.new/Regex.compile/Regex.union

Regular Expressions

THE BASICS

Regular Expression patterns

- Constructed as
  - /pattern/
  - /pattern/options
  - %r{pattern}
  - %r{pattern}options
- Options provide additional info about how pattern match should be done, for example:
  - i – ignore case
  - m – multiline, newline is an ordinary character to match
  - u,e,s,n – specifies encoding, such as UTF-8 (u)

From: http://www.ruby-doc.org/docs/ProgrammingRuby/html/language.html#UJ

Pattern Matching

- =~ is pattern match operator
- string =~ pattern

OR

- pattern =~ string

- Returns the **index** of the first match
- Returns nil if no matches
  - Note that nil doesn't show when printing, but you can test for it

Literal characters

- /ruby/
- /ruby/i

Character classes

- /[0-9]/ match digit
- /[^0-9]/ match any non-digit
- /[aeiou]/ match vowel
- /[Rr]uby/ match Ruby or ruby

Anchors – location of exp

- /^Ruby/ # Ruby at start of line
- /Ruby$/ # Ruby at end of line
- /\ARuby/ # Ruby at start of line
- /Ruby\Z/ # Ruby at end of line
- /\bRuby\b/ # Matches Ruby at word boundary

- Using \A and \Z are preferred in Ruby (vs $ and ^)

http://stackoverflow.com/questions/577653/difference-between-a-z-and-in-ruby-regular-expressions

Alternatives

- /cow|pig|sheep/ # match cow or pig or sheep

Special character classes

- /./ #match any character except newline
- /./m # match any character, multiline
- /\d/ # matches digit, equivalent to [0-9]
- /\D/ #match non-digit, equivalent to [^0-9]
- /\s/ #match whitespace /[\r\t\n\f]/ \f is form feed
- /\S/ # non-whitespace
- /\w/ # match single word chars /[A-Za-z0-9_]/
- /\W/ # non-word characters

- NOTE: must escape any special characters used to create patterns, such as . \ + etc.

Repetition

- + matches one or more occurrences of preceding expression
  - e.g., /[0-9]+/ matches “1” “11” or “1234” but not empty string
- ? matches zero or one occurrence of preceding expression
  - e.g., /-?[0-9]+/ matches signed number with optional leading minus sign
- * matches zero or more copies of preceding expression
  - e.g., /yes[!]*/ matches “yes” “yes!” “yes!!” etc.

More Repetition

- /\d{3}/ # matches 3 digits
- /\d{3,}/ # matches 3 or more digits
- /\d{3,5}/ # matches 3, 4 or 5 digits

Non-greedy Repetition

- Assume s = <ruby>perl<
- /<.*>/ # greedy repetition, matches <ruby>perl<
- /<.*?>/ # non-greedy, matches <ruby>

- Where might you want to use non-greedy repetition?

Extra info, good to know but not on exams etc.

Grouping

() can be used to create groups

- /\D\d+/ # matches non-digit followed by digits, e.g., a1111
- /(\D\d)+/ # matches a1b2a3…
- ([Rr]uby(,\s)?)+
- Would this recognize (play with this in rubular)
  - “Ruby”
  - “Ruby, ruby”
  - “Ruby and ruby”
  - “RUBY”

Finite State Automata

A BRIEF INTRO

Finite Automata – formal definition

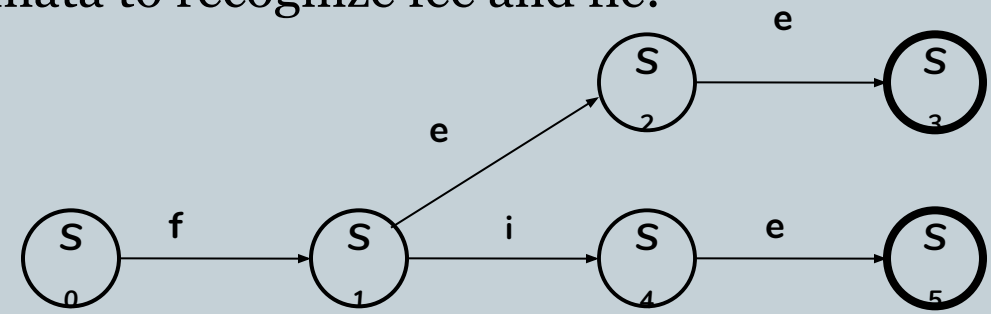
Formally a *finite automata* is a five-tuple(S,Σ,δ, s_0, S_F) where

- S is the set of states, including error state S_e. S must be finite.
- Σ is the alphabet or character set used by recognizer. Typically union of edge labels (transitions between states).
- $\delta(s,c)$ is a function that encodes transitions (i.e., character c in Σchanges to state s in S.)
- s_0 is the designated start state
- S_F is the set of final states, drawn with double circle in transition diagram

Theory of Computation view – we won't be too formal in csci400

Simple Example

Finite automata to recognize fee and fie:



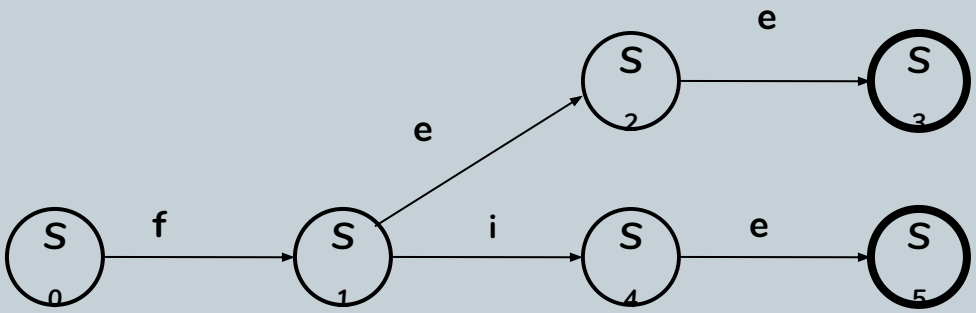
- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_e\}$
- $\Sigma = \{f, e, i\}$
- $\delta(s,c)$ *set of transitions shown above*
- $s_o = s_o$
- $S_F = \{ s_3, s_5\}$

Set of words accepted by a finite automata F forms a language L(F). Can also be described by regular expressions.

What type of program might need to recognize fee/fie/etc.?

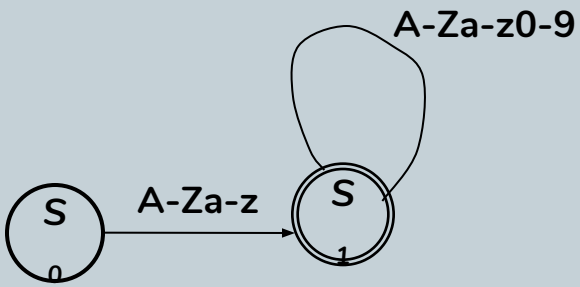
Finite Automata & Regular Expressions

- /fee|fie/
- /f[ei]e/
- Note: events/transitions are on the lines. Putting them in the nodes/circles is the #1 mistake.
- Note 2: end states should be in double lines, see next slide



Another Example: Pascal Identifier

- Pascal id is a letter followed optionally by letters and digits
- /[A-Za-z][A-Za-z0-9]*/



Quick Exercise

Go to rubular.com and review RegEx quick reference (same material as prior slides, but more concise)

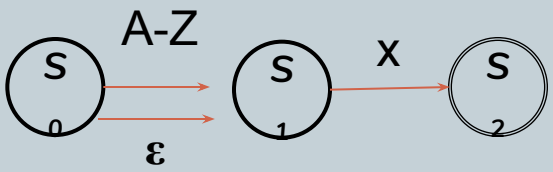
Look up the rules and create both FSA and RE to recognize:

- C identifier
- Perl identifier
- Ruby method identifier

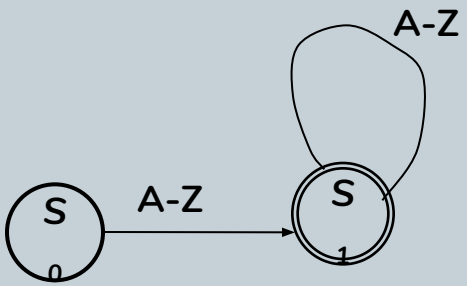
Turn in for class participation

RegExp to FSA

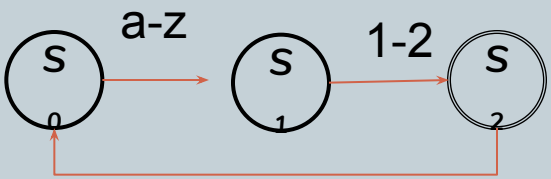
- ? = 0 or 1
- [A-Z]?x



- + = 1 or more
- [A-Z]+

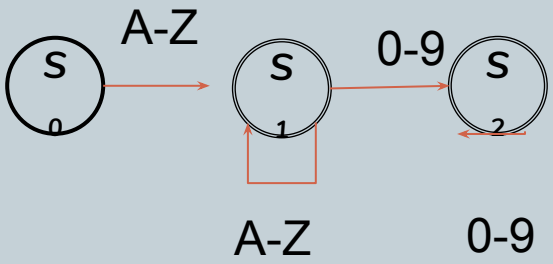


- () = group
- ([a-z][1-2])+



Reg Exp to FSA

- * = 0 or more
- [A-Z]+[0-9]*



RegExp in Ruby

SOME HANDY FEATURES

MatchData

- After a successful match, a MatchData object is created.
- Accessed as $~.
- Example:
  - "I love petting cats and dogs" =~ /cats/
  - puts "full string: #{$~.string}"
  - puts "match: #{$~.to_s}"
  - puts "pre: #{$~.pre_match}"
  - puts "post: #{$~.post_match}"

Named Captures

```
str = "Ruby 1.9"
if /(?<lang>\w+) (?<ver>\d+\.(\d+)+)/ =~ str
  puts lang
  puts ver
end
```

- Read more:
- http://blog.bignerdranch.com/1575-refactoring-regular-expressions-with-ruby-1-9-named-captures/
- http://www.ruby-doc.org/core-1.9.3/Regexp.html (look for Capturing)

Regexp class

- Can create regular expressions using Regexp.new or Regexp.compile (synonymous)

```
ruby_pattern = Regexp.new("ruby",
  Regexp::IGNORECASE)
puts ruby_pattern.match("I love Ruby!")
=> Ruby
puts ruby_pattern =~ "I love Ruby!“
=> 7
```

Regexp Union

- Creates patterns that match any word in a list

```
lang_pattern = Regexp.union("Ruby", "Perl", /Java(Script)?/)
puts lang_pattern.match("I know JavaScript")
=>
JavaScript
```

- Automatically escapes as needed

```
pattern = Regexp.union("()","[]","{}")
```

Resources

Some Resources

- http://www.bluebox.net/about/blog/2013/02/using-regular-expressions-in-ruby-part-1-of-3/
- http://www.ruby-doc.org/core-2.0.0/Regexp.html
- http://rubular.com/
- http://coding.smashingmagazine.com/2009/06/01/essential-guide-to-regular-expressions-tools-tutorials-and-resources/
- http://www.ralfebert.de/archive/ruby/regex_cheat_sheet/
- http://stackoverflow.com/questions/577653/difference-between-a-z-and-in-ruby-regular-expressions (thanks, Austin and Santi)

Topic Exploration

- http://www.codinghorror.com/blog/2005/02/regex-use-vs-regex-abuse.html
- http://programmers.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions
- http://coding.smashingmagazine.com/2009/05/06/introduction-to-advanced-regular-expressions/
- http://stackoverflow.com/questions/5413165/ruby-generating-new-regexps-from-strings

A little more motivation to use…

- http://blog.stevenlevithan.com/archives/10-reasons-to-learn-and-use-regular-expressions
- http://www.websiterepairguy.com/articles/re/12_re.html

No longer required – so explore on your own.