CSCE A351
Intro to Unix, Java Regular Expressions

Regular expressions are used quite frequently in Unix applications and in various programming languages to search for patterns in text. For example, the grep program is able to search for patterns based on regular expressions. Lex is able to use regular expressions to perform a lexical analysis. It is intended to be used for a compiler of a programming language so that the language may be parsed and lexically analyzed in terms of the corresponding syntax and semantics.

Unix regular expressions are character-based and have a somewhat different syntax from what we have described thus far. Here are the differences:

Wildcards

The shell and some other unix programs (e.g., find) use wildcard characters. The name comes from card games where a "wild" card can stand in for any other card. Wildcards are sometimes referred to as meta-characters.

*	- Match 0 or more instances of any character
?	- Match a single instance of any character
[123x]	- Match a single instance of any character in the brackets
[0-9]	- Shorthand for [0123456789], often used with [A-Z]
[^0-9]	- Match a single character **except** those specified in brackets

Examples:

ls a*	; Lists all files that start with the letter 'a'
ls ?.c	; List any files with a single letter followed by .c
ls *[0-9]*	; List any files with a digit in the name
ls [^t]*	; List any files that do not start with the letter 't'

Regular Expressions

Regular expressions in Unix (POSIX) use the same wildcard format except for * and ?.

. (dot)	- Stands for any character except newline
|	- Denotes union (i.e. OR, same as + in our algebra)
R?	- Zero or one of regular expression R (i.e. R+ε)
R+	- One or more of regular expression R (i.e. RR*)
R{n}	- Means n copies of R, e.g. R{3} means RRR
R*	- Zero or more repetitions of R, i.e. R*
^	- Match the beginning of the line (note different context than NOT)
$	- Match the end of the line
\	- Escape character, e.g. * means *
( )	- Precedence and expression grouping

[123x] - Match a single instance of any character in the brackets
[0-9] - Shorthand for [0123456789], often used with [A-Z]
[^0-9] - Match a single character **except** those specified in brackets

Note the difference between * in wildcards and * with regular expressions. Foo* as a wildcard will match anything starting with Foo. Foo* as a regular expression will match Foo, Fooo, Foooo, Fooooo, etc. There is a similar distinction with the ? and + operators.

Using RE's with grep

Let's take a look at using some regular expression with grep, a utility that will search for patterns in a text file(s). Here we'll actually use egrep, which is just grep with the –E option turned on (parses "extended" regular expressions).

The format for using egrep is:

egrep [flags] regular-expression filelist-to-search

The flags are optional. For example, the –i flag instructs egrep to ignore case. There are many other flags you can view from the man page.

Egrep will search all files in filelist for text that contains the regular expression as a substring. By default, it prints out the line that matches the regular expression.

For example:

egrep foo *

Will search all files for text containing "foo" and print any such lines that contain foo.

Consider the following file stored in 'bookfile':

This file tests for book in various places, such as
book at the beginning of a line or
at the end of a line book
as well as the plural books and
handbooks.

The commands:

egrep ' book ' bookfile

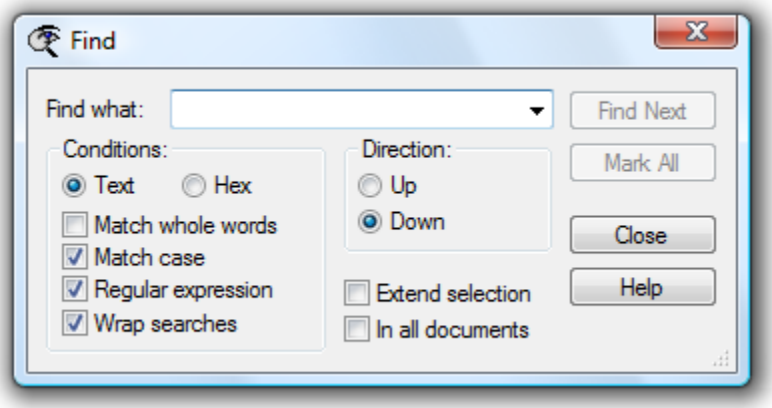
Will only match line 1.

egrep 'book' bookfile

Will match all lines. What does the following match?

egrep ‘book$| book |^book’ bookfile

Here is a more complex example. We could also use grep, but as a different example let’s use TextPad. TextPad includes regular expression searches if the box is ticked:



If you click on “Help” and search for regular expressions, there are a few differences. Notably, in older versions of textpad, parentheses and the | symbol need to be escaped by the \ character.
So to get parenthesis as regular expressions you would use:

\(\)

And to get the | you would use:

\|

This is not the case in more recent (7+) versions of Textpad.

Consider the following text:

Hello Myra,
The meeting is on 2/5 in room 10
and will last from about 3-4 PM.
The document number is 13/2.
Please confirm prior to 1-30-16.

Thanks.

Similar to the homework assignment, we would like to write a regular expression to find potential dates in the file. Let’s start by writing the regular expression to find a month followed by a slash as “digits from 1-9 or 1 followed by 0,1, or 2”:

```
([1-9]|1[0-2])/
```

If you try it out then it matches the following:

The meeting is on 2/5 in room 10
The document number is 13/2.

It doesn't quite work! It also finds 13/2 as a match to the regular expression. This is because the regular expression returns matches for any substring match. This includes 13/ since 3/ is included in 1-9, and the leading 1 is ignored as part of prior string. To address this issue we can include the preceding char in this case should exclude a digit:

```
([^0-9][1-9]|1[0-2])/
```

This matches:

The meeting is on 2/5 in room 10

We now only get the first line. Now let's add into our expression the trailing digit for the day, which can be 1-31. We can match this by allowing either a single 1-9 followed by a space or end of line, or 1-3 followed by 0-9 (this isn't quite perfect since it allows invalid dates like 39, but we won't worry about that here… you should be able to figure out how to address this though).

```
([^0-9][1-9]|1[0-2])/([1-9]|[1-3][0-9])
```

Now, we may note that some dates are separated by – instead of by /. We can expand the expression to include dashes by allowing either one:

```
([^0-9][1-9]|1[0-2])[/-]([1-9]|[1-3][0-9])
```

The matches are:

The meeting is on 2/5 in room 10
and will last from about 3-4 PM.
Please confirm prior to 1-30-16.

Note that we only get 1-3 and not the 0 because we are matching [1-9] before [1-3][0-9]. If we want to match the two digits first we can change the order:

```
([^0-9][1-9]|1[0-2])[/-]([1-3][0-9]|[1-9])
```

The meeting is on 2/5 in room 10
and will last from about 3-4 PM.
Please confirm prior to 1-30-16.

If we want to pick up a 2 digit year we could make it optional immediately after the day has been parsed:

```
([^0-9][1-9]|1[0-2])[/-]([1-3][0-9]|[1-9])([/-][0-9][0-9])?
```

The meeting is on 2/5 in room 10
and will last from about 3-4 PM.
Please confirm prior to 1-30-16.

The 3-4 PM is a false hit in this case. The easiest way to exclude it would be to add some programming. With some good old logic we could look at the next token after a potential date and see if there is an AM or PM and if so, exclude it. This suggests it would be helpful to be able to integrate regular expressions with programming languages!

Regular Expressions in Java

Most programming languages also support regular expressions, either natively or with some sort of add-on library. Java supports regular expressions as of Java 2 version 1.4.

For a detailed description of how to use regular expressions in Java, refer to the following webpages:

http://developer.java.sun.com/developer/technicalArticles/releases/1.4regex/

and

http://java.sun.com/j2se/1.4/docs/api/java/util/regex/package-summary.html

A short description of the Java regular expression utility follows.
In Java's regular expressions, the same format is allowed as with Unix regular expressions. There are a few differences.

Here are some useful predefined character classes:

`\d` A digit: `[0-9]`

`\D` A non-digit: `[^0-9]`

`\s` A whitespace character: `[ \t\n\x0B\f\r]`

`\S` A non-whitespace character: `[^\s]`

`\w` A word character: `[a-zA-Z_0-9]`

`\W` A non-word character: `[^\w]`

Here are other useful patterns:

\n	-	Whatever the nth capturing group matched (e.g. \1; \n is newline)
\b	-	A word boundary
\B	-	A non-word boundary
\A	-	The beginning of the input
\z	-	The end of the input
X{n,m}	-	X, at least n but not more than m times

Creating regular expressions in Java is complicated somewhat, because we need to create an actual string that contains the backslashes. For example, let's say that we want to create the regular expression that captures an integer separated by word boundaries. One way to do this is as:

\b[0-9]+\b

If we try to implement this string in Java as:

"\b[0-9]+\b"

then Java will interpret the \b as an escape character for backspace, which really translates to the string: (backspace)[0-9]+(backspace) which is not what we want!

Consequently, to properly construct the regular expression, we must use \\ in Java, which is the escape character itself. The proper string would be:

"\\b[0-9]+\\b"

The bottom line is to be careful how you create the regular expression in Java, as it must contain all the escape characters to print the backslash, double quotes, etc.

Here is a simple program that matches the word "cat":

```
import java.util.regex.*;
public class RegExprTest {
    public static void main(String[] args) throws Exception {
      String text = "One cat, two cats, how fun.";
      // Create a pattern to match cat
      Pattern p = Pattern.compile("cat");
      // Create a matcher with an input string
      Matcher m = p.matcher(text);
      boolean result = m.find();
      // Loop through and create a new String
      // showing what we matched
      while(result) {
            // m.group returns the matching String
            System.out.println(m.group());
```

```
            result = m.find();
        }
    }
}
```

The output of this program is:

cat
cat

matching each occurrence of ‘cat’ and ‘cats’ in the input string.

Here is another example that illustrates replacing the word “cat” with “dogs”:

```
/*
 * This code writes "One dog, two dogs in the yard."
 * to the standard-output stream:
 */
import java.util.regex.*;

public class Replacement {
    public static void main(String[] args)
                        throws Exception {
        // Create a pattern to match cat
        Pattern p = Pattern.compile("cat");
        // Create a matcher with an input string
        Matcher m = p.matcher("one cat," +
                    " two cats in the yard");
        StringBuffer sb = new StringBuffer();
        boolean result = m.find();
        // Loop through and create a new String
        // with the replacements
        while(result) {
            m.appendReplacement(sb, "dog");
            result = m.find();
        }
        // Add the last segment of input to
        // the new String
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

Going back to the format of the first example, can you tell how the following work?

```
1. Email addresses

String text = "Email me at me@my.address.com, it's my work address";
Pattern p = Pattern.compile("([A-Za-z0-9_\\.-]+)@([A-Za-z0-9_\\.-]+[A-
Za-z0-9_][A-Za-z0-9_])");

2.  Money

String text = new String("Make $50,000.00 fast with this no risk
system!");
```

```
Pattern p = Pattern.compile("\\$\\d{1,3}(,?\\d{3})*(\\.\\d{2})?\\b");
```

3. IP Address

```
String text = new String("My IP address is 234.21.123.0");
Pattern p = Pattern.compile("[1-2]?[0-9]{1,2}\\.[1-2]?[0-9]{1,2}\\.[1-
2]?[0-9]{1,2}\\.[1-2]?[0-9]{1,2}");
```

4. Repeated words

```
String text = new String("This is is is issued");
Pattern p = Pattern.compile("\\b(\\w+)(\\s*\\1)+\\b");
```

5. Caps Lock Syndrome

```
String text = new String("There is SOMETHING Wrong with mY cAPS lOCK
Button!");
Pattern p = Pattern.compile("\\b[a-z][A-Z]+\\b");
```