Regular Expressions & Script Programming

15-123

Systems Skills in C and Unix

Topics

- Formal Languages
- Finite State Machines
- Regular Expressions
- RegEx Grammer
  - Alternation
  - Grouping
  - Quantification
- Pattern search utilities in unix
  - grep, awk
- Perl Primer
  - examples

Formal Languages

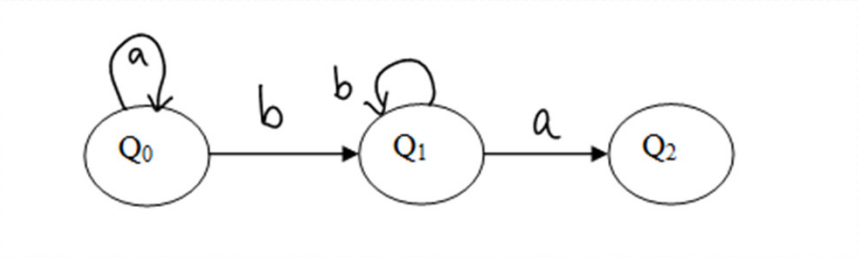
- Formal language consists of
  - An alphabet
  - Formal grammar
- Formal grammar defines
  - Strings that belong to language
- Formal languages with **formal semantics** generates rules for semantic specifications of programming languages

Automaton

- An **automaton** (or **automata** in plural) is a machine that can recognize valid strings generated by a **formal language**.
- A **finite automata** is a mathematical model of a **finite state machine** (FSM), an abstract model under which all modern computers are built.

Automaton

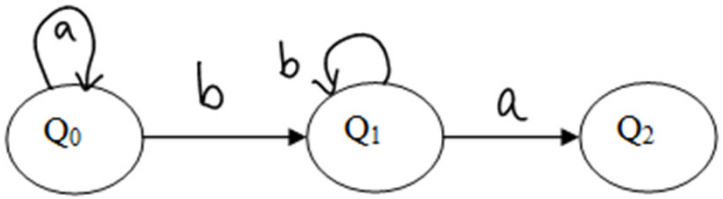
- A FSM is a machine that consists of a set of finite states and a transition table.



- The FSM can be in any one of the states and can transit from one state to another based on a series of rules given by a transition function.

Example

What does this machine represents? Describe the kind of strings it will accept.



Exercise

- Draw a FSM that accepts any string with even number of A's. Assume the alphabet is {A,B}

Build a FSM

- Stream: “Ilovecatsandmorecatsandbigcats ”
- Pattern: “cat”

Regular Expressions

Case for regular expressions

- Many web applications require pattern matching
  - look for <a href> tag for links
  - Token search
- A regular expression
  - A pattern that defines a *class of strings*
  - Special syntax used to represent the class
    - Eg; *.c - any pattern that ends with .c

Regex versus FSM

- A regular expressions and FSM's are equivalent concepts.
- *Regular expression is a pattern that can be recognized by a FSM.*
- *Regex is an example of how good theory leads to good programs*

Regular Expression

- regex defines a class of patterns
  - Patterns that ends with a “*”
- Regex utilities in unix
  - grep, awk, sed
- Applications
  - Pattern matching (DNA)

```
ttaatgacctttttttttttccatgccctcgaataggcttgagcttgccaattaacgcgcacg
ggctggccgggcgtataagccaaggtgtagtgaggttgcattatacatgccggcttgtgatta
acgcatgccataggacggttaggctcagaacccgcaaccaatacacgtgattttctcgtcccc
tg
```

Regex Engine

- A software that can process a string to find regex matches.
- Regex software are part of a larger piece of software
  - grep, awk, sed, php, python, perl, java etc..
- We can write our own regex engine that recognizes all "*caa*" in a strings
  - See democode folder
- Different regex engines may not be compatible with each other
  - Perl 5 is a popular one to learn

Regex machines

- Perl can do a “decent” job with simple regex’s
- But it can fail in cases where expressions can be of the form _____
- One of the best regex machines was written in C by Ken Thompson in the 70’s
  - 400 lines of C code
  - Superior to perl, python and other implementations when working with real world applications

Unix grep utility

The grep command

grep
NAME
grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS
grep [options] PATTERN [FILE...]
grep [options] [-e PATTERN | -f FILE] [FILE...]

DESCRIPTION
grep searches the named input FILEs (or standard input if no files are named, or the file name - is given) for lines containing a match to the given PATTERN. By default, grep prints the matching lines.

Source: unix manual

Simple grep examples

- grep “<a href” guna.html > output.txt
- ls | grep “guna”
- grep ‘regex’ filename
- man grep
  - For more info

regex grammer

Regular Expression Grammar

- Regex grammar defines a set of rules for finding patterns. Grammar categories
  - Alternation
  - Grouping
  - quantification

Regular Expression Grammar

- **Alternation**
- The vertical bar is used to describe alternating choices among two or more choices.
  - the notation **a | b | c** indicates that we can choose a or b or c as part of the string.
  - Another example is that "**(c|s)at**" describes the expressions "cat" or "sat". **n**

Regular Expression Grammar

Grouping

Parenthesis can be used to describe the scope and precedence of operators.

In the example above **(c|s)** indicates that we can either begin with c or s but must immediately follow by “at”

Regular Expression Grammar

- **Quantification**
  - Quantification is the notation used to define the number of symbols that could appear in the string.
- The most common quantifiers are
  - **?, * and +**
  - The **? mark indicates that there is zero or one** of the previous expression.
  - The "*" **indicates that zero or more** of the previous expression can be accepted.
  - The "+" **indicates that one or more** of the previous expression can be accepted.

Examples of *, ? , +

Other facts

- . matches a single character
- .* matches any string
- [a-zA-Z]* matches any string of alphabetic characters
- [ag].* matches any string that starts with a or g
- [a-d].* matches any string that starts with a,b,c or d
- ^(ab) matches any string that begins with ab. In general, to match all lines that begins with any string use ^string
- (ab)$ matches any string that ends with ab

Finding non-matches

- To exclude a pattern
  - [^class]
  - Eg: [^0-9]

Group Matches

- grep '<h\([1-4]\)>.*h\([1-3]\)>' filename
  - What patterns match?
- grep 'h\([1-4]\).*h\1' filename
  - Back-reference

Character Classes

- \d digit [0-9]
- \D non-digit [^0-9]
- \w word character [0-9a-z_A-Z]
- \W non-word character [^0-9a-z_A-Z]
- \s a whitespace character [\t\n\r\f]
- \S a non-whitespace character [^ \t\n\r\f]

More regex notation

- *{n,m} at least n but not more than m times*
- *{n,} – match at least n times*
- {n} – match exactly n times

More examples of regex

- Find all files that begins with “guna”
- Find all files that does not begins with “guna”
- Find all files that ends with guna
- Find all directories in current folder. Write them to an external file.

Exercise

- An email address must begin with an alpha character and can have any combination of alpha characters and characters from {0..9, %, _, +, -} followed by @ and a domain name {alpha-numeric} followed by {.} and any token from the set {edu, com, us, org, net}. Write a regex to describe this.

Summarized Facts about regex

- Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated sub expressions.
- Two regular expressions may be joined by the infix operator | **the resulting** regular expression matches any string matching either sub expression

Summarized Facts about regex

- Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole sub expression may be enclosed in parentheses to override these precedence rules
- The backreference \n, where n is a single digit, matches the substring previously matched by the nth parenthesized sub expression of the regular expression.
- In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \).

Text Processing Languages

- awk
  - Text processing language
  - awk '/pattern/' somefile
  - awk '{if ($3 < 1980) print $3, " ",$5,$6,$7,$8}' somefile
- sed
  - A stream editor
  - sed s/moon/sun/ < moon.txt >sun.txt
- Perl
  - A powerful scripting language
  - We will discuss this next
- We will discuss this *very* briefly for the fun of it. Sed and Awk will not be tested. We will extensively study perl though

Basics of sed

sed basics

- sed is a stream editor
- > sed ‘s/guna/foo/’ filename
  - Replaces guna by foo in the file
    - first occurrence on each line
  - output sent to stdout
- > sed ‘s/guna/foo/g’ filename
  - Globally replaces guna by foo in the file
- If you have special characters {.*[]^$\ }
  - Precede with \
  - eg: sed ‘s/guna\[me\.him\]/foobar/g’ filename

sed basics

- Replacing more than one token
  - sed -e ‘s/guna/foo/g’ -e ‘s/color/colour/g’ filename
- What if / is part of the string to replace?
  - Replace all *afs/andrew* with *afs/cs*
  - Solution: any character immediately following s is the delimiter
  - sed ‘s#afs/andrew#afs/cs’ filename

Basics of awk

Basics of awk

- Uses
  - Use information from text files to create reports
  - Translating files from one format to another
  - Adding functionality to “vi”
  - Mathematical operations on numeric files
- awk also has a basic interpreted programming language
- Basic commands
  - General form:
    - awk ‘<search pattern> {<program actions>} ‘
  - awk ‘/guna/ file -- prints all lines with guna
  - awk ‘/guna/’ {print $1,$2,$3} ‘ file
  - awk -F',' '{if ($5=="MCS") print $2}' roster.txt

exercises

- Download an index.html file from your favorite website
  - use wget
- Change all URL's for example, www.cnn.com to www.foxnews.com
  - use sed

Coding Examples

Scripting Languages

- Many routine programming tasks require custom designed solutions, environments and approaches
  - Extracting data from a roster file
- Scripting languages are ideal for tasks that do not require a "high level" compiled language solution
  - Some argue that this is the real way to learn programming
  - No need to worry about static typing
- Scripts are widely used as backend processing languages for web based applications
  - Authenticate passwords
  - Extract data from a database
  - Create dynamic web pages

Popular Scripting Languages

- JavaScript
  - Client side processing based on a built in browser interpreter
- PHP
  - Server side processing
- Python
  - Object oriented, interpreted, data structures, dynamic typing, dynamic binding, rapid application development, binding other programming components
- Perl
  - Also you can call it an "*interpreted*" language (more later)

Perl

- An interpreted scripting language
  - Practical extraction and Report Language
  - Developed as a tool for easy text manipulation and report generation
- Why Perl
  - Easy scripting with strings and regex
  - Files and Processes
- Standard on Unix
- Free download for other platforms

What's good for Perl?

- Scripting common tasks
- Tasks that are too heavy for the shell
- Too complicated (or short lived) for C

First Perl Program

```
#!  usr/bin/perl  –w
print (“hello world \n”);
```

- How does this work?
  - Load the interpreter and Execute the program
    - perl hello.pl

An interpreted language

- Program instructions do not get converted to machine instructions.
- Instead program instructions are executed by an "interpreter" or program translator
- Some languages can have compiled and interpreted versions
  - LISP, BASIC, Python
- Other interpreters
  - Java interpreter (byte code) and .net CIL
    - Generates just in time machine code

Perl Data Types

- Naming Variables
  - Names consists of numbers, letters and underscores
  - Names cannot start with a number
- Primitives
  - Scalars
    - Numeric : 10, 450.56
    - Strings
      - ‘hello there\n’
      - “hello there\n”

Perl Data Types

- arrays of scalars
  - ordered lists of scalars indexed by number, starting with o or with negative subscripts counting from the end.
- associative arrays of scalars, a.k.a``hashes".
  - unordered collections of scalar values indexed by their associated string key.

Variables

- $a = 1; $b = 2;
- All C type operations can be applied
  - $c = $a + $b; ++$c; $a +=1;
  - $a ** $b - something new?
- For strings
  - $s1 . $s2 - concatenation
  - $s1 x $s2 - duplication
- $a = $b
  - Makes a copy of $b and assigns to $a

Useful operations

- **substr($s, start, length)**
  - substring of $s beginning from **start** position of **length**
- **index string, substring, position**

  look for first index of the substring in string starting from position
- **index string, substring**

  look for first index of the substring in string starting from the beginning
- **rindex string, substring**

  position of substring in string starting from the end of the string
- **length(string)** – returns the length of the string

More operations

- **$_ = string; tr/a/z/; # tr is the transliteration operator**

  replaces all ‘a’ characters of string with a ‘z’ character and assign to $1.

- **$_ = string; tr/ab/xz/;**

  replaces all ‘a’ characters of string with a ‘x’ character and b with z and assign to $1.

- **$_ = string; s/foo/me/;**

  replaces all strings of “foo” with string “me”

- **chop**

  this removes the last character at the end of a scalar.

- **chomp**

  removes a newline character from the end of a string

- **split splits a string and places in an array**

o @array = split(/:/,$name); # splits the string $name at each : and stores in an array

o **The ASCII value of a character $a is given by ord($a)**

Comparison Operators

Comparison	Numeric	String
Equal	==	Eq
Not Equal	!=	Ne
Greater than	>	Gt
Less than	<	Lt
Greater or equal	>=	Ge
Less or equal	<=	Le

Operator Precedence and Associativity

Associativity	Operator
left	terms and list operators (leftward)
left	->
nonassoc	++ --
right	**
right	! ~ \ and unary + and -
left	=~ !~
left	* / % x
left	+ - .
left	<< >>
nonassoc	named unary operators (chomp)
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp
left	&
left	| ^
left	&&
left	||
nonassoc	.. ...
right	?:
right	= += -= *= etc.
left	, =>
nonassoc	list operators (rightward)
right	not
left	and
left	or xor

source: perl.com

More at: http://www.perl.com/doc/manual/html/pod/perlop.html

Arrays

- @array = (10,12,45);
- @A = (‘guna’, ‘me’, ‘cmu’, ‘pgh’);
- Length of an array
  - $len = $#A + 1
- Resizing an array
  - $len = desired size

repetition

A While Loop

```
$x = 1;
while ($x < 10){
  print "x is $x\n";
  $x++;
}
```

Until loop

```
$x = 1;
until ($x >= 10){
print "x is $x\n";
$x++;
}
```

repetition

Do-while loop

```
$x = 1;
do{
    print "x is $x\n";
    $x++;
} while ($x < 10);
```

for statement

```
for ($x=1; $x < 10; $x++){
   print “x is $x\n”;
}
```

foreach statement

```
foreach $x (1..9) {
    print "x is $x\n";
}
```

Parsing a roster entry

- S10,guna,Gunawardena,Ananda,SCS,CS,3,L,4,15123 ,A ,,

Perl IO

```
$size = 10;
open(INFILE, “file.txt”);
$#arr = $size-1; # initialize the size of the array to 10
$i = 0;
foreach $line (<INFILE>) {
   $arr[$i++] = $line;
   if ($i >= $size) {
      $#arr = 2*$#arr + 1; # double the size
      $size = $#arr + 1;
   }
}
```

Perl IO

- open(OUT, “>out.txt”);
- print OUT “hello there\n”;
- Better file open
  - open (OUT, “>out.txt”) || die “sorry out.txt could not be opened\n”

Perl and Regex

Perl and Regex

- Perl programs are perfect for regex matching examples
  - Processing html files
    - Read any html file and create a new one that contains only the outward links
    - Do the previous exercise with links that contain cnn.com only

Regex syntax summary

- ?, +, *
- ( ) - grouping
- (exp (exp)) ➔ \1, \2 or $1 , $2 backreference matching
- ^startwith
- [^exclusion group]
- [a-z,A-Z] – alpha characters

Perl and regex

```
open(INFILE, "index.html");
foreach $line (<INFILE>) {
  if ($line =~ /guna/ ){
      print $line;
   }
}
close(INFILE);
```

Lazy matching and backreference

```
open(IN, “guna.htm”);
while (<IN>){
   if ($_ =~ /mailto:(.*?)"/){
      print $1."\n";
   }
}
```

Global Matching

- How to find all matches on the same line

```
open(IN, “guna.htm”);
while (<IN>){
   if ($_ =~ /mailto:(.*?)"/g){
       print $1."\n";
   }
}
```

Global Matching and Replacing

The statement

```
$str =~ s/oo/u/;
```

would convert "Cookbook" into "Cukbook", while the statement

```
$str =~ s/oo/u/g;
```

would convert "Cookbook" into "Cukbuk".

CGI Scripts and Perl

- CGI is an interface for connecting application software with web servers
- CGI scripts can be written in Perl and resides in CGI-bin
- Example: Passwd authentication

```
while (<passwdfile>) {
 ($user, $passwd)= split (/:/, $_);
 .....
 }
```

LWP
Library for www in Perl

- LWP contains a collection of Perl modules
  - ***use LWP::Simple;***
  - ***$_ = get($url);***
  - ***print $_;***
- *Good reference at*
  - *http://www.perl.com/pub/a/2002/08/20/perlandlwp.html*

Getopt

- The Getopt::Long module implements an extended getopt function called GetOptions().
- Command line arguments are given as
  - **-n 20   or –num 20**
  - **-n 20 -t test**

- ***use Getopt::Long;***
- ***$images_to_get = 20;***
- ***$directory = ".";***
- ***GetOptions("n=i" => \$images_to_get, "t=s" => \$directory);***

References: http://perldoc.perl.org/Getopt/Long.html