CMSC 330: Organization of Programming Languages

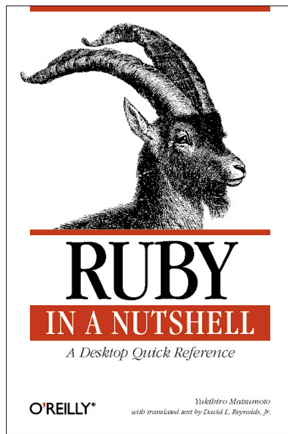
Ruby, Regular Expressions, and Automata

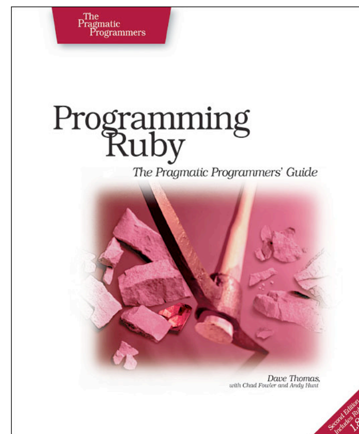
Introduction

- Ruby is an *object-oriented, imperative scripting language*
  - "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language."
  - "I believe people want to express themselves when they program. They don't want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby."

– Yukihiro Matsumoto ("Matz")

Books on Ruby

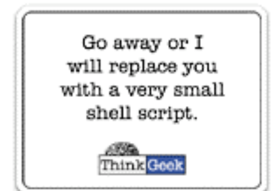




- Earlier version of Thomas book available on web
  - See course web page

Applications of Scripting Languages

- Scripting languages have many uses
  - Automating system administration
  - Automating user tasks
  - Quick-and-dirty development



- Major application:

Text processing

Output from Command-Line Tool

```
% wc *
     271      674     5323 AST.c
     100      392     3219 AST.h
     117     1459   238788 AST.o
    1874     5428    47461 AST_defs.c
    1375     6307    53667 AST_defs.h
     371      884     9483 AST_parent.c
     810     2328    24589 AST_print.c
     640     3070    33530 AST_types.h
     285      846     7081 AST_utils.c
      59      274     2154 AST_utils.h
      50      400    28756 AST_utils.o
     866     2757    25873 Makefile
     270      725     5578 Makefile.am
     866     2743    27320 Makefile.in
      38      175     1154 alloca.c
    2035     4516    47721 aloctypes.c
      86      350     3286 aloctypes.h
     104     1051    66848 aloctypes.o
...
```


Climate Data for IAD in August, 2005

```
================================================================================
1   2   3   4   5  6A  6B    7    8   9  10  11  12  13   14  15   16   17  18
                                         AVG MX 2MIN
DY MAX MIN AVG DEP HDD CDD  WTR  SNW DPTH SPD SPD DIR MIN PSBL S-S WX    SPD DR
================================================================================


 1  87  66  77   1   0  12 0.00  0.0    0 2.5  9 200   M    M  7 18      12 210
 2  92  67  80   4   0  15 0.00  0.0    0 3.5 10  10   M    M  3 18      17 320
 3  93  69  81   5   0  16 0.00  0.0    0 4.1 13 360   M    M  2 18      17 360
 4  95  69  82   6   0  17 0.00  0.0    0 3.6  9 310   M    M  3 18      12 290
 5  94  73  84   8   0  19 0.00  0.0    0 5.9 18  10   M    M  3 18      25 360
 6  89  70  80   4   0  15 0.02  0.0    0 5.3 20 200   M    M  6 138     23 210
 7  89  69  79   3   0  14 0.00  0.0    0 3.6 14 200   M    M  7 1       16 210
 8  86  70  78   3   0  13 0.74  0.0    0 4.4 17 150   M    M 10 18      23 150
 9  76  70  73  -2   0   8 0.19  0.0    0 4.1  9  90   M    M  9 18      13  90
10  87  71  79   4   0  14 0.00  0.0    0 2.3  8 260   M    M  8 1       10 210
...
```


Raw Census 2000 Data for DC

```
u108_S,DC,000,01,0000001,572059,72264,572059,12.6,572059,572059,572059,0,0,
    0,0,572059,175306,343213,2006,14762,383,21728,14661,572059,527044,15861
    7,340061,1560,14605,291,1638,10272,45015,16689,3152,446,157,92,20090,43
    89,572059,268827,3362,3048,3170,3241,3504,3286,3270,3475,3939,3647,3525
    ,3044,2928,2913,2769,2752,2933,2703,4056,5501,5217,4969,13555,24995,242
    16,23726,20721,18802,16523,12318,4345,5810,3423,4690,7105,5739,3260,234
    7,303232,3329,3057,2935,3429,3326,3456,3257,3754,3192,3523,3336,3276,29
    89,2838,2824,2624,2807,2871,4941,6588,5625,5563,17177,27475,24377,22818
    ,21319,20851,19117,15260,5066,6708,4257,6117,10741,9427,6807,6175,57205
    9,536373,370675,115963,55603,60360,57949,129440,122518,3754,3168,22448,
    9967,4638,14110,16160,165698,61049,47694,13355,71578,60875,10703,33071,
    35686,7573,28113,248590,108569,47694,60875,140021,115963,58050,21654,36
    396,57913,10355,4065,6290,47558,25229,22329,24058,13355,10703,70088,657
    37,37112,21742,12267,9475,9723,2573,2314,760,28625,8207,7469,738,19185,
    18172,1013,1233,4351,3610,741,248590,199456,94221,46274,21443,24831,479
    47,8705,3979,4726,39242,25175,14067,105235,82928,22307,49134,21742,1177
    6,211,11565,9966,1650,86,1564,8316,54,8262,27392,25641,1751,248590,1159
    63,4999,22466,26165,24062,16529,12409,7594,1739,132627,11670,32445,2322
    5,21661,16234,12795,10563,4034,248590,115963,48738,28914,19259,10312,47
    48,3992,132627,108569,19284,2713,1209,509,218,125
...
```


A Simple Example

- Let's start with a simple Ruby program

ruby1.rb:

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

```
% ruby -w ruby1.rb
42
%
```

Language Basics

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

comments begin with #, go to end of line

variables need not be declared

line break separates expressions (can also use ";" to be safe)

no special main() function or method

Run Ruby, Run

- There are three ways to run a Ruby program
  - ruby -w *filename* – execute script in *filename*
    - tip: the -w will cause Ruby to print a bit more if something bad happens
  - irb – launch interactive Ruby shell
    - can type in Ruby programs one line at a time, and watch as each line is executed

```
irb(main):001:0> 3+4
=> 7
irb(main):002:0> print("hello\n")
hello
=> nil
```


Run Ruby, Run (cont'd)

- Suppose you want to run a Ruby script as if it were an executable

```
#!/usr/local/bin/ruby -w
print("Hello, world!\n")
```

  - ./filename # run program
  - The first line ("shebang") tells the system where to find the program to interpret this text file
  - Must chmod u+x *filename* first
    - Or chmod a+x *filename* so everyone has exec permission
  - Warning: Not very portable
    - Depends on location /usr/local/bin/ruby

Explicit vs. Implicit Declarations

- Java and C/C++ use *explicit variable declarations*
  - variables are named and typed before they are used
    - int x, y; x = 37; y = x + 5;
- In Ruby, variables are *implicitly declared*
  - first use of a variable declares it and determines type
    - x = 37; y = x + 5;
      - x, y exist, will be integers

Tradeoffs?

Explicit Declarations	Implicit Declarations
Higher overhead	Lower overhead
Helps prevent typos	Easy to mistype variable name
Forces programmer to document types	Figures out types of variables automatically

Methods in Ruby

Methods are declared with def...end

List parameters at definition

May omit parens on call

Invoke method

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)
```

(Methods must begin with lowercase letter and be defined before they are called)

Method (and Function) Terminology

- *Formal parameters* – The parameters used in the body of the method
  - message, n in our example
- *Actual parameters* – The arguments passed in to the method at a call
  - "hello", 3 in our example

More Control Statements in Ruby

- A *control statement* is one that affects which instruction is executed next
  - We've seen two so far in Ruby
    - while and function call
- Ruby also has conditionals

```
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You're not doing so well"
end
```

What is True?

- The *guard* of a conditional is the expression that determines which branch is taken

```
if grade >= 90 then
...
```

Guard

- The *true* branch is taken if the guard evaluates to anything except
  - false
  - nil
- **Warning** to C programmers: 0 is *not* false!

Yet More Control Statements in Ruby

- unless *cond* then *stmt-f* else *stmt-t* end
  - Same as "if not cond then *stmt-t* else *stmt-f* end"

```
unless grade < 90 then
  puts "You got an A"
else unless grade < 80 then
  puts "You got a B"
end
end
```

- until *cond body* end
  - Same as "while not *cond body* end"

```
until i >= n
  puts message
  i = i + 1
end
```


Using If and Unless as Modifiers

- Can write if and unless *after* an expression
  - puts "You got an A" if grade >= 90
  - puts "You got an A" unless grade < 90
- Why so many control statements?
  - Is this a good idea?
  - Advantages? Disadvantages?

Classes and Objects

- Class names begin with an uppercase letter
- The "new" method creates an object
  - s = String.new creates a new String and makes s refer to it
- Every class inherits from Object

Everything is an Object

- In Ruby, *everything* is in fact an object
  - (-4).abs
    - integers are instances of Fixnum
  - 3 + 4
    - infix notation for “invoke the + method of 3 on argument 4”
  - "programming".length
    - strings are instances of String
  - String.new
    - classes are objects with a new method
  - (4.13).class
    - use the class method to get the class for an object
    - floating point numbers are instances of Float

Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- But in Ruby, classes themselves are objects!

Object	Class
10	Fixnum
-3.30	Float
"CMSC 330"	String
String.new	String
Fixnum	Class
String	Class

- Fixnum, Float, String, etc., (including Class), are objects of type Class

Two Cool Things to Do with Classes

- Since classes are objects, you can manipulate them however you like

```
if p then x = String else x = Time end   # Time is
 …                                         # another class
y = x.new          # creates a String or a Time,
                   # depending upon p
```

- You can get names of all the methods of a class
  - Object.methods
    - => ["send", "name", "class_eval", "object_id", "new", "autoload?", "singleton_methods", ...]

The nil Object

- Ruby uses a special object nil
  - All uninitialized fields set to nil (@ refers to a class field)

```
irb(main):004:0> @x
=> nil
```

  - Like NULL or 0 in C/C++ and null in Java
- nil is an object of class NilClass
  - It’s a *singleton object* – there is only one instance of it
    - NilClass does *not* have a new method
  - nil has methods like to_s, but not other methods that don’t make sense

```
irb(main):006:0> @x + 2
NoMethodError: undefined method `+' for nil:NilClass
```

What is a Program?

- In C/C++, a program is...
  - A collection of declarations and definitions
  - With a distinguished function definition
    - int main(int argc, char *argv[]) { ... }
  - When you run a C/C++ program, it's like the OS calls main(...)
- In Java, a program is...
  - A collection of class definitions
  - With a class Cl that contains a method
    - public static void main(String[] args)
  - When you run java Cl, the main method of class Cl is invoked

A Ruby Program is...

- The class Object
  - When the class is loaded, any expressions not in method bodies are executed

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)
```

defines a method of Object (points to `def sayN`)

invokes self.sayN (points to `x = sayN("hello", 3)`)

invokes self.puts (part of Object) (points to `puts(x)`)

Ruby is Dynamically Typed

- Recall we don't declare types of variables
  - But Ruby does keep track of types at run time

    x = 3; x.foo

    NoMethodError: undefined method 'foo' for 3:Fixnum
- We say that Ruby is *dynamically typed*
  - Types are determined and checked at run time
- Compare to C, which is *statically typed*

```
# Ruby
x = 3
x = "foo"   # gives x a
            # new type
```

```
/* C */
int x;
x = 3;
x = "foo"; /* not allowed */
```


Types in Java and C++

- Are Java and C++ statically or dynamically typed?
  - A little of both
  - Many things are checked statically

    Object x = new Object();

    x.println("hello"); // No such method error at compile time
  - But other things are checked dynamically

    Object o = new Object();

    String s = (String) o; // No compiler warning, fails at run time

    // (Some Java compilers may be smart enough to warn about above cast)

Tradeoffs?

Static types	Dynamic types
More work to do when writing code	Less work when writing code
Helps prevent some subtle errors	Can use objects incorrectly and not realize until execution
Fewer programs type check	More programs type check

Classes and Objects in Ruby

```
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end

  def addX(x)
    @x += x
  end

  def to_s
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end

p = Point.new(3, 4)
p.addX(4)
puts(p.to_s)
```


Classes and Objects in Ruby (cont'd)

- Recall classes begin with an uppercase letter
- inspect converts *any* instance to a string
  - irb(main):033:0> p.inspect
  - => "#<Point:0x54574 @y=4, @x=7>"
- Instance variables are prefixed with @
  - Compare to local variables with no prefix
  - *Cannot be accessed outside of class*
- The to_s method can be invoked implicitly
  - Could have written puts(p)
    - Like Java's toString() methods

Inheritance

- Recall that every class inherits from Object

```
class A
  def add(x)
    return x + 1
  end
end

class B < A
  def add(y)
    return (super(y) + 1)
  end
end

b = B.new
puts(b.add(3))
```

Global Variables in Ruby

- Ruby has two kinds of global variables
  - Class variables beginning with @@
  - Global variables across classes beginning with $

```
class Global
  @@x = 0

  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end

  def Global.get
    return @@x
  end
end
```

```
$x = 0
Global.inc
$x = $x + 1
Global.inc
puts(Global.get)
puts($x)
```

define a class ("singleton") method

Special Global Variables

- Ruby has a bunch of global variables that are implicitly set by methods
- The most insidious one: $_
  - Default method return, argument in many cases
- Example:

```
gets     # implicitly reads input into $_
print    # implicitly writes $_
```

- Using $_ leads to shorter programs
  - And confusion
  - It's suggested you avoid using it

Creating Strings in Ruby

- Substitution in double-quoted strings with #{}
  - course = "330"; msg = "Welcome to #{course}"
  - "It is now #{Time.new}"
  - The contents of #{} may be an arbitrary expression
  - Can also use single-quote as delimiter
    - No expression substitution, fewer escaping characters
- Here-documents

```
s = <<END
  This is a long text message
  on multiple lines
  and typing \\n is annoying
END
```


Creating Strings in Ruby (cont'd)

- Ruby also has printf and sprintf
  - printf("Hello, %s\n", name);
  - sprintf("%d: %s", count, Time.now)
    - Returns a string
- The to_s method returns a String representation of a class object

Standard Library: String

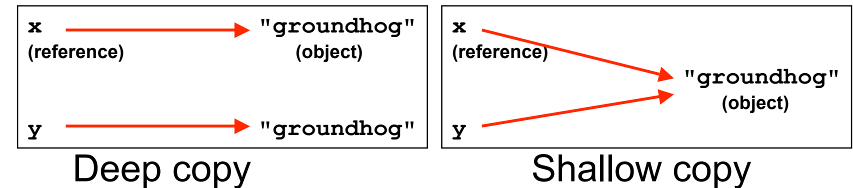
- The String class has many useful methods
  - s.length # length of string
  - s1 == s2 # “deep” equality (string contents)
  - s = "A line\n"; s.chomp # returns "A line"
    - Return new string with s's contents except newline at end of line removed
  - s = "A line\n"; s.chomp!
    - Destructively removes newline from s
    - *Convention:* methods ending in ! modify the object
    - *Another convention:* methods ending in ? observe the object
  - "r1\tr2\t\tr4".each("\t") { |rec| puts rec }
    - Apply code block to each tab-separated substring

Digression: Deep vs. Shallow Copy

- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - (Or even two pointers pointing to the same structure)

```
x = "groundhog" ; y = x
```

- Which of these occurs?



Deep copy Shallow copy

Deep vs. Shallow Copy (cont'd)

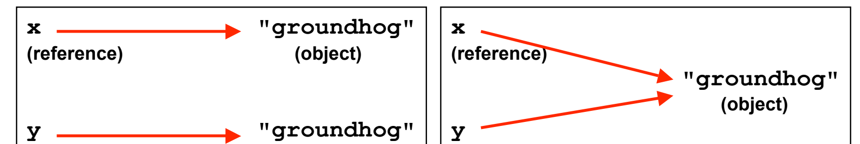
- Ruby and Java would both do a shallow copy in this case
- But this Ruby example would cause deep copy:

```
x = "groundhog"
y = String.new(x)
```

  - Note: In Java, new String(x) is probably not useful; in Ruby, String.new might be. Why?

Deep vs. Shallow Equality

- Consider these cases again:



- If we compare x and y, what is compared?
  - The references, or the contents of the objects they point to?
- If references are compared the first would return false but the second true
- If objects are compared both would return true

String Equality

- In Java, x == y is shallow equality, always
  - Compares references, not string contents
- In Ruby, x == y for strings uses deep equality
  - Compares contents, not references
  - == is a method that can be overridden in Ruby!
  - To check shallow equality, use the equal? method
    - Inherited from the Object class
- It's always important to know whether you're doing a deep or shallow copy
  - And deep or shallow comparison

Standard Library: String (cont'd)

- "hello".index("l", 0)
  - Return index of the first occurrence of string in s, starting at n
- "hello".sub("h", "j")
  - Replace first occurrence of "h" by "j" in string
  - Use gsub ("global" sub) to replace all occurrences
- "r1\tr2\t\tr3".split("\t")
  - Return array of substrings delimited by tab
- Consider these three examples again
  - All involve *searching* in a string for a certain pattern
  - What if we want to find more complicated patterns?
    - Find first occurrence of "a" or "b"
    - Split string at tabs, spaces, and newlines

Regular Expressions

- A way of describing patterns or sets of strings
  - Searching and matching
  - Formally describing strings
    - The symbols (lexemes or tokens) that make up a language
- Common to lots of languages and tools
  - awk, sed, perl, grep, Java, OCaml, C libraries, etc.
- Based on some really elegant theory
  - We'll see that soon

Example Regular Expressions in Ruby

- /Ruby/
  - Matches exactly the string "Ruby"
  - Regular expressions can be delimited by /'s
  - Use \ to escape /'s in regular expressions
- /(Ruby|OCaml|Java)/
  - Matches either "Ruby", "OCaml", or "Java"
- /(Ruby|Regular)/ or /R(uby|egular)/
  - Matches either "Ruby" or "Regular"
  - Use ()'s for grouping; use \ to escape ()'s

Using Regular Expressions

- Regular expressions are instances of Regexp
  - But you won't often use its methods
- Basic matching using =~ method of String

```
line = gets                  # read line from standard input
if line =~ /Ruby/ then       # returns nil if not found
  puts "Found Ruby"
end
```

- Can use regular expressions in index, search, etc.

```
offset = line.index(/(MAX|MIN)/)    # search starting from 0
line.sub(/(Perl|Python)/, "Ruby")   # replace
line.split(/(\t|\n| )/)             # split at tab, space,
                                    # newline
```


Using Regular Expressions (cont'd)

- Invert matching using !~ method of String
  - Matches strings that *don't* contain an instance of the regular expression

Repetition in Regular Expressions

- /(Ruby)*/
  - {"", "Ruby", "RubyRuby", "RubyRubyRuby", ...}
  - * means *zero or more occurrences*
- /Ruby+/
  - {"Ruby", "Rubyy", "Rubyyy", ... }
  - + means *one or more occurrence*
  - so /e+/ is the same as /ee*/
- /(Ruby)?/
  - {"", "Ruby"}
  - ? means *optional*, i.e., zero or one occurrence

Watch Out for Precedence

- /(Ruby)*/ means {"", "Ruby", "RubyRuby", ...}
  - But /Ruby*/ matches {"Rub", "Ruby", "Rubyy", ...}
- In general
  - * and + bind most tightly
  - Then concatenation (adjacency of regular expressions)
  - Then |
- Best to use parentheses to disambiguate

Character Classes

- /[abcd]/
  - {"a", "b", "c", "d"} (Can you write this another way?)
- /[a-zA-Z0-9]/
  - Any upper or lower case letter or digit
- /[^0-9]/
  - Any character except 0-9
- /[\t\n]/
  - Tab, newline or space
- /[a-zA-Z_\$][a-zA-Z_\$0-9]*/
  - Java identifiers ($ escaped...see next slide)

Special Characters

.	any character
^	beginning of line
$	end of line
\$	just a $
\d	digit, [0-9]
\s	whitespace, [\t\r\n\f]
\w	word character, [A-Za-z0-9_]
\D	non-digit, [^0-9]
\S	non-space, [^\t\r\n\f]
\W	non-word, [^A-Za-z0-9_]

Extracting Substrings Based on r.e.'s

- Can be done using the String.scan method
- Or, use *backreferences*
  - Ruby remembers which strings matched the parenthesized parts of r.e.'s
  - These parts can be referred to using special global variables called backreferences (named $1, $2,…)
  - Examples:
    - `/^Status: (.*)/`
      - Capture all chars to the right on lines beginning with "Status"
    - `/^Min: (\d+) Max: (\d+)$/`
      - Capture digits following "Min" and "Max"

Backreference Example

- Extract information from a report

```
gets =~ /^Min: (\d+)  Max: (\d+)$/
min, max = $1, $2
```

- **Warning:** Despite their names, $1 etc are *local* variables

```
def m(s)
  s =~ /(Foo)/
  puts $1    # prints Foo
end
m("Foo")
puts $1      # prints nil
```

The scan Method

- Also extracts substrings based on regular expressions
- Can optionally use parentheses in regular expression to affect how the extraction is done
- Has two forms which differ in what Ruby does with the matched substrings
  - The first form returns an array
  - The second form uses a code block
    - We'll see this later

First Form of the scan Method

- *str*.scan(*regexp*)
  - If *regexp* doesn't contain any parenthesized subparts, returns an array of matches
    - An array of all the substrings of *str* which matched)

```
s = "CMSC 330 Fall 2006"
s.scan(/\d+/)  # returns array ["330", "2006"]
```

  - If *regexp* contains parenthesized subparts, returns an array of arrays
    - Each subarray contains the parts of the string which matched one occurrence of the parenthesized subparts)

```
s = "CMSC 330 Fall 2006"
s.scan(/(\S+) (\S+)/)  # [["CMSC", "330"],
                       #  ["Fall", "2006"]]
```


Standard Library: Array

- Arrays of objects are instances of class Array
  - Arrays may be heterogeneous
    - a = [1, "foo", 2.14]
  - C-like syntax for accessing elements, indexed from 0
    - x = a[0]; a[1] = 37
- Arrays are *growable*
  - Increase in size automatically as you access elements

```
irb(main):001:0> b = []; b[0] = 0; b[5] = 0; puts b.inspect
[0, nil, nil, nil, nil, 0]
```

  - [] is the empty array, same as Array.new

Standard Library: Arrays (cont'd)

- Arrays can also shrink
  - Contents shift left when you delete elements

```
a = [1, 2, 3, 4, 5]
a.delete_at(3)        # delete at position 3; a = [1,2,3,5]
a.delete(2)           # delete element = 2; a = [1,3,5]
```

- Can use arrays to model stacks and queues

```
a = [1, 2, 3]
a.push("a")      # a = [1, 2, 3, "a"]
x = a.pop        # x = "a"
a.unshift("b")   # a = ["b", 1, 2, 3]
y = a.shift      # y = "b"
```

Iterating through Arrays

- It's easy to iterate over an array with while

```
a = [1,2,3,4,5]
i = 0
while i < a.length
  puts a[i]
  i = i + 1
end
```

- Looping through all elements of an array is very common
  – And there's a better way to do it in Ruby

Iteration and Code Blocks

- The Array class also has an each method, which takes a code block as an argument

```
a = [1,2,3,4,5]
a.each { |x| puts x }
```

code block delimited by {}'s or do...end

parameter name

body

More Examples of Code Blocks

- Sum up the elements of an array

```
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

- Print out each segment of the string as divided up by commas
  – Can use any delimiter

```
s = "Student,Sally,099112233,A"
s.each(',') { |x| puts x }
```

("delimiter" = symbol used to denote boundaries)

Yet More Examples of Code Blocks

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

  – n.times runs code block n times
  – n.upto(m) runs code block for integers n..m
  – a.find returns first element x of array such that the block returns true for x
  – a.collect applies block to each element of array and returns new array
- Any method can be called with a code block. Inside the method, the block is called with yield

Still Another Example of Code Blocks

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

- open method takes code block with file argument
  - File automatically closed after block executed
- readlines reads all lines from a file and returns an array of the lines read
  - Use each to iterate

So What are Code Blocks?

- A code block is just a special kind of method
  - { |y| x = y + 1; puts x } is almost the same as
  - def m(y) x = y + 1; puts x end
- The each method takes a code block as an argument
  - This is called *higher-order programming*
    - In other words, methods take other methods as arguments
    - We'll see a lot more of this in OCaml
- We'll see other library classes with each methods
  - And other methods that take code blocks as arguments
  - Your own methods can also take code block args

Second Form of the scan Method

- Remember the scan method?
  - Gave back an array of matches
  - Can also take a code block as an argument
- str.scan(regexp) { |match| block }
  - Applies the code block to each match
  - Short for str.scan(regexp).each { |match| block }
  - The regular expression can also contain parenthesized subparts

Example of Second Form of scan

```
sum_a = sum_b = sum_c = 0
while (line = gets)
  line.scan(/(\d+)\s+(\d+)\s+(\d+)/) { |a,b,c|
    sum_a += a.to_i
    sum_b += b.to_i
    sum_c += c.to_i
  }
end
printf("Total: %d %d %d\n", sum_a, sum_b, sum_c)
```

Sums up three columns of numbers

Standard Library: Hash

- A hash acts like an associative array
  - Elements can be indexed by any kind of values
  - Every Ruby object can be used as a hash key, because the Object class has a hash method

- Elements are referred to using [] like array elements, but Hash.new is the Hash constructor

```
italy["population"] = 58103033
italy["continent"] = "europe"
italy[1861] = "independence"
```


Hash (cont'd)

- The Hash method values returns an array of a hash's values (in some order)
- And keys returns an array of a hash's keys (in some order)
- Iterating over a hash:

```
italy.keys.each {
    |key| puts("key: #{key}, value: #{italy[key]}")
}
```


Hash (cont'd)

Convenient syntax for creating literal hashes

- Use { key => value, ... } to create hash table

```
credits = {
  "cmsc131" => 4,
  "cmsc330" => 3,
}

x = credits["cmsc330"]   # x now 3
credits["cmsc311"] = 3
```


Standard Library: File

- Lots of convenient methods for IO

File.new("file.txt", "rw")	# open for rw access
f.readline	# reads the next line from a file
f.readlines	# returns an array of all file lines
f.eof	# return true if at end of file
f.close	# close file
f << object	# convert object to string and write to f
$stdin, $stdout, $stderr	# global variables for standard UNIX IO

  By default stdin reads from keyboard, and stdout and stderr both write to terminal

- File inherits some of these methods from IO

Exceptions

- Use begin...rescue...ensure...end
  - Like try...catch...finally in Java

```
begin
  f = File.open("test.txt", "r")
  while !f.eof
    line = f.readline
    puts line
  end
  f.close
rescue Exception => e
  puts "Exception:" + e.to_s +
   " (class " + e.class.to_s + ")"
end
```

Class of exception to catch

Local name for exception

The Theory Behind r.e.'s

- That's it for the basics of Ruby
  - If you need other material for your project, you'll either see it in discussion section, or you'll need to learn it on your own
- Next up: How do r.e.'s really work?
  - Mixture of a very practical tool (string matching with r.e.'s) and some nice theory
  - A great computer science result

A Few Questions about Regular Expressions

- What does a regular expression represent?
  - Just a set of strings
- What are the basic components of r.e.'s?
  - E.g., we saw that e+ is the same as ee*
- How are r.e.'s implemented?
  - We'll see how to turn a r.e. into a program
- Can r.e.'s represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the regular languages

Some Definitions

- An *alphabet* is a finite set of symbols
  - Usually denoted Σ
- A *string* is a finite sequence of symbols from Σ
  - ε is the empty string ("" in Ruby)
  - $|s|$ is the length of string s
    - $|Hello| = 5$, $|\varepsilon| = 0$
  - Note: $\varnothing$ is the empty set (with 0 elements); $\varnothing \neq \{ \varepsilon \}$
- *Concatenation* is indicated by juxtaposition
  - If s_1 = super and s_2 = hero, then $s_1 s_2$ = superhero
  - Sometimes also written $s_1 \cdot s_2$
  - For any string s, we have $s\varepsilon = \varepsilon s = s$

Languages

- A *language* is a set of strings over an alphabet
- Example: The set of phone numbers over the alphabet Σ = {0, 1, 2, 3, 4, 5, 6, 7, 9, (,), -}
  - Give an example element of this language
  - Are all strings over the alphabet in the language?
  - Is there a Ruby regular expression for this language?
    - Is the Ruby regular expression over the same alphabet?
- Example: The set of all strings over Σ
  - Often written Σ^*

Languages (cont'd)

- Example: The set of all valid Ruby programs
  - Is there a Ruby regular expression for this language?
- Example: The set of strings of length 0 over the alphabet Σ = {a, b, c}
  - $\{s \mid s \in \Sigma^* \text{ and } |s| = 0\} = \{\varepsilon\} \neq \emptyset$

Operations on Languages

- Let Σ be an alphabet and let L, L_1, L_2 be languages over Σ
- Concatenation L_1L_2 is defined as
  - $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
  - Example: L_1 = {"hi", "bye"}, L_2 = {"1", "2"}
    - L_1L_2 = {"hi1", "hi2", "bye1", "bye2"}
- Union is defined as
  - $L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2\}$
  - Example: L_1 = {"hi", "bye"}, L_2 = {"1", "2"}
    - $L_1 \cup L_2$ = {"hi", "bye", "1", "2"}

Operations on Languages (cont'd)

- Define L^n inductively as
  - $L^0 = \{\varepsilon\}$
  - $L^n = LL^{n-1}$ for $n > 0$
- In other words,
  - $L^1 = LL^0 = L\{\varepsilon\} = L$
  - $L^2 = LL^1 = LL$
  - $L^3 = LL^2 = LLL$
  - ...

Examples of L^n

- Let $L = \{a, b, c\}$
- Then
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{a, b, c\}$
  - $L^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

Operations on Languages (cont'd)

- *Kleene closure* is defined as

$$L^* = \bigcup_{i \in [0..\infty]} L^i$$

- In other words...

  L^* is the language (set of all strings) formed by concatenating together zero or more strings from L

Definition of Regexps

- Given an alphabet Σ, the *regular expressions* over Σ are defined inductively as

regular expression	denotes language
$\varnothing$	$\varnothing$
ε	$\{\varepsilon\}$
each element $\sigma \in \Sigma$	$\{\sigma\}$

  - ...

Definition of Regexps (cont'd)

- Let A and B be regular expressions denoting languages L_A and L_B, respectively

regular expression	denotes language
AB	$L_A L_B$
(A|B)	$L_A \cup L_B$
A*	L_A^*

- There are no other regular expressions for Σ
- We use ()'s as needed for grouping

The Language Denoted by an r.e.

- For a regular expression e, we will write [[e]] to mean the language denoted by e
  - [[a]] = {a}
  - [[(a|b)]] = {a, b}
- If s∊[[re]], we say that re *accepts*, *describes*, or *recognizes* s.

Example 1

- All strings over Σ = {a, b, c} such that all the a's are first, the b's are next, and the c's last
  - Example: aaabbbbccc but not abcb
- Regexp: a*b*c*
  - This is a valid regexp because...
  - a is a regexp ([[a]] = {a})
  - a* is a regexp ([[a*]] = {ε, a, aa, ...})
  - Similarly for b* and c*
  - So a*b*c* is a regular expression

Which Strings Does a*b*c* Recognize?

aabbbcc
: Yes; aa ∊ [[a*]], bbb ∊ [[b*]], and cc ∊ [[c*]], so entire string is in [[a*b*c*]]

abb
: Yes, abb = abbε, and ε ∊ [[c*]]

ac
: Yes

ε
: Yes

aacbc
: No

abcd
: No -- outside the language

Example 2

- All strings over Σ = {a, b, c}
- Regexp: (a|b|c)*
- Other regular expressions for the same language?
  - (c|b|a)*
  - (a*|b*|c*)*
  - (a*b*c*)*
  - ((a|b|c)*|abc)
  - etc.

Example 3

- All whole numbers containing the substring 330
- Regular expression: (0|1|...|9)*330(0|1|...|9)*
- What if we want to get rid of leading 0's?
- ((1|...|9)(0|1|...|9)*330(0|1|...|9)* | 330(0|1|...|9)*)
- Any other solutions?
- What about all whole numbers not containing the substring 330?
  - Is it recognized by a regexp?

Example 4

- What language does (10|0)*(10|1)* denote?
  - (10|0)*
    - 0 may appear anywhere
    - 1 must always be followed by 0
  - (10|1)*
    - 1 may appear anywhere
    - 0 must always be preceded by 1
  - Put together, all strings of 0's and 1's where every pair of adjacent 0's precedes any pair of adjacent 1's

What Strings are in (10|0)*(10|1)* ?

00101000 110111101

First part in [[(10|0)*]]

Second part in [[(10|1)*]]

Notice that 0010 also in [[(10|0)*]]

But remainder of string is not in [[(10|1)*]]

0010101

Yes

101

Yes

011001

No

Example 5

- What language does this regular expression recognize?
  - ((1|ε)(0|1|...|9) | (2(0|1|2|3))) : (0|1|...|5)(0|1|...|9)
- All valid times written in 24-hour format
  - 10:17
  - 23:59
  - 0:45
  - 8:30

Two More Examples

- (000|00|1)*
  - Any string of 0's and 1's with no single 0's
- (00|0000)*
  - Strings with an even number of 0's
  - Notice that some strings can be accepted more than one way
    - 000000 = 00·00·00 = 00·0000 = 0000·00

Regular Languages

- The languages that can be described using regular expressions are the *regular languages* or *regular sets*
- Not all languages are regular
  - Examples (without proof):
    - The set of palindromes over Σ
    - $\{a^n b^n \mid n > 0\}$ (a^n = sequence of n a's)
- Almost all programming languages are not regular
  - But aspects of them sometimes are (e.g., identifiers)
  - Regular expressions are commonly used in parsing tools

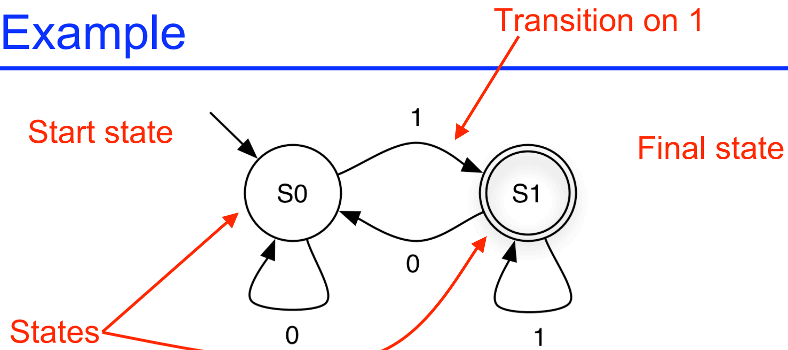
Ruby Regular Expressions

- Almost all of the features we've seen for Ruby r.e.'s can be reduced to this formal definition
  - /Ruby/ – concatenation of single-character r.e.'s
  - /(Ruby|Regular)/ – union
  - /(Ruby)*/ – Kleene closure
  - /(Ruby)+/ – same as (Ruby)(Ruby)*
  - /(Ruby)?/ – same as (ε|(Ruby)) (// is ε)
  - /[a-z]/ – same as (a|b|c|...|z)
  - / [^0-9]/ – same as (a|b|c|...) for a,b,c,... ∈ Σ - {0..9}
  - ^, $ – correspond to extra characters in alphabet

Implementing Regular Expressions

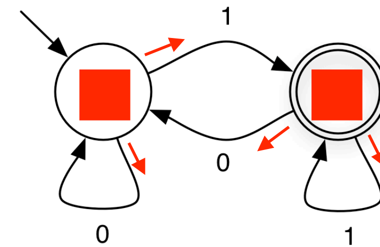
- We can implement regular expressions by turning them into a *finite automaton*
  - A "machine" for recognizing a regular language

Example



- Machine starts in *start* or *initial* state
- Repeat until the end of the string is reached:
  - Scan the next symbol s of the string
  - Take transition edge labeled with s
- The string is *accepted* if the automaton is in a *final* or *accepting* state when the end of the string is reached

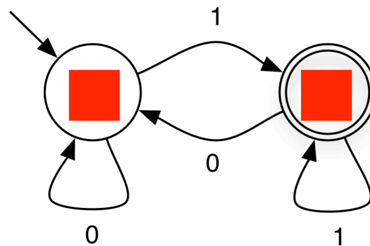
Example



0 0 1 0 1 1

accepted

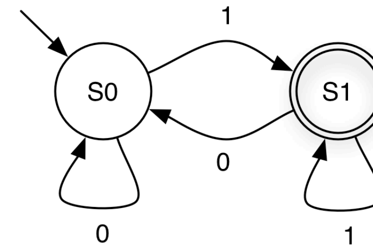
Example



0 0 1 0 1 0

not accepted

What Language is This?



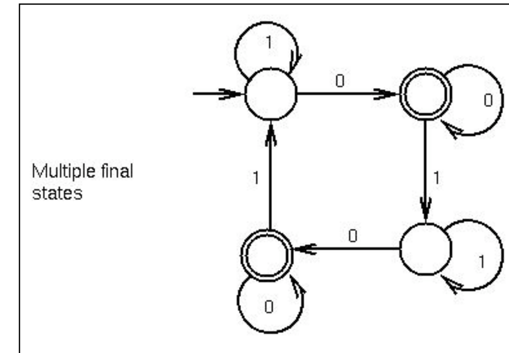
- All strings over {0, 1} that end in 1
- What is a regular expression for this language?
  (0|1)*1

Formal Definition

- A *deterministic finite automaton (DFA)* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - Σ is an alphabet
    - the strings recognized by the DFA are over this set
  - Q is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
    - How many can there be?
  - $\delta : Q \times \Sigma \rightarrow Q$ specifies the DFA's transitions
    - What's this definition saying that δ is?

More on DFAs

- An FSA can have more than one final state:

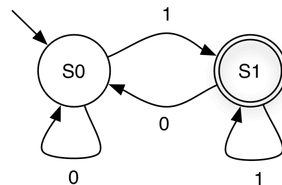


- A string is accepted as long as there is at least one path to a final state

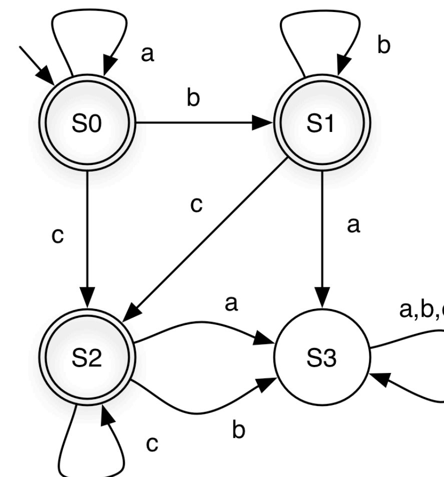
Our Example, Formally

- $\Sigma = \{0, 1\}$
- $Q = \{S0, S1\}$
- $q_0 = S0$
- $F = \{S1\}$
-

δ	0	1
S0	S0	S1
S1	S0	S1



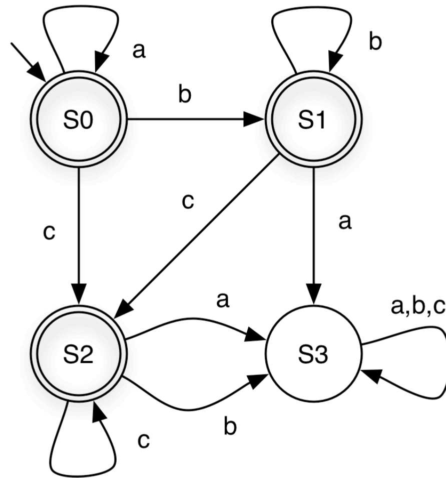
Another Example



string	state at end	accepts ?
aabcc	S2	Y
acc	S2	Y
bbc	S2	Y
aabbb	S1	Y
aa	S0	Y
ε	S0	Y
acba	S3	N

(a,b,c notation shorthand for three self loops)

Another Example (cont'd)

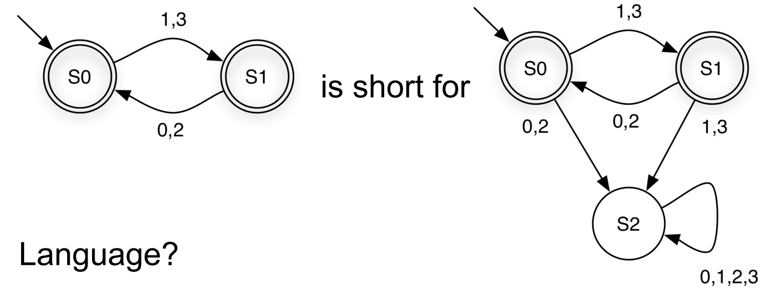


What language does this DFA accept? a*b*c*

S3 is a *dead state* – a nonfinal state with no transition to another state

Shorthand Notation

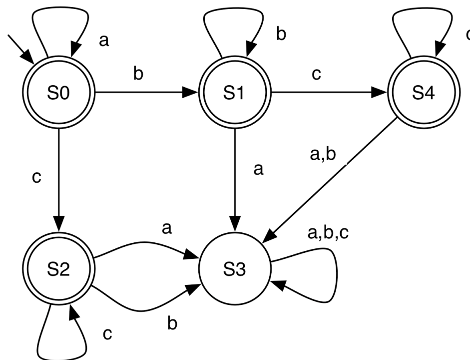
- If a transition is omitted, assume it goes to a dead state that is not shown



Language?

Strings over {0,1,2,3} with alternating even and odd digits, beginning with odd digit

What Lang. Does This DFA Accept?

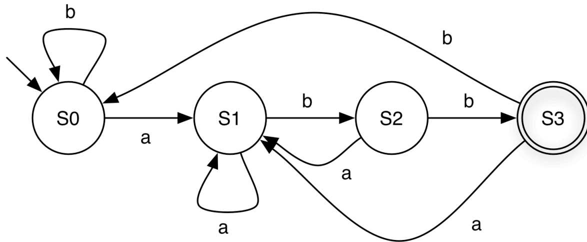


a*b*c* again, so DFAs are not unique

Nondeterministic Finite Automata (NFA)

- An *NFA* is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where
  - Σ is an alphabet
  - Q is a nonempty set of states
  - $q_0 \in Q$ is the start state
  - $F \subseteq Q$ is the set of final states
    - There may be 0, 1, or many
  - $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ specifies the NFA's transitions
    - Transitions on ε are allowed – can optionally take these transitions without consuming any input
    - Can have more than one transition for a given state and symbol
- An NFA accepts s if there is *at least one* path from its start to final state on s

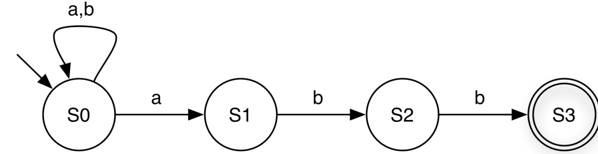
Example DFA



- S0 = "Haven't seen anything yet"
- S1 = "Last symbol seen was an a"
- S2 = "Last two symbols seen were ab"
- S3 = "Last three symbols seen were abb"

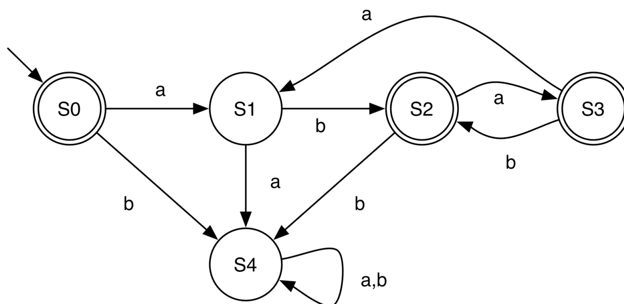
- Language?
- (a|b)*abb

NFA for (a|b)*abb



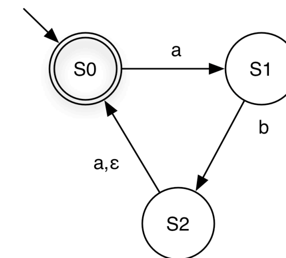
- ba
  - Has paths to either S0 or S1
  - Neither is final, so rejected
- babaabb
  - Has paths to different states
  - One leads to S3, so accepted

Another example DFA



- Language?
- (ab|aba)*

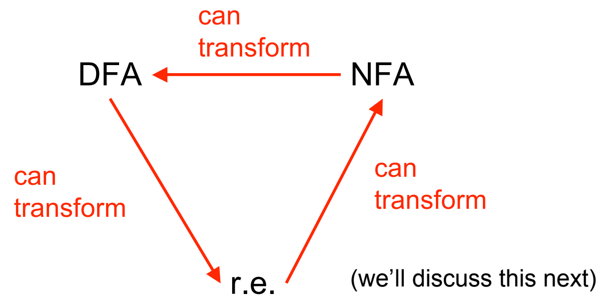
NFA for (ab|aba)*



- aba
  - Has paths to states S0, S1
- ababa
  - Has paths to S0, S1
  - Need to use ε-transition

Relating R.E.'s to DFAs and NFAs

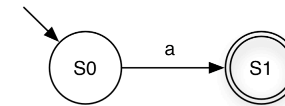
- Regular expressions, NFAs, and DFAs accept the same languages!



(we'll discuss this next)

Reducing Regular Expressions to NFAs

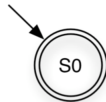
- Goal: Given regular expression e, construct NFA <e> = $(\Sigma, Q, q_0, F, \delta)$
  - Remember r.e. defined recursively from primitive r.e. languages
  - Invariant: |F| = 1 in our NFAs
- Base case: a



<a> = ({a}, {S0, S1}, S0, {S1}, {(S0, a, S1)})

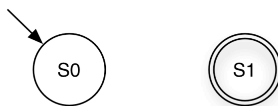
Reduction (cont'd)

- Base case: ε



<ε> = (ε, {S0}, S0, {S0}, ∅)

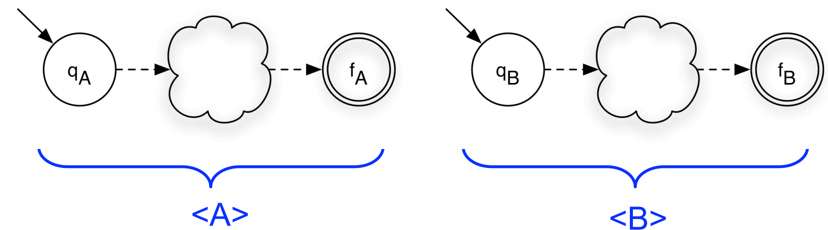
- Base case: ∅



<∅> = (∅, {S0, S1}, S0, {S1}, ∅)

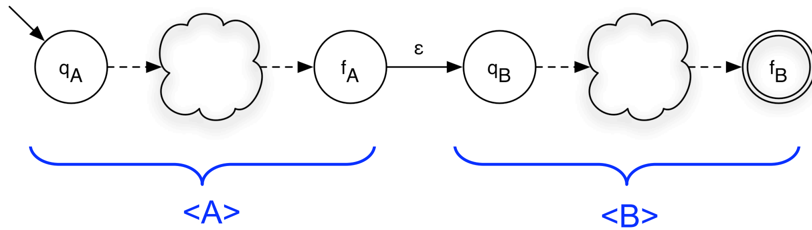
Reduction (cont'd)

- Induction: AB



Reduction (cont'd)

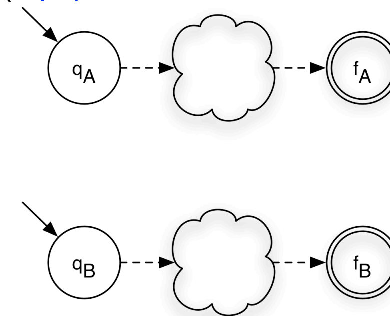
- Induction: AB



- <A> = $(\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- <B> = $(\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
- <AB> = $(\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \{f_B\}, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_B)\})$

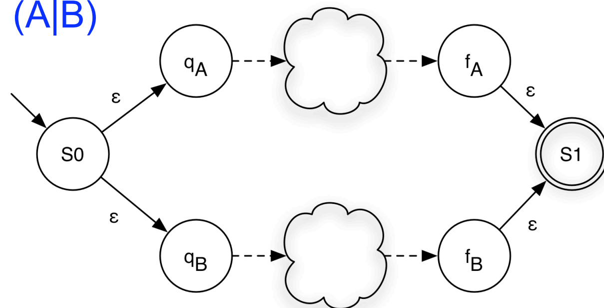
Reduction (cont'd)

- Induction: (A|B)



Reduction (cont'd)

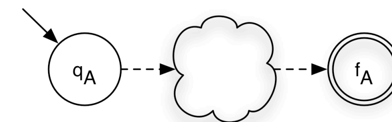
- Induction: (A|B)



- <A> = $(\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- <B> = $(\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
- <(A|B)> = $(\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0,S1\}, S0, \{S1\},$
  $\delta_A \cup \delta_B \cup \{(S0,\varepsilon,q_A), (S0,\varepsilon,q_B), (f_A,\varepsilon,S1), (f_B,\varepsilon,S1)\})$

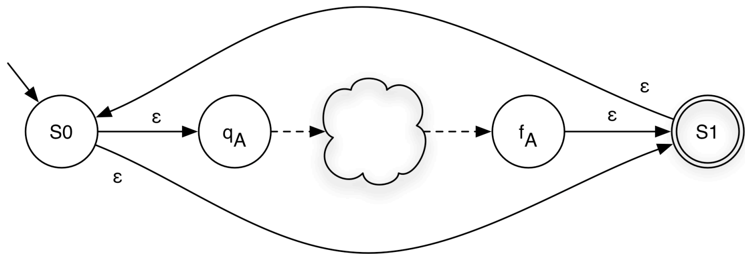
Reduction (cont'd)

- Induction: A*



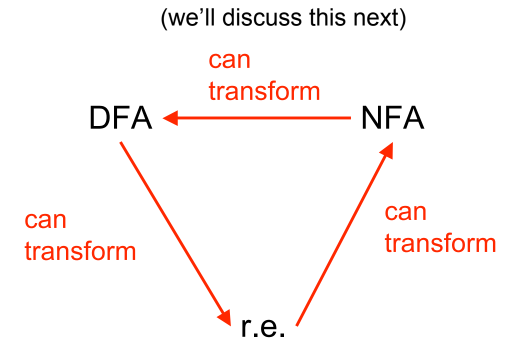
Reduction (cont'd)

- Induction: A*



- $\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $\langle A^* \rangle = (\Sigma_A, Q_A \cup \{S0,S1\}, S0, \{S1\},$
  $\delta_A \cup \{(f_A,\varepsilon,S1), (S0,\varepsilon,q_A), (S0,\varepsilon,S1), (S1,\varepsilon,S0)\})$

Relating R.E.'s to DFAs and NFAs

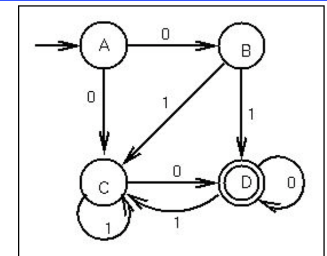


Reduction Complexity

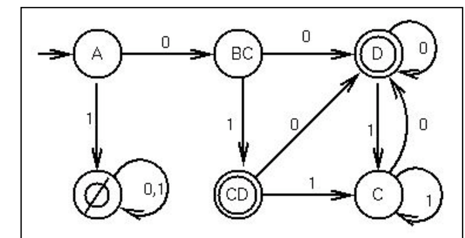
- Given a regular expression A of size n...
  Size = # of symbols + # of operations
- How many states does <A> have?
  - O(n)
  - That's pretty good!
- NFA to DFA reduction
  - Intuition: Build DFA where each DFA state represents a set of NFA states
  - Given NFA with n states, DFA may have 2^n states
  - Not so good, since DFAs are what we can implement easily

Equivalence of DFAs and NFAs

- Let subsets of states be states in DFA
- Keep track of which subset you can be in

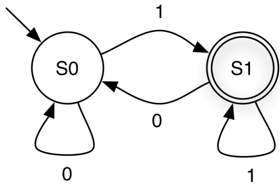


- Any string from {A} to either {D} or {CD} represents a path from A to D in the original NFA.



Implementing DFAs

It's easy to build a program which mimics a DFA




```
cur_state = 0;
while (1) {

  symbol = getchar();

  switch (cur_state) {

    case 0: switch (symbol) {
              case '0':  cur_state = 0; break;
              case '1':  cur_state = 1; break;
              case '\n': printf("rejected\n"); return 0;
              default:   printf("rejected\n"); return 0;
            }
            break;

    case 1: switch (symbol) {
              case '0':  cur_state = 0; break;
              case '1':  cur_state = 1; break;
              case '\n': printf("accepted\n"); return 1;
              default:   printf("rejected\n"); return 0;
            }
            break;

    default: printf("unknown state; I'm confused\n");
             break;
  }
}
```


Implementing DFAs (Alternative)

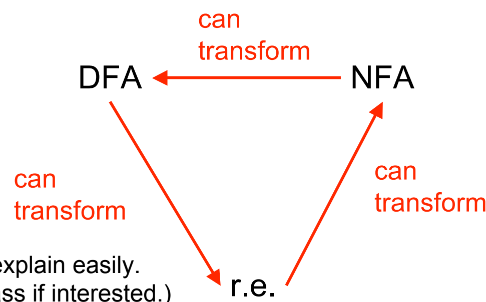
Alternatively, use generic table-driven DFA

given components (Σ, Q, q_0, F, δ) of a DFA:
let q = q_0
while (there exists another symbol s of the input string)
  q := δ(q, s);
if qϵF then
  accept
else reject

– q is just an integer
– Represent δ using arrays or hash tables
– Represent F as a set

Relating R.E.'s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!



(Difficult to explain easily. Ask after class if interested.)

Run Time of Algorithm

- Given a string s, how long does algorithm take to decide whether s is accepted?
  – Assume we can compute δ(q0, c) in constant time
  – Then the time per string s to determine acceptance is O(|s|)
  – Can't get much faster!
- But recall that constructing the DFA from the regular expression A may take $O(2^{|A|})$ time
  – But this is usually not the case in practice
- So there's the initial overhead, but then accepting strings is fast

Regular Expressions in Practice

- Regular expressions are typically "compiled" into tables for the generic algorithm
  - Can think of this as a simple byte code interpreter
  - But really just a representation of $(\Sigma, Q_A, q_A, \{f_A\}, \delta_A)$, the components of the DFA produced from the r.e.
- Regular expression implementations often have extra constructs that are non-regular
  - I.e., can accept more than the regular languages
  - Can be useful in certain cases
  - Disadvantages: nonstandard, plus can have higher complexity

Considering Ruby Again

- Interpreted
- Implicit declarations
- Dynamically typed
  - These three make it quick to write small programs
- Built-in regular expressions and easy string manipulation
  - This and the three above are the hallmark of scripting languages
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

Other Scripting Languages

- Perl and Python are also popular scripting languages
  - Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  - Both include optional "compilation" for speed of loading/execution
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever one you like best