Muddling Through Regular Expressions

Presented by:
Fred Mallett frederm@aol.com
FAME Computer Education
250 Beach Blvd
Laguna Vista, TX 78578
(956) 943-4040
http://www.famece.com

Note: This material was extracted from the F.C.E. training course
“ Regular Expressions”

Introduction & Basics

Description

Regular Expressions are used to match, or manipulate strings of text.

A Regular Expression is developed using a pattern notation consisting or literal and metacharacters, and is used to describe sequences of text characters.

In a simpler definition, a Regular Expression is a set of literal and special characters (meta-characters) used to search for matches in text.

Or even: A Regular Expressions describes a set of possible strings.

It is good to think of the Regular Expression Notation itself as a programming language. This language (in several flavors) is embedded into many common tools and programming languages, such as:

ex grep egrep vi nedit emacs sed more less perl tcl python awk JavaScript lex expect

and more.....

In this class, the term ” Regular Expression” will be abbreviated as: RE, re, or rexexp.

Regexps add a great deal of power to text manipulating programs. Using regexps, the various tools can:

Determine if there is a match for an inexact string

Substitute a string or expression result for what was matched

Return the matched portion of a string

Address matched portions for use as an operators operand

Why ” Regular” Expressions?

Below is a summary of the ” Mastering Regular Expressions” books description of how the term came about in computers (Page 60).

Around 1950 Stephen Kleene (a mathematician) formally described models of how some neurophysiologists thought the nervous system worked.

Stephen described them in an algebra he called regular sets. He then developed a notation to express these regular sets, and called them Regular Expressions.

These regular expressions were used primarily in mathematic circles.

In 1968 Ken Thompson wrote an article called Regular Expression Search Algorithm which described the roots of the ed editor, which became ex. This was the beginning of regular expressions in computer circles.

ex had the g command, whose common use was g/RE/p to print lines which contained a match for RE. Eventually, this grep function became its own command.

Learning Rexexps

How a regexp is used is under the control of the program it is used by
This can vary widely in:

- Which metacharacters are accepted (and which need to be escaped)
- Details about what a metacharacter can ” change” to
- How the search is performed
- What happens when a match is found
- What the RE will match
- How you can use the results of the match (and what is returned)

The good news is, once you understand the language of Regular Expressions, learning any particular ” dialect” and its various peculiarities is eas(y|ier).

This course covers writing regexps, using tools like sed, grep, and perl since they are found on most systems. Other tools are addressed when needed.

There are many levels of proficiency in the trip towards competence in regular expressions:

Getting started

- Know the basic RE metacharacters, and create RE’s using them
- This level allows you to save time in performing simple tasks

Getting there

- Know how the expressions characters interact, and know what the ERE characters mean
- This level allows you to use many different tools, and solve many common problems

You made it

Know how all meta-characters interact, and can use all metacharacters that your choice of tools is capable

This level allows you to solve difficult problems with ease

Above and Beyond

Can write efficient dynamic expressions, understanding the difference between expressions that might give the same result using multiple tools

As Jeffrey E.F. Friedl said:

> ” To Master regular expressions is to master your Data”

Regexp usage

Literal regexp

A literal regexp consists entirely of ” ordinary” characters, those with no special (programatic) meaning.

Though simple, these are very useful, and can save a great deal of time.

Examples of literal expressions are strings like: error warning

We will use literal expressions in many of the following examples.

Binary Match

The simplest use of regexps is the binary match usage.

In this usage a tool ” tests” a string to see if it contains a match for a regexp, exactly what it matches is irrelevant.

Almost all tools that accept regexps can perform this type of match. Some tools are limited to this type of match.

The command grep is probably the most commonly used regexp tool, and is limited to the binary match function. The problem with using grep to learn regexps is that you never know exactly what was matched, only if there was a match. grep prints the entire line from an input file if there is a match anywhere on that line, even if the match length is zero characters.

Binary matches are commonly used for addressing lines in an editing type function.

Addressing match (or specific match)

In this type of match, exactly which characters matched is important, as the matched characters are what is acted upon by the operation that called the regexp search to be performed.

Many tools have a substitute operation which uses an addressing match.

Other uses are for things like:

- addressing a substring to be used as the operand of an operator
  - (mayby perform math on a matched numeric substring)
- splitting a string into fields by removing matches
- returning the matched string, removing the matched string

Match side-effects

Many tools also retain information about the most recent regexp search, using this information is often called using the side-effects of a match.

RE Testing tool:

This simple Perl (but verbose) program works reasonably well for testing re's:

```
#!/usr/local/bin/perl
$re='\d*';
while (<STDIN>) {
      if (/$re/) {
        print " Matched line $.:\n"  ;
        print $_;
        if (length($&) == 0) {
            print " Zero length match!!\n" ;
        } else {
            print " $&\n" ;
        }
      } else {
```

```
        print ” No match line $.\n”  ;
}     }
```

Issues

Thinking of regexps as a programming language embedded into many tools and higher level languages is a good idea.

Thinking of regexps as a type of foreign language helps understand many of the issues you will face when trying to learn them:

- As with any language, there are many local terms, not found elsewhere (why have 100 names for snow in florida?), and many different dialects that can make understanding things difficult, even when using the same language.

The next few pages have some more details of these dialect differences, and other issues we must deal with when learning regexps.

Note: UNIX shell wildcards are not regexps (though they share some characteristics, do not confuse the two, they have very little in common.

Passing Regexps

Though regexps are interpreted by a piece of code within the various tools, the regexp string itself is often ” handled” by programs that do not ” understand” that it is a regular expression. This can complicate the escaping needed in a regular expression. A good example of this is a command that takes a regular expression as an argument on the command line, like grep:

```
grep x;y # matches and prints lines with x, then runs y
```

The above had to be written in one of the manners below, since the ’;’ had special meaning to the shell, though it is just a literal (ordinary) character to the regexp language:

```
grep ’x;y’      grep ” x;y”          grep x\;y
```

It can get rather complex when a character has meanings to the interpreter, and regexps. For example, since ’*’ has meaning to both the shell, and regexps, if we wanted to match a literal ’*’, we would have to escape its meaning from the shell, and also from the regexps of grep:

```
grep x\\*y   grep ’x\*y’    grep ” x\*y”
```

This means we need to understand text issues with the tools used, as well as know regexps.

In some languages, regexps are first parsed as strings, then used as regular expressions, which can add even similar complexity.

Tool differences

Another issue is that there are so many different tools using regular expressions, it is difficult to learn the myriad of details. The good part is that most ”masters” of regexps use them primarily in one or two tools, and use only simpler expressions in the other tools as needed. (There is a benefit to specialization).

The Major categories of differences in regexps are driven by what ”engine” is used to ”execute” the regexp. This will be delved into in much greater detail later in the course, for now a brief summary of the two:

DFA (Deterministic Finite Automaton)
- Often faster than NFA.
- You do not need to write efficient regexps, speed will be the same, no matter how it is written.
- Does not support backreferencing (the ability to create dynamic expressions).
- Always matches the longest possible match.
- Examples of commands that use the DFA engine are: *awk*, *grep*, *lex*.
- (DFA is fast and consistent)

NFA (Nondeterministic Finite Automaton)
- You must write efficient expressions or speed can suffer.
- Usually matches the longest match, but might not.
- Often support backreferences (easy to do with this engine).

Examples of commands that use the NFA engine are just about everything else (like *perl*, *vi*, *sed*, *javascript, tcl...)*.

In addition to the engines in use, there are two categories of sets of meta-characters used by tools. These are:

BRE (Basic Regular expressions)
ERE (Extended Regular Expressions)

There are also many tool specific metacharacters that don’t fit into these two catagories.

POSIX regexps

Posix defines a list of what metacharacters are in BRE, and which are in ERE. We will follow that list fairly closely in the next few chapters when listing metacharacters, with some tool specific exceptions. (Text book page 64)

The POSIX standard affects how regexps work. For example, it states that the longest match must always be returned, so, if POSIX is enabled, and an NFA tool is used, it must still always return the longest match for a given expression (even though NFA tools don't normally do that for all expressions).

In addition, the POSIX definition tries to provide definitive answers to how every set of metacharacters should be interpreted (what they should match), but, like always, detailed instructions are hard to follow, so there are still some differences in interpretation between two " fully compliant" POSIX versions of tools.

More dialect issues

We will see as we go along that in addition to the above mentioned differences, there are many minor dialects to regexps, and are often found in the details like:

" Should a metacharacter that matches any character,
also match the newline character?"

" When supplied a set of items to match, should it stop on the first successful, or

check them all to match the longest possible of the items?”

```
(a|abba)
```

Using Binary matches

All tools support binary matches.

Binary matches are commonly used to check if a pattern exists in a string, commonly, in commands, a string is an entire line in a file, but in programming languages, the string to be tested is often assigned.

The most common use of a binary match is to act on a line if it contains a match for a regexp. The tool grep uses a binary expression test to decide if it should print a line of input.

In the large majority of tools, you can both invoke, and define a regular expression by enclosing it in slashes:

```
/regexp to test for/
```

This is the method we will use in most examples in the handout.

The exceptions are those commands that take a regular expression as a command line argument, such as grep:

```
grep 'error' result_file
```

The command above is the same as the perl code:

```
while (<>){
  print if /error/;
}
```

or the awk command (code):

```
awk '{if (/error/)  print }' result_file
```

or the sed command:

```
sed -n '/error/p' result_file
```

All examples would print all lines in the file result_file that contain the string *error*.

For example, they would print both of the lines below:

```
error 127
Holy Terror!!!
```

Due to the simplicity of the regexp (a literal string), it is easy to see exactly what was matched in each line, though the tools were instructed to print the entire line if there was a match anywhere.

Using Binary matches

Many editors and web sites allow the use of regexps to search for something. Generally, they either display what matched (like a web search engine), or move the cursor to the first match (many editors), or possibly display all matches of the expression.

When working interactively, getting more than you asked for is not that bad.

When trying to automate procedures, it can be very bad.

What if we wanted only lines with the word error on them, instead of having the word terror also match?

That is the basis for the power of regular expressions: the ability to describe not only the literal text desired, but variable situations of text.

Most importantly, you must know your data!! If the word error was always lower case, at the beginning of a line, and followed by a ” :” , life might be much simpler.

Basic Regular Expression Metacharacters

Meta-characters come in 4 flavors, grouped by usage:

Position matchers (anchors)

These characters do not match anything in the string or line you are testing, they match only a location. The availability of thes characters varies widely among the different commands and tools.

^ $ \b \B \A \Z \G \<

Single character matches

The trick to remember here is that these meta-characters, and meta-sequences will always match exactly one character, or fail the match if there is not a character for them to match.

. [ ] [^] \w \W \s [[:set:]]

Quantity modifiers (quantifiers)

These characters have no meaning when used alone, but take on a significant meaning if there is any character (literal or mets) before them in a regexp. They are extremely powerful, they add a numeric quantity to another characters meaning (how many of those).

+ * ? { }

Quantifiers take a bit of a trick to get used to reading. You must read the quantity before what is being quantified or the result is confusing.

It is also important to know how many preceeding characters the numeric modification affects, and sadly, this can vary by tool.

Control, usage, and grouping

These characters are somewhat difficult to get used to, as different escaping (or none) is used indifferent tools. They fill in the highest power levels, such as changing precedence of characters, back referencing, and dynamic control within an expression.

( ) \ $match pos()

Basic Regular Expression Anchors

There are several locations that can be referenced in an expression, but BRE's only have two.

Another way to say this is that you can anchor a match of an expression to specific locations within the search string.

Some of these can occur multiple times in the string being searched, thus multiple times in an expression.

These two anchor symbols are available in all regexp implementations:

```
^  beginning of string
$  end of string
```

The meaning of beginning and end can vary a bit depending on the tool. We will revisit these in a later chapter. For now, we will take the common meanings.

For tools like grep, ex, and vi, these anchors refer to the beginning or end of a line, as they are line based tools.

For programming type tools, (awk, perl, tcl, JavaScript,...), they usually mean beginning and end of the string being tested for a match. This is usually an arbitrary string assigned to a variable. (You can aso guess that the string might contain less than (no newline), or more than a line of text).

Continuing the previous example, looking for *error* only when it is the first thing on a line, we could use this:

```
/^error/
```

Which means ” error at the beginning of a line (string)” . What would these expressions mean (in verbose english):

```
/^error$/  /^$/       /END$/          /^  /
```

These anchor characters only have meaning when they are used as the first and last characters in a string.

For example, the regexp /x^y/ means match those three characters literally, as the ” ^” was not the first character in the regexp, therefore it is literal.

A ” $” used other than as the last character can have different meaning depending on the tool, we will discuss this in detail later.

Basic Regular Expression Single Character Matches

Remember that these metacharacters match exactly one character in the string (line) being searched.

There are 3 ways to match a single character:

.	Match any single character
[ ]	Match any character in the enclosed class
[^]	Match any character except those in the enclosed class

When there must be a character at a certain location, but you don't care what it is, use the period metacharacter.

When there must be a character at a certain location, and it can only be one from a set of possible characters, use the character class metacharacter.

For example:

/^.../	Match the first 3 characters on lines that have at least 3 characters
/error: .../	Match " error" , colon, space, and the next 3 characters

Looking at the " error" expression above, what if the data we were looking for was the word error, a space, then a 3 character number. We could use:

/error .../

Which would work, but it would also match: " terror to go there"

Like always, it is easy to match what we want, it is more difficult to only match exactly what we want.

Using the character class metacharacter, we can define a match for number (integer) characters:

```
[0123456789]   which is easier written: [0-9]
```

Then we can write a more explicit expression:

```
/error [0-9][0-9][0-9]/
```

Many tools allow abbreviations of common character class sets. Digits for example, can be written \d in several tools (note we no longer use the brackets):

```
/error \d\d\d/
```

Basic Regular Expression Single Character Matches

We can supply a backslash to add meaning to some characters in a character class (which varies by tool):

```
[\t] tab  [\n] newline  [\r] return  [\f] formfeed
```

Most tools also accept a special first character in the class that negates the set of characters in the class, meaning ” any character that is not one of these” :

```
[^0-9]     match any single character that is not
a digit
```

Some of the more common ranges and sets of characters used in character classes and the most common abbreviations:

```
[0-9]      digits                    \d
[^0-9]          non-digits           \D
[a-z]      lower case alphas
[A-Z]      upper case alphas
[A-Za-z]   alphas
[a-zA-Z0-9]alphanumerics             \w
[^a-zA-Z0-9]    non-alphanumerics          \W
     Note that some tools include ” _”  in
alphanumerics
[ \t\n\r\f]white space           \s
[^ \t\n\r\f]    non-white space      \S
```

POSIX enabled tools also accept another set of abbreviations:

```
[[:alpha:]]alpha
[[:alnum:]]alphanumerics
[[:blank:]]space and tab
[[:space:]]whitespace characters
[[:cntrl:]]control characters
[[:digit:]]digits
[[:lower:]]lower case alpha
[[:upper:]]upper case alpha
```

There are several more, see page 80 in the text book.

To use these, note that the inner set of brackets are part of the character class name:

```
/error:[[:space:]][[:digit:]][[:digit:]][[:digit:]]/
```

Which means error, any space character, then 3 digits. The beauty of the POSIX classes is that they understand Locale information.

Basic Regular Expression Single Character Matches

Generally speaking, any character inside brackets loses its meaning. Thus, you can match a literal period with:

```
/[.]/
```

Remember that /./ means match any single charater.

Another method would be to escape the period with a backslash:

```
/\./
```

Some people prefer to use brackets, some prefer the backslash. Keep in mind that in some cases you would need two backslashes, as the shell reading the command, or the string parsing of the programming language, would try to interpret the backslash.

Some characters already have special meaning inside brackets. In the case of those characters, escaping them inside the brackets removes special meaning, allowing them to become a member of a character class set:

```
/[:-_]/     Matches all characters between : and _
            in ascii chart   (:;[]^@<>\?=_)

/[:\-_]/    Match a :, -, or _

/[-;_]/     Match a :, -, or _ (- can’t mean range
here)
```

Don’t be suprised if you see spaces put in brackets (for readability):

```
/error:[ ]\d\d\d/
```

A difficult issue to get used to is that a single character can have so many meanings,

depending on where in a regexp it is used. For example, the ^ character can be literal, a beginning anchor, or negate a character class. Hint: Get used to it.

Here is a summary of the meanings of ^:

```
/^x/  beginning anchor (x at beginning of the string/line)
/x^/  literal (an x followed by a ^)
/[^x]/     negate a class (any character except an x)
/[x^]/     literal (an x or an ^)
```

Basic Regular Expression Quantifiers

Quantifiers are characters that add meaning to what preceeds them. Putting a quantifier after a literal character sequence, or metacharacter, means to allow a number of what those characters match.

Consider the previously used:

```
/error [0-9][0-9][0-9]/
```

It could have been written simpler with the {} quantifier, which accepts a numeric value:

```
/error [0-9]{3}/
```

{ , } is called the interval operator.

If the ” error code” might be from 1 to 1000, we might have from one to four digits. The following expression would match just one, two or up to 4 trailing digits, depending on how many were in the string:

```
/error [0-9]{1,4}/
```

This would have matched: error 4 error 92 error 573 error 234877

You can also create ” open ended” ranges of numbers as follows:

```
{,3}        match 0,1,2 or 3 of what preceedes
{1,}        match 1 or more or what preceedes
```

Some useful combinations using interval are:

```
{n}         exactly n
{,n}        up to n, zero is ok
{n,}        n or more
{n,m}       n is the minimum number, m is the
maximum
```

Basic Regular Expression Quantifiers

The primary issue when using quantifiers is what does it act on? Another way to say this is ″ what is considered to be ″ preceeding″ the quantifier?″ :

```
/test{3}/          match testtesttest or testtt?
/error[0-9]{3}/    match error and 1 digit 3 times?
or...
```

The general rule is that a quantifier acts on only the single preceeding item:

```
/test{3}/                  matches testtt
/error[0-9]{3}/            matches error and 3 digits
/^[ ]{0,1}CELL[ ][0-9]{1,}/
```

We will see later that you can use grouping (precedence control) in some tools to change the ″ scope″ of a quantifier:

```
/(the ){2}/                match ″ the the ″
```

NOTE: Although {} is in the POSIX definition of BRE, it is not found in all ″ basic″ tools.

Basic Regular Expression Quantifiers

The asterisk (*) is a quantifier meaning the same as {0,}.

For example, to re-write the error example using asterisk:

```
/error [0-9][0-9]*/   match error, space, and all following
                      digits (one or more)
```

Note that we used the digit class twice with an asterisk to mean one or more.

The difficult part to grasp is the zero in zero or more. For example:

```
/error [0-9]*/
```

Would match even if there were no digits following the ”error”. This is because * can mean zero of the preceeding item (digits). Think of it as ”zero or more, but no match needed”.

```
/error [0-9][0-9]*/
```

In this case, the underlined digit is NOT affected by the asterisk, so it MUST be matched, the second digit IS affected by the asterisk, so it is optional due to the ”zero or more” meaning. Thus, the above regexp actually reads in english as ”'error', space, one or more digits”.

Both cases (zero and one or more) are useful:

```
/^[ ]*CELL[ ][ ]*[0-9][0-9]*/
```

Note that the above expression, when used only as a binary match, could have been shortened to:

```
/^[ ]*CELL[ ][ ]*[0-9]/
```

As we do not really care how many digits, just that at least one was there. If we were going to act on what was actually matched (like in a substitution), it would be important to match all the digits.

As with the {} character, you can use grouping in most tools to make the asterisk act on more than the single preceeding item:

```
/([0-9][0-9]*[ ][ ]*)*/
```

NOTE: The asterisk is available in all regexp implementations.

Interpretation issues

As mentioned earlier, in the section called ” passing regexps” , there is more to writing a regexp than just getting it right. You also have to make sure that the tool reading the regexp gets the regexp passed to it exactly as intended.

The problem here, as always, is that different tools handle the reading of regexps differently.

There are some general rules though:

Escape any characters that are special to the tool reading the regexp.

If you intend a character to be literal, it must be escaped in the regexp, if the regexp is being parsed as a string before being read as a regexp, it will need to be escaped twice.

Methods used for escaping are typically quotes (both types) and backslash, though putting characters in a class set ([ ]) removes most meanings.

When passing a regexp to a command line tool, we must beware of any characters that the shell considers special (this varies by shell). For example:

```
grep #  script.pl
```

Causes a grep usage message, rather than printing lines with a # on them. We needed to escape the # symbol in some way.

Generally speaking, if you put a regexp in single quotes on the command line, you are safe. (use double quotes in DOS shells, and

beware the ! in csh). You will be experimenting with this in the lab exercise.

Interpretation issues

In programming languages, there are two major categories of how regexps are handled: those that parse a regexp as an interpreted string first, and those that don’t.

You must know the tool you are using (we will explore these issues in a later chapter).

For example, emacs, tcl, and python first parse regexps as strings, thus a \t would become a tab character due to string parsing, not regexp capabiities.

Therefore, if a regexp contained [\t\n], the \t would become a tab character before the regexp got to see it. Not a big deal.

But, there are other escapes in regexps that would be interpreted wrong, like \b. It would need to be written \\b so that the string parsing passed the desired \b to the regexp. In a string, \b often becomes a backspace character.

Even a * would become *, so it would have to be written *.

In Perl, regexps are not parsed as strings when supplied directly to the match operator, or if placed in single quoted strings.

Basic Regular Expression Example

Suppose that we had a text file that was read by another program. This program had certain rules about which lines could have optional leading white space, and when it is required.

To make checking this file out easier, we want to print just the lines that have leading white space. In the example below, datafile contains the text.

Here are several ways to do this, using a few different tools:

```
grep '^[   ]' datafile      # (space and tab in [ ])
sed -n '/^[      ]/p' datafile    # (space and tab in [ ])
perl -n -e 'print if /^[ \t]/;' datafile
perl -n -e 'print if /^\s/;' datafile
awk '{if (/^[ \t]/)  print }' datafile
```

Note that perl allows the \s abbreviation for all white space, and the \t tab character shorthand, but grep and sed do not. awk allows the \t, but not \s.

Does the second perl example do what we asked for?

Some special notes

Even within the basic regular expression characters, there are some major differences in tools.

As previously mentioned, not all tools support the BRE expressions of { , } and ( ).

More correctly, some tools do not support them, and others require them to be escaped in order to have meaning. Consider this command line usage of grep, note that grep prints lines that match, so when used from the command line, you enter a line of text, after pressing return, if it matched, the line will be echo'd back. Lines returned by grep are underlined below (for clarity):

```
$ grep '(error)'
error
(error)
(error)
```

This means the parentheses were taken as literal characters, but:

```
$ grep '\(error\)'
error
error
```

In this case, they were taken as grouping. This is also true for the interval operator:

```
$ grep '[0-9]\{2\}'
3
34
34
```

So in order to use grouping with the interval quantifier, we need:

```
$ grep '\([A-Z][0-9]\)\{2\}'
A3
```

A3B7
A3B7

Before thinking that a tool does not support a metacharacter or metasequence, try escaping the characters as above (or look at the charts we will be introducing later in the class).

For now, remember that grep, emacs, and vi all need grouping escaped.

Testing Regular Expressions

When testing regexps, it often becomes a trial and repeat scenario.

If an expression does not match, you can only try again. When an expression matches more (or more often) than you wanted it to, it is often a great time saver to see exactly what it did match.

Here are two ways to do this:

Using sed to substitute what was matched for something else:

```
sed 's/regexp/some-text/'
```

It is often helpful to change all occurences on each line with:

```
sed 's/regexp/some-text/g'
```

For example, if we wanted to match a set of digits, and used /[0-9]*/ for the regular expression, it would have a match on all lines in a file, even those without digits. To test it, we might try a substitution. Here is the sample input file:

```
testdata
testdata 1
123 testing 456
```

This sed line:

```
sed 's/[0-9]*/X/'
```

would change the above test input file to:

```
Xtestdata
Xtestdata 1
X testing 456
```

And a global match (sed 's/[0-9]*/X/g') would change it to:

XtXeXsXtXdXaXtXaX
XtXeXsXtXdXaXtXaX X
X XtXeXsXtXiXnXgX X

Testing Regular Expressions

Another way to test a regexp is to use perl to show exactly what was matched. perl saves some side effects of a match into special variables that we can use later for printing.

Taking advantage of this side effect allows us to do some convenient regexp testing. The program below is in your lab files directory for use during lab exercises.

It is called: testregexp

We will not worry about exactly how the program works for now, just use it as a tool when needed during lab.

```
#!/usr/local/bin/perl
while (<>){      # Get a line of input
  if (/[0-9]*/){ # regexp test, edit this regexp
as needed
    print;       # print the line if it matched
    print(' ' x length($') . '^' x length($&));#
show match
    if (length($&) == 0 ) {
      print " Zero length match" ;
    }
    print " \n" ;  # Add a newline
} }              # go back for the next line of
input
```

Above we have the same regexp as used on the previous page, below is some sample input to the program:

```
123 test
test 123
12 34 56
```

And the resultant output:

```
123 test
^^^
test 123
```

```
  Zero length match
  12 34 56
  ^^
```

Note that we are only showing the first match on each line. Input lines that have no match will not be printed.

Extensions

Extended Regular Expression Meta-charactrers

The POSIX specification for ERE's has the following additional quantifiers (with meanings listed):

```
?     Zero or one, same meaning as {0,1}
+     One or more, same meaning as {1,}
```

It also adds the alternation operator, which uses the following symbol:

```
|     Alternation
```

Alternation is very useful to supply a list of possible matches:

```
/error|Error|Warning/
```

As mentioned, the real power of regexps comes from the interactions of the many metacharacters.

What lines of a file do you suppose this would print:

```
egrep 'error|warning: [0-9]+'
```

There is precedence in metacharacters, that is, which characters are interpreted first can change the meaning of an expression. The alternation character has very low precedence, meaning it is interpreted later than most characters:

```
$ grep -E '^error|warning'
error
error
was an error
was a warning
was a warning
```

Note that the search was for " error at the beginning of a line, or, warning anywhere" .

Extended Regular Expression Meta-charactrers

Using grouping, we can make this work as desired:

```
$ grep -E '^(error|warning)'
```

Here is a more complex example:

```
/(error|warning):? [0-9]+/
```

What does that read in english?

We will address precedence in detail later.

There are often multiple ways to write the same thing:

```
/[eE]rror/      /(e|E)ror/
```

Note that alternation and character classes are NOT the same, the character class will never match more, or less than one character.

When matching only one character, it is best to use character classes, when you want to match any one of multiple character strings, you must use alternation (if the tool supports it).

Remember that DFA engines match the longest match (greedy):

```
% awk '{gsub(/(a|abba)/," X" );print}'
abba
X
```

But NFA engines do not (though they should under POSIX):

```
% perl -pe 's/(a|abba)/X/g;'
abba
Xbba
```

More Anchors

Most of the more powerful commands have some additional anchors, usually with meanings that make it easier to define ” words” of text.

If we needed an expression that: *” the string error only as a word” ,* we would need a way to say that the *e* must be preceeded by a beginning of word location, and the last *r* must be followed by an end of word location.

Remember that anchors match a position only, they never match any characters (they are a zero length required match).

The characters available varies by tools, but are typically either:

Vi, newer egrep, vi, emacs:

```
\<    Beginning of word location
\>    End of word location
```

or:

Perl, JavaScript, Tcl, emacs:

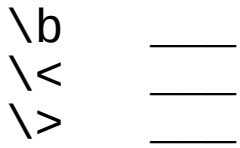
```
\b    Boundary between word, and non-word
characters

\B    No boundary at this location
```

The meanings of these two anchor types are almost identical, except one differentiates between beginning and end, the other is a generic ” word edge” anchor.

These anchors mark the places where there is a change from alphanumerics, to non-alphanumerics.

For example, how many of these anchor locations are there in the string?:



wilbur:Hy2K5ij:124.23 ” there was a” ;:(CELL 4)

More Anchors

Taking an example from the last chapter, when looking for error, we also matched terror. We could try to get just the word error with something like this:

```
/(^|[ ])error([ ]|$)/
```

Which would catch only two of the following, so we need to do better:

```
there was an error
error was there
there was an error?
```

Using the word anchors, we could use either of the below to catch all three lines above (and most other situations of the word error):

```
/\<error\>/      /\berror\b/
```

The regexp on the left would work in vi. In Perl we use the one on the right.

Most tools definitions of ”word” begin and end with a transition from [a-zA-Z0-9] characters to [^a-zA-Z0-9], or the end/beginning of the string/line (remember the character sets for word, and non-word can be abbreviated as \w and \W in some tools).

Note that some tools also include the underscore as a member of the set of word characters (perl and some versions of awk and sed).

Tools using the Perl regexp engine also take anchors like:

```
\A    beginning of string
\Z    end of string
```

\G   location of previous match (or position in string)

Precedence

The order of precedence for these metacharacters is as follows, listed from high to low:

```
( )          parenthesis
[ ]  .             single character matches
* + ? {n,m}  quantifiers
^ $ \b \< \>       anchors
abc          literal concatenation
|            alternation
```

Read precedence charts using the axiom of ” lower operators can be applied to higher precedence operators” . For example, quantifiers can be applied to single character matches, or parenthesis (sub expressions in parenthesis):

```
/” .+” /           # Match one or more characters
in quotes
/(\d+\s+){2,}/ # Match 2 or more ” sets”  of digits
followed
             # by whitespace
```

Below we use parenthesis to override default precedence:

```
/^a|b/           # Match a at beginning, or b
anywhere
/^(a|b)/    # Match a or b at beginning
/file a|b/ # Match ” file a”  or b
/file (a|b)/     # Match ” file a”  or ” file b”
/file [ab]/# Match ” file a”  or ” file b”
```

Backreferences

As mentioned, parenthesis can be used for grouping.

The purpose of grouping can be either to change precedence:

```
/^(error|warning)/
```

or to control the scope of quantifiers:

```
/error( code)?: \d+/
```

Note that in BRE tools, the parenthesis must often be escaped (for example in most grep, vi and emacs versions):

```
/^\(error|warning\):? [0-9][0-9]*/
```

Another purpose of parenthesis is to force the regexp engine to ” remember” what was matched by the sub-expression in parenthesis. This is called backreferencing or tagging because the ” memorized” text can be referred to later in the regexp.

Most tools use an escaped number to recall backreferenced text, the number being which set of parens, counted from the left. (\1 \2 etc..)

Backreferences

Backreferencing can be useful for finding repeating sequences of text:

```
/\b([a-z]+)[ ]+\1/
```

This reads ” a word boundary, followed by a lower case word, one or more spaces, and the same word again (the boundary making it a word)” . It would match the following:

```
and the the cat
she saw that that was correct
```

Due to the boundaries, it would not match in the following data:

```
the thesis was
blithe the move was
```

It can also be useful for data validation:

```
/\w+(:| )\d+\1\w+/
```

Specific data cases:

```
/x=(\d+);y=\1;/
```

What is a ” valid” string (is matched by the expression above)?

What invalid strings are we checking for in the above?

Amusingly, grep, emacs, perl, tcl, and vi can perform back references, but egrep cannot.

Some tools will only remember up to 9, some tools will remember as many as you care to spend the cpu time on.

Some tools allow you to act on, or return the ’tagged’ matches separately.

Addressing type matches

It is often important to write explicit regexps when you are performing a binary match. For example, /May/ would match in both lines below:

```
Those five days in May
Maybe he can catch a ball
```

When performing addressing matches, it is always important to write explicit regexps:

- It is easy to match what you want, it is hard to match only what you want, and exactly the scope of what you want.

An addressing match allows you to act on exactly the characters matched by the regexp. In some cases it allows you to act on all matches in a string/line.

In various tools, using various constructs, the matched substring:

- Might be returned by the match
- Might have its character location/length returned
- Might be the target of a text operators result (substitute operations)

In some tools the matched substring is assigned into a special variable for later access. We call this a side-effect of the match. Some tools return only a true/false (binary) from the match, then make all the information above available through special variables, or methods.

When performing binary match operations, say to print all matching lines:

```
grep -E  '/(error|warning):? [0-9]+/'  logfile
```

The + in the regexp was not needed. If there was one digit, print the line, it does not matter that there were more digits following, as we are acting on the entire line.

In the case of an addressing match, we would need the +, as we want all the digits to be matched so we can act on the entire matched substring, but not necessarily the entire string searched.

Regexp Substitutes

A common use for addressing matches is in a substitute operation. The operator used to invoke a substitute varies by command, but the simplest syntax, and most common construction is:

```
s/regexp/replacement-string/flag-options
```

This s operator is found in commands like vi, ed, ex, perl , and sed.

Note that the replacement string allows various metacharacters, depending on the command. Typically the replacement string is an interpreted string, and thus can contain variables, character abbreviations (\t \n ...) and other command specific string escapes, such as perls \u \U \l \L \E \Q operators.

The tools listed above all allow backreferences to be used in the replacement string, but perl uses *$n* instead of *\n* for backreferences.

All these tools also allow the use of $& in the replacement string to print the entire substring matched by the regexp.

The flag options can modify either the regexp, or the replacement string. For example:

In sed the following flags are valid:

```
g    global, perform all substitutions in a
string
n    1 to 512 meaning which match to substitute
w    write string after substitution to a file
```

In perl the following flags are valid:

```
g    global, perform all substitutions in a
string
i    make the regexp case insensitive
```

m make the regexp act on multi-line string
e evaluate replacement string as an expression
there are also several others to be discussed later

In ex and vi, the following flags are valid:

g global, perform all substitutions in a string
p print lines with substitutes
c prompt for confirmation before each substitute

Regexp Substitutes

Another form of regexp based substitute is found in programming languages, which typically use function calls.

For example, awk and python use:

```
sub(regexp,replacement,search-string)
gsub(regexp,replacement,search-string)
```

JavaScript uses:

```
search-string.replace(regexp)
```

Tcl uses:

```
regsub [options] regexp target replacement
destination
```

Emacs uses the following interactive commands:

```
replace-regexp
query-replace-regexp
```

Match Returns

Most command line tools (like ex, vi, sed, grep, egrep) do not allow you to act directly on the text matched by a regexp, except by using a substitute operation.

Most programming tools do allow you to access the the matched text in some way. It might be assigned into a special variable, returned from a special method or function, or it might be returned by the match operation itself. Documentation of the regexp utility of that language should list which of these are available, and how to use them.

Perl provides multiple ways to access the exact substring of a match, so we will use it as an example.

Special variables:

In perl, there are several variables set after every match operation:

```
$string = 'before the match after';
$string =~ /th(.*)ch/;
print " $` \n$& \n$' \n$1 \n" ;
```

The three variables named ` & and ' are components of the searched string ($string in example above) representing the match (&) the characters before the match (`), and the characters after the matched substring ('). The sample code above results in the output below:

```
before
the match
 after
e mat
```

Note that the tagged section of the match (in parenthesis) gets moved into the variable $1, with subsequent tagged sections going into $2, $3, etc...

These variables allow for many different manipulations. For example, we can use the length of the $` variable to locate the position of the matched string for substring manipulation:

```
$string='...variable=ABC;known=123';
$string =~ /known/;
$needed=substr($string,length($`)-4,3);
print(" $needed\n"  , length($`) , " \n" );
```

Results in:

```
ABC
16
```

Another method to access the same data as above (using a backreference):

```
$string='...variable=ABC;known=123';
$string =~ /(...);known/;   # tag the characters
print " $1\n" ;             # use special variable
for access
```

Other tools have similar capabilities, for example, tcl provides a Match variable that gives begin and end locations of the match (when -indices is used).

Returned substring

Perl has a way to let you return backreferenced substrings from the match operator.

```
$string = 'before the match after';
($match) = ($string =~ /th(.*)ch/);# $match gets
" e mat"
print " $& \n$1 \n$match\n" ;     # $1 is also  " e
mat"
```

Beware that backrefernces are only returned when the match is used in an array context (a very Perlish thing):

```
$string = 'once upon a Jul 1 1997';
$match = ($string =~ /((\w{3}) (\d{1,2})
(\d{2,4}))/);
# $match gets " 1"  since match was " successful"
@date = ($string =~ /((\w{3}) (\d{1,2})
(\d{2,4}))/);
# Date array gets list of " Jul 1 1997" , " Jul" ,
" 1" , " 1997"
```

Perl also allows you to perform a global match, which can return all matches at once (in array context):

```
$string = " 1247 dsec 14 brof 237" ;
@allnums=($string=~/(\d+)/g);     # array context
print " @allnums\n" ;        # prints 1247 14  237
```

How OO does it

In the languages that support regexps in an Object Oriented manner, a regexp is typically ” compiled” , or created as an Object, then ” applied” to strings. In these languages, a variable within the object is often set to the matched substring. If not, a method is provided that returns the matched string. Here is an example from JavaScript:

```
re=/(\w+)=(\w+)/;
document.links[1]=” http://www.famece.com/cgi-
bin/request.cgi?SUBJ=UNIX&level=det” ;
data=document.links[1].match(re);
re=/(\w+)=(\w+)/g;
data2=document.links[1].match(re);
```

After the above, the following are true:

```
data[0] = ’SUBJ=UNX’   data2[0] = ’SUBJ=UNX’
data[1] = ’SUBJ’       data2[1] = ’level=det’
data[2] = ’UNIX’
```

We could have written this in a more OO manner using the following:

```
re = new RegExp(” (\w+)=(\w+)” , ” ig” );
document.links[1]=” http://www.famece.com/cgi-
bin/request.cgi?SUBJ=UNIX&level=det” ;
data = re.exec(document.links[1]);
```

Performance

The book ” Mastering Regular Expressions” by Jeffrey E.F. Friedl (O’Reilly) does a great job with covering these issues (chapter 5).

An (apparently) simple change from:

```
/($DOUBLE|$SINGLE)|$COMMENT|$COMMENT2/
```

to

```
/($DOUBLE|$SINGLE|$OTHER)|$COMMENT|$COMMENT2/
```

reduced the run time against a 60K string from 37 seconds to 2.9 seconds by adding in a possible match for things that would prevent matching one of the comment alternatives.