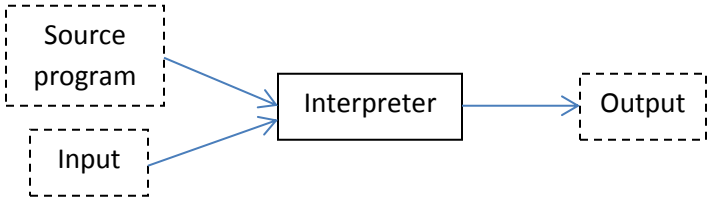
CMSC330 Fall 2013: Midterm 1 Review
Ruby, Regular Expressions, Finite Automata, Basic OCaml
Ilse Haim

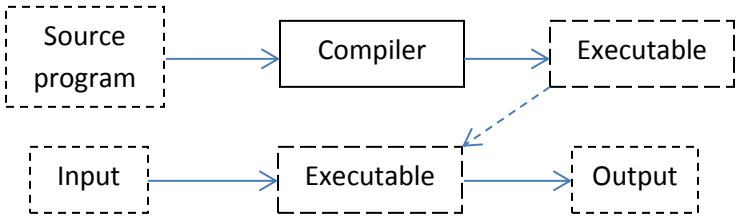
Ruby

General Features:

- **Object-Oriented**: Everything is an object!
- **Imperative**: Procedures and statements are building blocks (as opposed to functions, like in OCaml)
- **Scripting Language**: Good for text processing (built-in regular expressions)
- **Interpreted**: Source code and input are directly executed to create output
  - vs. compiled—no separate executable in Ruby
  - Interpretation:



  - Compilation:



- **Implicit Declarations**: First use of a variable declares it and determines type
  - Ruby: x = 7; y = x – 2;
  - vs. Explicit (Java, C, C++): int x, y; x = 37; y = x + 5; (must first declare a variable and its type)
- **Dynamically Typed**: Types are determined and checked at run time
  - x = 3; x = "foo" -- give x a new type
  - vs. Statically Typed (C): above code would not compile

Writing Classes

- variable types/scopes:

declaration	variable type
x	local
@x	instance (each instance of class gets its own copy)
@@x	static (shared among all instances of class)
$x	global (shared across classes)

- *initialize* is the constructor
  - ex:

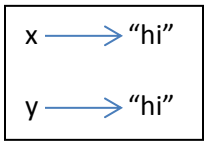
```
class Point
    def initialize(x, y)
      @x = x
      @y = y
    end
end
```

- getter:
```
def x
  @x
end
```
setter:
```
def x= (value)
  @x = value
end
```
- shortcut: *attr_accessor* automatically defines both getters and setters for specified instance variables
  - ex: attr_accessor "x", "y"

Important Methods and Classes

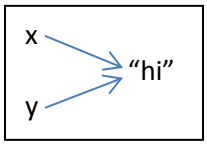
- = = vs. .equal?
  - Opposite of Java
  - = = is *structural equality*: compares contents
  - .equal? is *physical equality*: compares references (memory addresses)
  - ex:

x = “hi”
y = “hi”



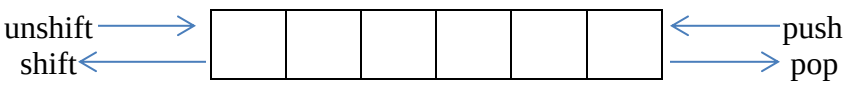
x == y : true
x.equal?(y) : false

x = “hi”
y = x



x == y : true
x.equal?(y) : true

- new : creates a new object by calling the constructor (ex: arr = Array.new)
- puts vs. print : puts automatically adds newline character \n
- to_s : like toString() in Java
- Style
  - Methods that return a boolean should end in ?
  - Methods that change the state in place should end in !
- What is false in Ruby?
  - Only false and nil
  - Everything else is true -- true, 0 (unlike in C), “hi”, -34.3, [1,2], etc.
- String class
  - Useful methods: length, chomp, each, index, include?
  - Embed other expressions with #{ }
    - ex: y = [1,2]
      print “my Array is: #{y}” → prints: my Array is [1,2]
    - ex: print “2 + 3 is #{2+3}” → prints: 2 + 3 is 5
- Array class
  - May be heterogeneous: arr = [1, “foo”, 3.14]
  - Growable: arr = [“Hi”]; arr[3] = “Ilse” → arr = [“Hi”, nil, nil, “Ilse”]
  - Useful methods: length, each, sort!, uniq!, reverse!, delete, delete_at, include?
  - push/pop vs. unshift/shift:
    - push adds to the end of the Array and pop removes from the end
    - unshift adds to the front of the Array and shift removes from the front



  - example of .each: arr = [1,2,3]
    arr.each {|x| print x} → prints: 123

- Hash class
  - Stores {key => value} pairs
  - May have heterogeneous pair types: {1 => "hello", [2,5] => 3.4}
  - hash_name[key] returns the value associated to that key, or nil if key not present
  - Useful methods: keys, values, empty?, has_key?, has_value?, each{|k,v| …}, each_key{|k| …}, each_value{|v| …}
- Range class
  - (m..n) represents numbers between m and n, both inclusive
    - ex: (1..4) is 1,2,3,4
  - (m…n) represents numbers between m (inclusive) and n (exclusive)
    - ex: (1…4) is 1,2,3
  - Not just for integers
    - 'a'..'z' is range of letters 'a' to 'z'
    - 1.5…2.7 is continuous range [1.5, 2.7)
    - Can't iterate over continuous ranges, but can check if [1.5, 2.7).include?(num)

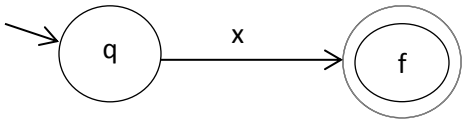
Regular Expressions (class Regexp):

- Matching with "String" =~ /Regexp/ (or not matching: "String" !~ /Regexp/)
- Combinations:
  - R_1R_2 : Match Regexps R_1 and R_2 concatenated together (need to match both in correct order)
  - $R_1 | R_2$: Match R_1 or R_2
  - R* : Match *0 or more* occurrences of R (accepts empty string ε)
  - R+ : Match *1 or more* occurrences of R
  - R? : Match *0 or 1* occurrences of R (accepts empty string ε)
  - R{m,n} : Match *m to n* occurrences of R (both m and n included: m, m+1, m+2, …, or n times)
  - R{m,} : Match *m or more* occurrences of R
  - R{m} : Match *exactly m* occurrences of R
- Precedence
  - *, +, ?, {m,n}, {m,}, {m} bind most tightly
  - Then concatenation
  - Then |
  - Add parentheses to avoid confusion
  - ex: /ab*|c/ same as /(a(b*))|c/
- Special characters
  - ^ beginning of line
  - $ end of line
  - . any character
  - \d digit, \s whitespace, \w word character, \D non-digit, \S non-whitespace, \W non-word char
- Character Classes
  - Shortcut to group together accepted characters
  - ex: [abc] same as (a|b|c)
  - ex: [a-zA-Z0-9] is any lower or upper case letter or any digit
  - ^ means "not" and applies to everything in the character class
    - ex: [^ab] means any character except a or b
- Back References
  - Can refer back to parts of a regular expression that are in parentheses
  - Assigned variable names $1, $2, …
    - These are local variables, and are reset to nil after next search
  - ex: if ("min: 3 max: 76" =~ /min: (\d+) max: (\d+)/)
    print ($1 + " " + $2)
    end → prints: 3 76

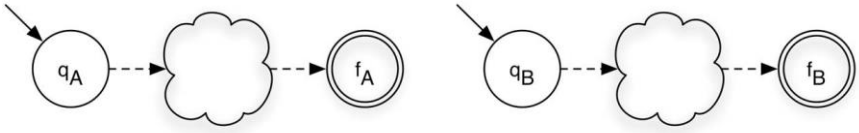
- Scan: string.scan(regexp)
    - If regexp doesn't contain parenthesized subparts, then returns Array of matches
        - ex: s = "CMSC 330 Fall 2013"
          arr = s.scan(/\d+/) → arr = ["330", "2013"]
    - If regexp does contain parenthesized subparts, then returns Array of Arrays
        - Each sub-Array contains the parts of the string which matched one occurrence of search
        - Each sub-array has same number of entries as the number of parenthesized subparts
        - ex: s = "CMSC 330 Fall 2013"
          arr = s.scan(/(\S+) (\S+)/) → arr = [["CMSC", "330"], ["Fall", "2013"]]

Finite Automata

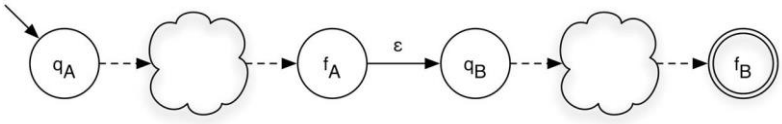
- NFA: Nondeterministic Finite Automata
- DFA: Deterministic Finite Automata
- All have one start state
- May have multiple final states (states inside double circles)
- A string is accepted if automaton is in final state when end of string reached (for some path if NFA)
- Differences
    - NFAs
        - May have ε-transitions (transitions on the empty string epsilon)
        - May have more than one transition leaving a state on the same symbol
    - DFAs
        - No ε-transitions
        - Allow only one transition per symbol coming out of each state
- Algorithms (see examples in the slides):
    - Regexp → NFA
        - Recursively build
        - Base case: /x/ for any single character x is drawn as:



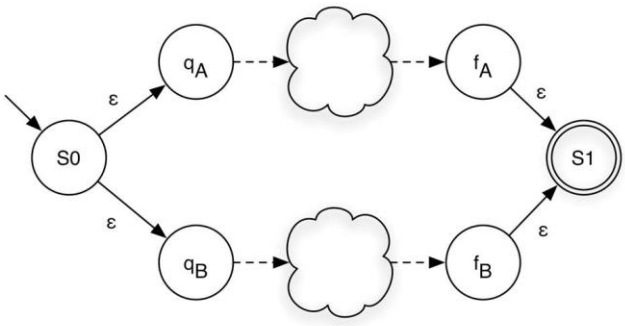
        - Given RE_1 (left) and RE_2 (right), where clouds represent anything between start and final:



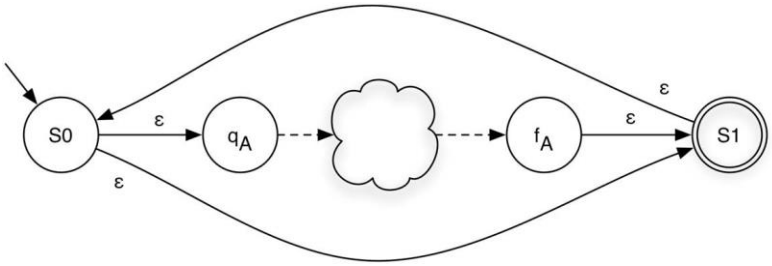
            - Concat RE_1 RE_2 :



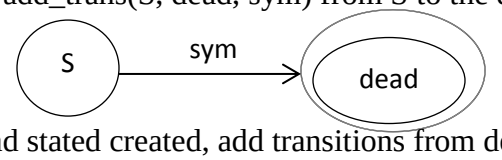
            - Union RE_1 | RE_2 :



- Kleene Closure RE_1* :



- NFA → DFA
  - Subset Construction:
  - (Note: ε-closure(S) = {S, all states reachable from S using only ε-transitions})
  1. The start state of the DFA is the ε-closure of the start state of the NFA.
  2. Perform the following for the new DFA state:
     For each symbol in the alphabet:
     - Apply move to the state and the input symbol; this will return a set of states.
     - Apply the ε-closure to this set of states, possibly resulting in a new set.
     - This set of NFA states will be a single state in the DFA.
  3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 to existing DFA states does not yield any new states.
  4. The final states of the DFA are those which contain any of the final states of the NFA.
- DFA Complement
  1. Swap which states are final and non-final
  2. for each state S {
     for each symbol sym in alphabet {
     if there is no transition from state S on sym
     create dead state if not yet created (and make it final)
     add_trans(S, dead, sym) from S to the dead state as follows:



  3. if dead stated created, add transitions from dead state back to itself for all symbols
- DFA Minimization
  - Initial partition: P1 = {final states}, P2 = {non-final states}
  - Recursively split each partition P until either
    1) P has only 1 state, or
    2) All states in P "agree" on transitions on each symbol
  - Example of what is meant by "agree":
    Given P1 = {S, T}, P2 = {R}
    - All states in P1 "agree" below:
      move(S,a) = R which is in P2
      move(T,a) = R which is in P2
      move(S,b) = S which is in P1
      move(T,b) = T which is in P1
    - States in P1 do not "agree" below:
      move(S,a) = R which is in P2
      move(T,a) = R which is in P2
      move(S,b) = S which is in P1
      move(T,b) = R which is in P2

      } Same letter leads to different partitions… need to split

OCaml

General Features:

- **Mostly Functional**: Functions are building blocks; no or few writes to memory
- **Compiled**: Input runs on separate executable file
- **Type Inference**: No need to write types in the source code
- **Statically Typed**: Type conflicts are checked at compile time
- **Pattern Matching**: Used to deconstruct data structures
- **Parametric Polymorphism**: Function parameters can be polymorphic—similar to Java Generics
- **Higher-order functions**: Functions can be parameters and return values of functions

"Let"

- Variable ex: let x = 5;;
  - Binds x to 5 from now on
- Variable ex: let x = 5 in expression
  - x is only bound to 5 within expression; not in scope after expression
- Function ex: let next x = x + 1;;
  - x is a parameter to the function "next", the return value is x + 1
  - Type of x is inferred to be int (because + operator is only defined for int)
- Need to use "let rec" to define a recursive function
  - ex: let rec factorial n =

```
if n = 0 then
  1
else
  n * factorial (n-1);;
```

Lists:

- Homogeneous: all elements must be of the same type
  - [1; 2; 3] is a valid list, [1; "hi"] is not
- May build up using :: operator
  - element::list adds element to the front of list
  - ex: 1::[] evaluates to [1]
  - ex: 1::(3::(4::[])) evaluates to [1;3;4]
- Type inference: append the word "list" after the type of its elements
  - ex: the elements of x = [1; 2] are ints, so x has type: int list
  - ex: [[1; 2]; [4; 2]] has type: int list list
  - ex: [[1; 2]; ["a"; "b"]] is not a valid list—not homogeneous

Tuples:

- Heterogeneous: may have elements of different types
  - (1,2,3) is a valid tuple and so is (1,"hi")
  - Have a fixed size (there is no :: operator to increase the size of a tuple)
- Type inference: concatenate the types of its elements using *
  - ex: (1,2) has type: int * int
  - ex: ("CMSC",330) has type: string * int
  - ex: ('c', 3, ["hi", "bye"], 4.5) has type: char * int * string list * float
  - ex: [(2,3); (1,7)] has type (int * int) list

Function Type Inference

- Use –> to specify type
  - let f param = ret has type: param_type –> ret_type
  - ex: let next x = x + 1 has type: int –> int
- Every function takes exactly one parameter as input and returns exactly one value as its result
  - May fake having multiple parameters or return values with Tuples
    - ex: let add(x, y) = x + y looks like takes two arguments, but really only taking one tuple
    - type is: int * int –> int
- May have polymorphic types
  - No type-specific operators on parameters are used within the function
  - Polymorphic parameters and return values have type ‘a, ‘b, ‘c, …
  - ex: let hd (h::_) = h;; returns the head of a list, and will work for any type of list
    - hd has type: 'a list –> 'a
  - ex: let swap (x, y) = (y, x);; has type: 'a * 'b –> 'b * 'a

Example: Recursive List Function (see more in slides/code on webpage):

- let rec length l =
  match l with
  [ ] –> 0
  | (_::t) –> 1 + length t;; (* note: underscore _ means we don’t care enough to give the head a name *)
- Type: ‘a list –> int (can take any type of list but always returns an int)

Good Luck on the Exam!