

CS306: Introduction to Perl

Pattern Matching and Regular Expressions

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

Slide 1

What are Regular Expressions?

- A regular expression is a template which defines a pattern of text which you then compare to other text looking for a match. The result is either “match” or “doesn't match”.
- Regular expression syntax is its own mini-language which you use to write the mini programs, or templates.

Slide 2

What are Regular Expressions? Con't

- Not unique to perl - UNIX utilities like the shell, sed, awk, egrep and other programming languages also support regular expressions
- Perl has the best implementation of regular expressions

Slide 3

Simple Patterns

- ```
my $string = "Hello world!";
if ($string =~ /ello/) {
 print "Match!\n";
}
```
- The `/ello/` is the regular expression. It looks for “ello” **anywhere** in a string.
- The `//` act just like “`“`”, so all escapes like `\t` work. For example, `/fran\tinstructor/` #looks for a tab

Slide 4

## Pattern Behavior

- A pattern by default tries to match anywhere in the string.
- You can think of it as starting at the front of the string and “sliding” along the string until it matches or reaches the end.
- **The Leftmost Longest Rule.** Given a choice, a pattern will make the leftmost, longest match it can. This is very important. (Hint, Hint).

Slide 5

## Metacharacters

- Metacharacters allow us to be less literal in our patterns
- `.` - anything except `\n`
- `*` - match preceding thing zero or more times
- `+` - match preceding thing one or more times
- `?` - match preceding thing zero or one times
- The `\` turns metacharacters back into regular ones

Slide 6

## Metacharacter examples

- `/Fran.Fabrizio/`

Slide 7

## Metacharacter examples

- `/Fran.Fabrizio/`
  - `FranLFabrizio`
  - `Mr. Fran Fabrizio`
  - `FranFabrizio`
  - `Fran\Fabrizio`
  - `Francis Fabrizio`

Slide 8

## Metacharacter examples

- `/Fran.Fabrizio/`
  - `FranLFabrizio`
  - `Mr. Fran Fabrizio`
  - `FranFabrizio` # No match
  - `Fran\Fabrizio`
  - `Francis Fabrizio` # No match

Slide 9

## Metacharacter Examples

- `/Hello*World/`

Slide 10

## Metacharacter Examples

- `/Hello *World/`
  - `Hello World Leaders`
  - `Hello` `World`
  - `Helloo World`
  - `HelloWorld`

Slide 11

## Metacharacter Examples

- `/Hello *World/`
  - `Hello World Leaders`
  - `Hello` `World`
  - `Helloo World` # No match
  - `HelloWorld`

Slide 12

## Metacharacter Examples

- `/books?/`
  - Matches book and books
- `/Echo+/`
  - Matches Echo, Echoo, Echooo, Echoooo etc...

Slide 13

## Metacharacter Combinations

- `/hey.*there/`
  - “Hey, followed by zero or more anything characters, followed by there”
  - Which match?
    - hey there
    - hey Joe, how's the weather there in Phoenix?
    - they are late again, and therein lies the problem

Slide 14

## Metacharacter Combinations

- `/hey.*there/`
  - “Hey, followed by zero or more anything characters, followed by there”
  - Which match?
    - hey there
    - hey Joe, how's the weather there in Phoenix?
    - they are late again, and therein lies the problem

Slide 15

## Grouping Patterns

- Use `()` to group characters
- `/(fred)+/`
  - One or more “fred”s
  - Matches:
    - fred
    - fredfredfred
    - barneyfredbambam
    - Alfred, pull the Batmobile around to the front please.

Slide 16

## “Or” in Patterns

- Use the pipe | character
- `/Perl|Java|C/`
  - What's the English translation?
- `/Fran( \t)+Fabrizio/`
  - What's the English translation?
- `/Fran( +\t+)Fabrizio`
  - What's the English translation?

Slide 17

## Saying “or” in Patterns

- Use the pipe | character
- `/Perl|Java|C/`
  - “Perl or Java or C”
- `/Fran( \t)+Fabrizio/`
  - “Fran, followed by one or more spaces or tabs in any combination, followed by Fabrizio”
- `/Fran( +\t+)Fabrizio`
  - now it has to be all spaces or all tabs, not mixed

Slide 18

## Character Classes

- A *character class* appears between [ ] and matches any single character in the class.
- `[abcdxyz]` - match any of those seven characters.
- `[a-zA-Z]` - you can use ranges in character classes
- `/0x[0-9A-Fa-f]+/` - match hexadecimal numbers

Slide 19

## Saying “not” in Character Classes

- Use the ^ at the beginning to say “not”. Sometimes that's easier.
- `[^aeiou]` - will match anything BUT lowercase vowels

Slide 20

## Magic Character Class Shortcuts

- Some character classes are so common they have magic shortcuts
- `\d` is short for `[0-9]` (“digit” character)
- `\w` is short for `[0-9a-zA-Z_]` (“word” character)
- `\s` is short for `[\f\t\n\r ]` (“whitespace” character)
- Note that these abbreviations eliminate the `[ ]` too.  
So, `/cs[0-9]+/` becomes `/cs\d+/`

Slide 21

## Everything but the shortcuts...

- `\d` means “any digit character”. `\D` means “anything but a digit character”.
- `\w` means “any word character”. `\W` means “anything but a word character.”
- You see the pattern. `\S` is “any non-whitespace character”.

Slide 22

## Nested Shortcuts

- They work inside of other patterns. So, our hexadecimal pattern can be rewritten as `/0x[\dA-Fa-f]+/`

Slide 23

## Quantifiers

- We've seen some already:
  - \* - zero or more
  - + - one or more
  - ? - zero or one
- Others:
  - `{3}` - exactly 3 times. `\w{3}/`
  - `{3,}` - at least 3 times `/(fred){3,}/`
  - `{,3}` - DOESN'T WORK!

Slide 24

## Anchors

- *Anchors* force the pattern to match at a specific place, rather than sliding down the string looking for a match
- ^ - match at the beginning of a string.
  - This ^ is different than the one inside of character classes, which means “not”
- \$ - match at the end of a string
- \b - match at a word boundary (\B is opposite)

Slide 25

## Anchor Examples

- /^Subject/ - look for a line that begins “Subject”
- /phone home\$/ - look for a line that ends “phone home”.
  - A note here: \$ matches either at the end of a string or at a newline at the end of a string. Therefore, it will match “phone home” and “phone home\n”
- /\s\*\$/ - Who can tell me what this one does?

Slide 26

## More Anchor Examples

- A word boundary is the beginning or end of a group of \w characters
- \bman\b/ - matches “a man apart” but not “marksmanship”, “mailman” or “mantle”
- \Bman\b/ - Any word that ends in “...man”
- Note that “Let's go!” has three “words”... “Let”, “s” and “go”. The ' is not a \w character. So it doesn't quite mean “word” in the English sense

Slide 27

## Take a Breather

- Everything we just talked about was general to regular expressions and pattern matching. Very little of it was Perl-specific
- Now we will look at how to use patterns within perl to do powerful things

Slide 28

## The m// match operator

- This is what we've been doing all along today, with the /pattern/ syntax. We use the match operator m/pattern/ to test a pattern on a string.
- Just like qw//, the m/pattern/ can use other delimiters besides //. For instance, m#pattern# or m!pattern!.
- However, when using the m/pattern/ form we can omit the m, leaving just /pattern/

Slide 29

## m// and \$\_

- ```
while (my $line = <>) {  
    if ($line =~ /^Subject:/) {  
        print "Found the subject line: $line\n";  
    }  
}
```

Slide 30

Option Modifiers

- Case-insensitive matching: m/pattern/i
- Making . match \n: m/pattern/s

Slide 31

When m/pattern/i Is Useful

- ```
while (my $line = <>) {
 if ($line =~ /^received:/i) {
 print "Found a received: header: $line\n";
 }
}
```
- This will match lines that begin with Received:, received:, rEcEiVeD:, etc....

Slide 32

## When m/pattern/s Is Useful

- Say we have this as input:  
The C-USA tournament was won by UAB.\n  
The Blazers beat Memphis in the final.\n
- We want to find any mention of “UAB” followed somewhere by “Blazers”
- /UAB.\*Blazers/ will fail because the . does not match a \n, so they'd have to be on the same line
- /UAB.\*Blazers/s will match the above text

Slide 33

## Combining Modifiers

- /UAB.\*Blazers/is applies both modifiers.
- Input:  
Uab won with a combination of scoring and defense.\n  
Final: BLAZERS 82 Tigers 65\n
- The pattern above will find this match because of the case-insensitivity of i and the newline matching of s

Slide 34

## Storing Patterns in Variables

- ```
my $search = 'cats';  
my $string = "It's raining cats and dogs.";  
if ($string =~ /($search)/) {  
    print "I found what you're looking for.\n";  
}
```

Slide 35

Remembering Matches

- () have previously been used for grouping patterns, like /(fred)+/ is one or more “fred”s
- () are also *memory parentheses*. When you use the (), the regular expression engine remembers the substring that matched that part of the pattern.
- So, if /\d+/ matches one or more digits, /(\d+)/ also matches one or more digits, and remembers which digits caused the match

Slide 36

Memory Example

- `/^Subject: (.*)/` - will match from the beginning of the string “Subject: “ and then zero or more anything and remember those zero or more anything. i.e. it will grab the rest of the line.
- So, “Subject: Pizza Party Friday”, the memory would be “Pizza Party Friday”

Slide 37

Multiple Memories

- `/(\d{3})-(\d{3})-(\d{4})/` - phone number, remembering each piece separately
- `/(\w+) (\d+), (\d+)/` - Month, day, year
 - Jan 6, 2005
 - January 06, 2005

Slide 38

Accessing Memory - Match Variables

- ```
print "Enter phone no. (XXX-XXX-XXXX): ";
chomp(my $number = <STDIN>);
if ($number =~ /(\d{3})-(\d{3})-(\d{4})/) {
 my ($areacode, $prefix, $num) = ($1, $2, $3);
} else {
 print "Phone number invalid.\n";
}
}

```
- \$1, \$2, \$3, etc... are the match variables

Slide 39

## Match Variable Shortcut

- ```
print "Enter phone no. (XXX-XXX-XXXX): ";
chomp(my $pn = <STDIN>);
unless (my ($ac, $pf, $num) = $pn
        =~ /(\d{3})-(\d{3})-(\d{4})/) {
    print "Phone number invalid.\n";
}

```
- In list context, the regular expression returns a list of the memory vars, enabling syntax above.

Slide 40

Persistence of Match Variables

- The match variables get populated on successful match, and stay populated until next -successful-match.
- This implies that you should always check for success when doing a pattern match, otherwise you might not have what you think in \$1, \$2, ...
- So, patterns almost always in if() or while()

Slide 41

Substitution with the s/// operator

- “Search and Replace”
- my \$str = “Today's class is on Wednesday”;
\$str =~ s/Wednesday/Thursday/;

```
my $str = “I'm playing racquetball with Hari
today.\n”;
$str =~ s/with ([A-Z]\w+)/against $1/;
```

Slide 42

Global Search and Replace

- Use the g modifier
- my \$str = “I play racquetball with Hari today and with Vijay tomorrow.\n”;

```
$str =~ s/with ([A-Z]\w+)/against $1/g;
```

```
# Now str is “I play racquetball against Hari
today and against Vijay tomorrow.\n”;
```

Slide 43

A Couple of Notes

- Modifiers i and s also work here like with m//
- Different delimiters too: s#word#newword#
- s/// will also work on \$_ in the absence of the binding operator =~
- A common bug:

```
$str = s/someword/someotherword/g;
```

Slide 44

Using patterns with split()

- We've already seen things like:
my \$str = "The cow jumped over the moon.";
my @words = split //, \$str;
- The // is a pattern. So this also works:
my \$str= "Fee Fi\Fo\nFum";
my @words = split /\s+/, \$str;

Slide 45

Phew, You Made It

- This will all take a while to get used to, it takes practice.
- These are just the basics, patterns can get -extremely- complex. Do not get overwhelmed by the book info, these slides provide all the stuff you need to be productive with patters.
- Practice reading patterns like sentences of any other language.

Slide 46