Lecture 20-1
Regular Expressions

In this lecture

- Background
- Text processing languages
- Pattern searches with grep
- Formal Languages and regular expressions
- Finite State Machines
- Regular Expression Grammer
- Sed and regex
- Summary
- Exercises

Background

Many of today's web applications require matching patterns in a text document to look up for specific information. Web searches require matching simple patterns (such as a key word) or complex patterns such as a phrase or even more complex combinations of search strings. In these cases, it is not efficient to compare character by character to look for pattern(s). A good example of looking for a specific pattern is parsing a html file to extract <img> tags of a web document. If the image locations are available, then we can write a script to automatically download these images to a location we specify. Looking for tags like <img> is a form of searching for a pattern. Pattern searches are widely used in many applications like search engines. A **regular expression(regex)** is defined as a **pattern** that defines a class of strings. Given a string, we can then test if the string belongs to this class of patterns. Regular expressions are used by many of the unix utilities like *grep, sed, awk, vi, emacs* etc. We will learn the syntax of describing regex later.

Text Processing Languages

There are many scripting languages that can be used to process **textual data** to extract and manipulate patterns of interest. Among some of the popular languages are **awk, sed and perl**. In this course, we will not go in detail into awk (pattern scanning and processing language) and sed (a utility that can be used to transform text streams), but will encourage students to read and understand some of the use cases of these utilities as a general topic of interest. We will however, later focus on **perl**, a popular programming language for parsing textual data. Before we learn perl programming, we will focus on learning regular expressions, a powerful way to describe general string patterns in perl. With the understanding of regular expressions and perl syntax, we can write powerful programs to accomplish interesting tasks.

Pattern search is a useful activity and can be used in many applications. We are already doing some level of pattern search when we use wildcards such as *. For example,

> ls *.c

Lists all the files with c extension or

- **ls ab***

lists all file names that starts with **ab** in the current directory. These type of commands (ls,dir etc) work with windows, unix and most operating systems. That is, the command ls will look for files with a certain name patterns but are limited in ways we can describe patterns. The wild card (*) is typically used with many commands in unix. For example,

- **cp *.c /afs/andrew.cmu.edu/course/15/123/handin/Lab6/guna**

copies all .c files in the current directory to the given directory

Unix commands like ls, cp can use simple wild card (*) type syntax to describe specific patterns and perform the corresponding tasks. However, there are many powerful unix utilities that can look for patterns described in general purpose notations. One such utility is the **grep** command.

The grep command

Grep command is a unix tools that can be used for pattern matching. Its description is given by the following.

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

The grep (Global Regular Expression Print) is a unix command utility that can be used to find specific patterns described in **"regular expressions"**, a notation which we will learn shortly. For example, the "grep" command can be used to match all lines containing a specific pattern. For example,

- **grep "<a href" guna.html > output.txt**

writes all lines containing the matching substring **"<a href"** to the file **output.txt**

grep unix command can be an extremely handy tool for searching for patterns. If we do

- **grep "foo" filename**

it returns all lines of the file given by filename that matches string foo.

Unix provide the | command (pipe command) to send an input from one process to another process. Say for example, we would like to find all files that have the pattern "guna". We can do the following to accomplish that task.

> ls | grep guna

We note again that Pipe command | is used to send the output from one command as input to another. For example, in the above command, we are sending the output from ls as input to grep command.

If we need to find a pattern **a.c** in a file name (that is any file name that has the substring **a.c**, where dot(**.**) indicates that any single character can be substituted) we can give the command

- **ls | grep a.c**

So we can find file name that has the substring aac, abc, a_c etc.

Regular expressions

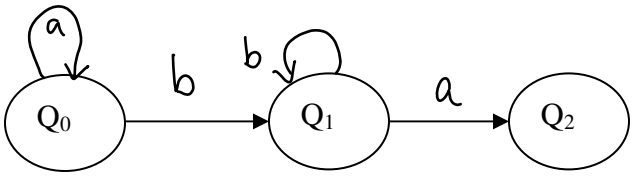
Regular expressions, that defines a **pattern in a string**, are used by many programs such as grep, sed, awk, vi, emacs etc. The PERL language (which we will discuss soon) is a scripting language where regular expressions can be used extensively for pattern matching.

The **origin of the regular expressions** can be traced back to formal language theory or automata theory, both of which are part of **theoretical computer science**.

A **formal language** consists of an alphabet, say {a,b,c} and a set of strings defined by the language. For example, a language defined on the alphabet {a,b,c} could be all strings that has at least one 'a'. So "ababb" and "abcbbc" etc are valid strings while "ccb" is not.

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. 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: think about a FSM that has an alphabet {a,b} and 3 states, **Q_0, Q_1, and Q_2**
Define Q_0 as the initial state, Q_1 as intermediate and Q_2 as the final or accepting state. Complete the transitions so that the machine accepts any string that begins with zero or more a's immediately followed by one or more b's and then ending with an 'a'. So the strings accepted by this FSM would include "aba", "aaba", "ba", "aaaaaabbbbbbbbba" etc.



Finite automata (FA) can be built to recognize valid strings defined by a formal language. For example, we can use a machine as defined above to find all substrings that begins with zero or more a's immediately followed by one or more b's and then ending with an 'a'.

One important feature of a finite state machine is that it cannot be used to count. That is, FSM's have no memory. For example, we can build a machine to accept all strings that has even number of a's, but cannot build a FSM to count the a's in the string.

Our discussion on FSM's now leads to regular expressions. A regular expressions and FSM's are equivalent concepts. *Regular expression is a pattern that can be recognized by a FSM. We note that we have kept our discussion to Deterministic Finite Automata(DFA)*

Regular Expression Grammar

Regular expression grammar defines the notation used to describe a regular expression. We discuss here the basic concepts of regular expression grammar **including alternation, grouping and quantification**. It should be noted that these grammar may not work exactly as is in every system. The grammar defined here are applicable to strong regex based languages such as perl.

Alternation

The vertical bar is used to describe alternating choices among two or more choices. For example, 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".

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".

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. For example "aab?b" would represents the strings "aab" and "aabb" etc. Color and Colour can be described by the regex "Colou?r".

The **"*" indicates that zero or more** of the previous expression can be accepted. For example: "**a(ab)*b**" indicates any string that begins with a, ends with a b, but can have any number of the substring "ab" in the middle. The strings "ab", "aabb", "aababb" are all valid strings described by the regex **a(ab)*b**.

The **"+" indicates at least one** of the previous expression. For example "go+gle" would describe the expressions "gogle" , "google" , "gooogle" etc

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**

grep command used with regular expression notation can make a powerful scripting language. When using grep, be sure to escape special characters with \.
Eg: grep 'b(a|c) b' looks for patterns bab or bcb specifically

Example
1. Find all subdirectories within a directory
Answer: > ls –l | grep "^d"

Character Classes

Character classes such as digits (0-9) can be described using a short hand syntax such as \d. A PERL(a language we shortly learn) programmer is free to use either [0-9] or \d to describe a character class.

- **\d digit [0-9]**
- **\b matches a word boundary – zero length matching**
- **\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]**

Note: we will not use these shorthand notations with grep

Other more general regex grammar includes
1. {*n*,*m*} at least *n* but not more than *m* times
2. {*n*,} – match at least *n* times

3. {n} – match exactly n times

Examples:
Find all patterns that has at least one but no more than 3, 'a's
(ans: **grep "a\{1,3\}" filename)**

Finding non-matches

To exclude patterns we can use [^class]. For example, to exclude patterns that may contain a numeric digit, write [^0-9]. For example, we can exclude all lines that begins with a-z by writing
> grep "^[^a-z]" filename

Group Matching

If we group a match by using ( ) then the matching groups are given by **1, 2** etc..
For example a regex

"<h\([1-4]\)>.*h\([1-3]\)>"

Returns **1** as the number that matched with the first group **\([1-4]\)** and
2 as the number that matched with the second group **\([1-3]\)**

This can be useful in looking for patterns based on previous patterns found. For example The regex

h[1-4] can match to h1, h2, h3, or h4.

Suppose later in the expression we need to match the same index. We can do this by grouping **[1-4] as \([1-4]\)** --- note we need \(to make sure that (is not used as a literal match ---
Now the match is saved in a variable **1** (must refer to as **\1**) it can be used later in the expression to match similar indices. An application of this would be to look for
<h1> …. <\h1> but not **<h1> …. <\h2>**

Here is how you do that.

> grep "h\([1-4]\).*h\1" filename

In general, the back-reference **\n**, where **n is a single digit**, matches the substring previously matched by the nth parenthesized sub expression of the regular expression. This concept is sometimes called **back reference**. We will expand these ideas later in Perl programming.

Sed and RegEx

Sed is a utility that can transform text streams. For example a pattern in a stream can be substituted by a new pattern.

➢ **sed -e 's/hand/feet/' filename**
will substitute pattern *hand* with *feet*

or
➢ **sed -e '/^h/d' files.out**
will delete all lines that starts with h from files.out. It is important to keep in mind that sed is only changing the stream that passes through a sed filter. Therefore in the above example if one wants to alter files.out then the resulting stream needs to be directed to another file.

An interesting application of sed would be to highlight the words in a given file. Suppose we need to highlight every occurrence of "she" in a file. So we would write
➢ **sed –e 's/she/ she<\/span>/' index.html**

References: http://www.ibm.com/developerworks/linux/library/l-sed1.html

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 subexpressions.

- Two regular expressions may be joined by the infix operator | the resulting regular expression matches any string matching either subexpression.

- Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression 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 subexpression of the regular expression.

- **In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \).**

Source: Unix Manual

Exercises

1. Build a FSM that can accept any string that has even number of a's. Assume the alphabet is {a,b}.
2. Using grep command and regular expressions, list all files in the default directory that **others** can read or write
3. Write a regex that matches the emails of the form userid@domain.edu. Where userid is one of more word characters or '+' and the domain is one or more word characters.

4. Construct a FSM and a regular expression that matches patterns containing at least one “ab” followed by any number of b’s.
5. Write the grep commands for each of the following tasks
   a. Find all patterns that matches the pattern “ted” or “fred”
   b. Find all patterns that matches ed, ted or fed
   c. Find all patterns that does not begin with “g”
   d. Find all patterns that begins with g or any digit from 0-9
   e. Find all patterns that begins with “guna”
   f. Find lines in a file where the pattern “sam” occurs at least twice
   g. Find all lines in a file that contain email addresses
6. Write a regex that matches any number between 1000 and 9999
7. Write a regex that matches any number between 100 and 9999
8. Write a regex that lists all the files in the current directory that was created in Nov and are txt files.

ANSWERS

1. Build a FSM that can accept any string that has even number of a's. Assume the alphabet is {a,b}.
2. Using grep command and regular expressions, list all files in the default directory that **others** can read or write
   Ans: ls –l | grep '.\{7\}rw'
3. Write a regex that matches the emails of the form userid@domain.edu. Where userid is one of more alpha characters or '+' and the domain is one or more alpha characters.
   Ans: grep [a-z+]\+@[a-z]\+.edu
4. Construct a FSM and a regular expression that matches patterns containing at least one "ab" followed by any number of b's.

   Ans: grep '\(ab\)+b*'

5. Write the grep commands for each of the following tasks
   a. Find all patterns that matches the pattern "ted" or "fred" **[t/fr]ed**
   b. Find all patterns that matches ed, ted or fed
      [t|f]?ed
   c. Find all patterns that does not begin with "g"
      ^[^g]
   d. Find all patterns that begins with g or any digit from 0-9
      ^(g|[0-9])
   e. Find all patterns that begins with "guna"
      ^(guna)
   f. Find lines in a file where the pattern "sam" occurs at least twice
      (sam).*\1
   g. Find all lines in a html file that contain email addresses (hint: mailto)

6. Write a regex that matches any number between 1000 and 9999
   Ans: [1-9][0-9]{3}