Programming in Perl

Week five

regular expressions

working with text

Jump to first page

Exercise 4.1

```
#!/usr/bin/perl -w
use strict;

my @table;

while(<DATA>){
    chomp;
    push @table, [split];
}

@table = transpose(@table);

foreach my $row (@table){
    print join("\t", @$row),"\n";
}

sub transpose {
    my @mat = @_;
    my @return;
    for(my $i = 0; $i < @mat; $i++){
        for(my $j = 0; $j < @{$mat[$i]}; $j++){
            $return[$j][$i] = $mat[$i][$j];
        }
    }
    return @return;
}

__DATA__
one   two   three
four  five  six
seven eight nine
```

Regular Expressions

- Review of regular expressions
- Concatenation
  - There is an implicit concatenation of of simpler patterns to make more complex ones

```
/c/ # a simple pattern
/cat/ # three simple patterns concatenated
```

- Alternation
  - Allow a match between two or more patterns

```
/cat|dog|rabbit/
```

Regular Expressions

- Grouping
  - Parentheses allow you to group patterns to create sub-patterns that can be treated a as single unit
  - Parentheses also trigger memory so that if the sub-pattern is matched, the value of the match is saved in $1,$2, etc. (And \1, \2, etc. For substitutions)

```
/(dog|cat)burt/ # match dogburt or catburt with dog or cat in $1
```

- Iteration
  - The star operation (*) allows you to match zero or more of the preceding patterns

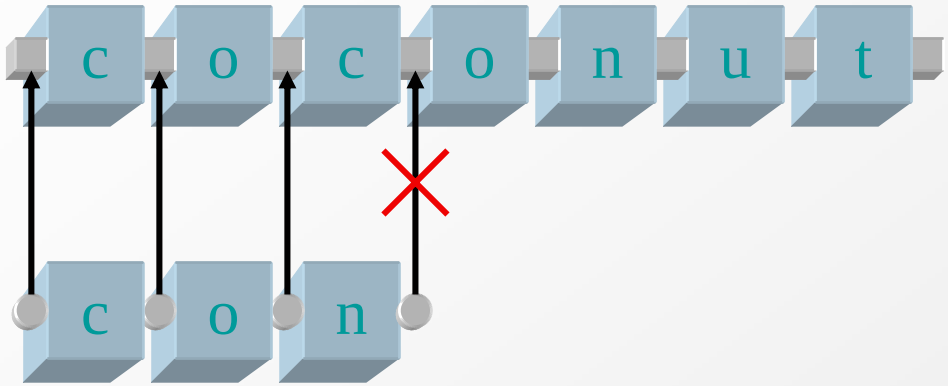
```
/(cat)*/ # match nothing, or cat, or catcat, etc.
```

- Dot
  - The dot operation will match any single character

```
/c.t/ # match cat, cot, cct, c1t, etc.
```

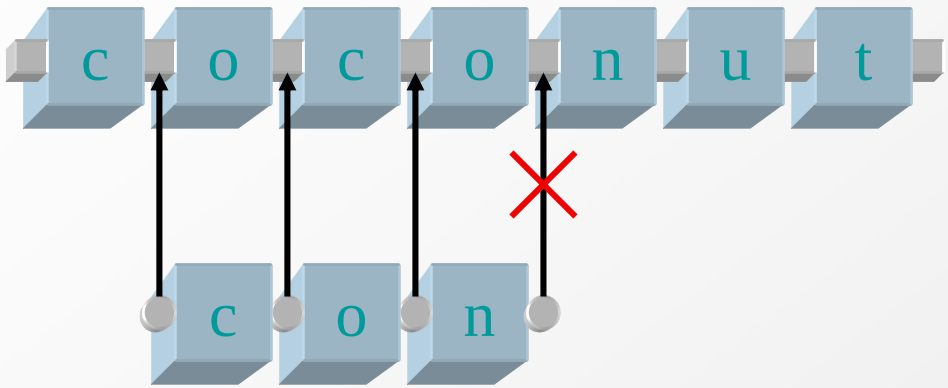
Regular Expressions

- Let's look at the pattern `/con/` working over the string `coconut`



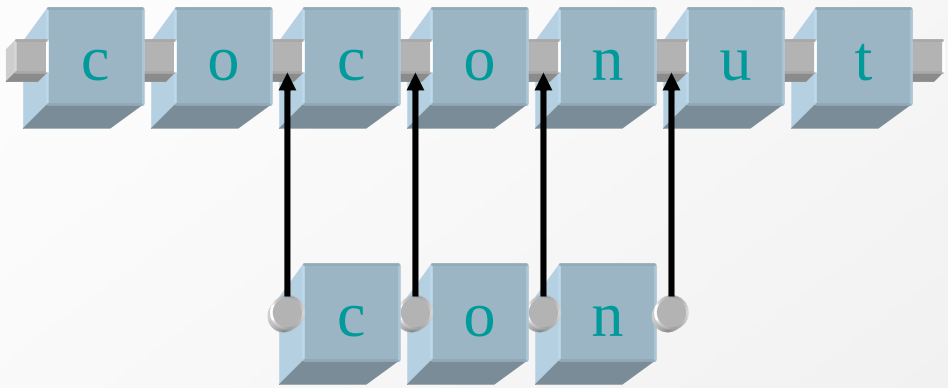
Regular Expressions

- Let's look at the pattern `/con/` working over the string `coconut`



Regular Expressions

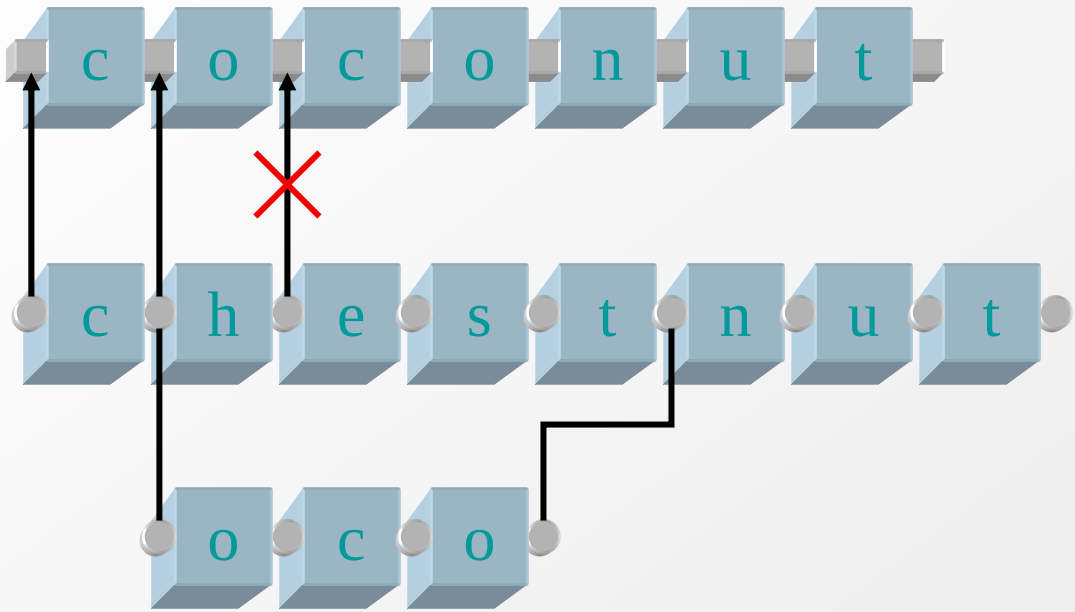
- Let's look at the pattern `/con/` working over the string `coconut`



TRUE

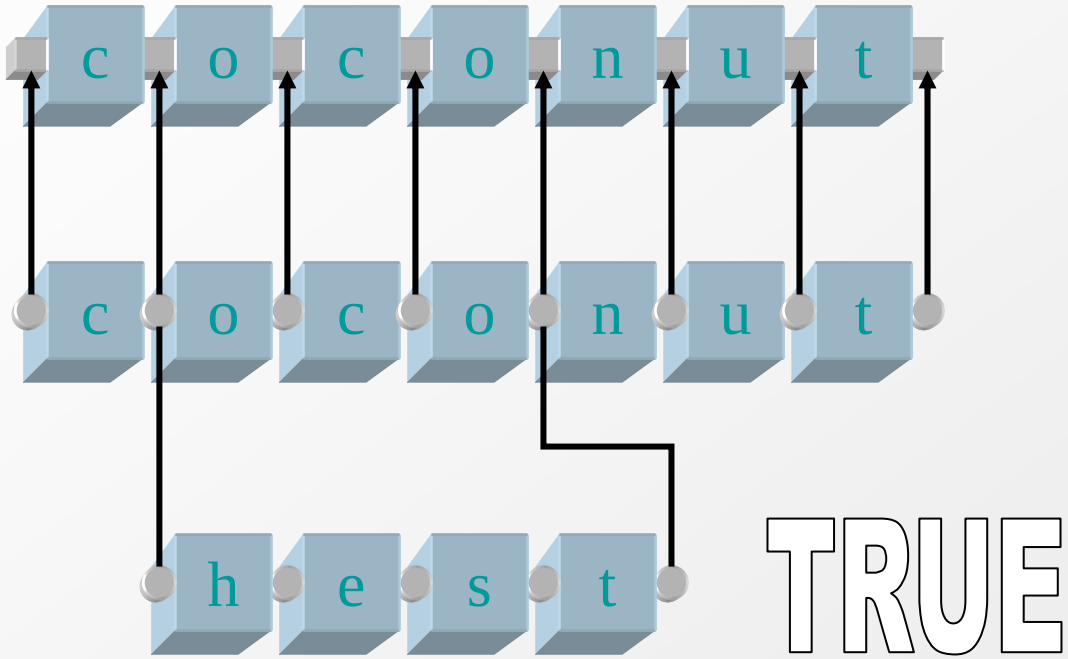
Regular Expressions

- Working with alternation: `/c(hest|oco)nut/`



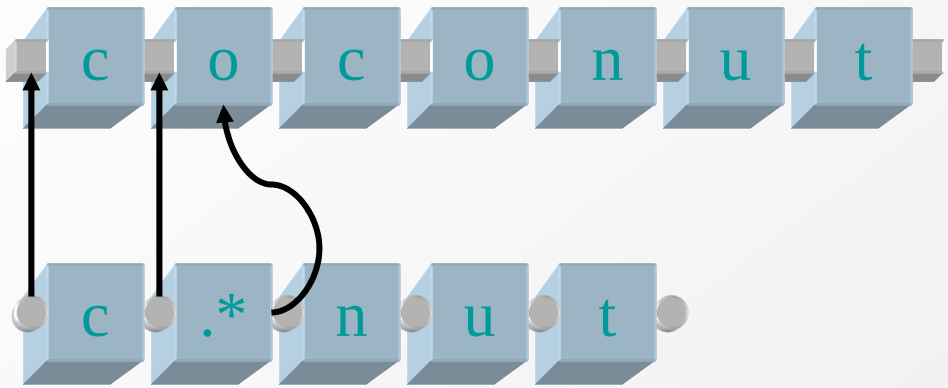
Regular Expressions

- Working with alternation: `/c(hest|oco)nut/`



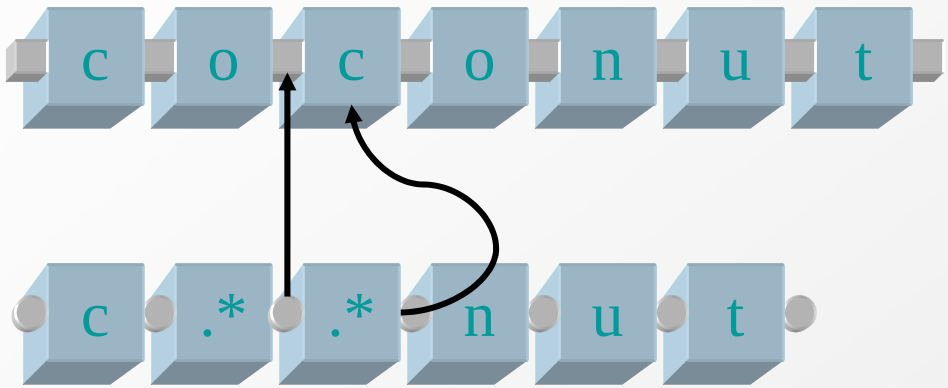
Regular Expressions

- Working with iterations: `/c.*nut/`



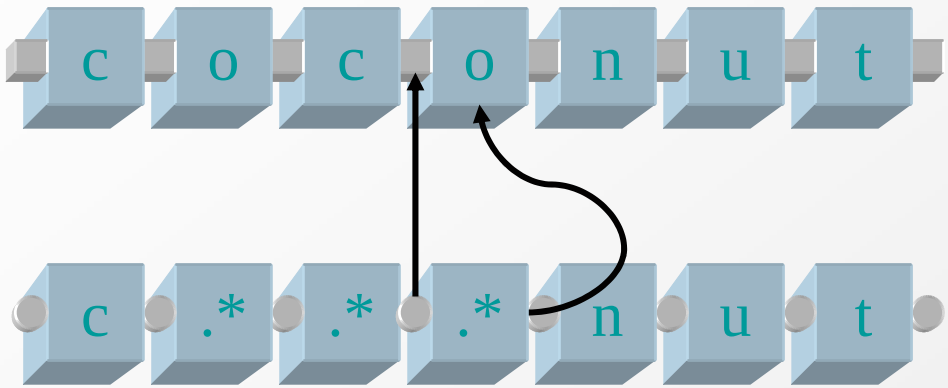
Regular Expressions

- Working with iterations: `/c.*nut/`



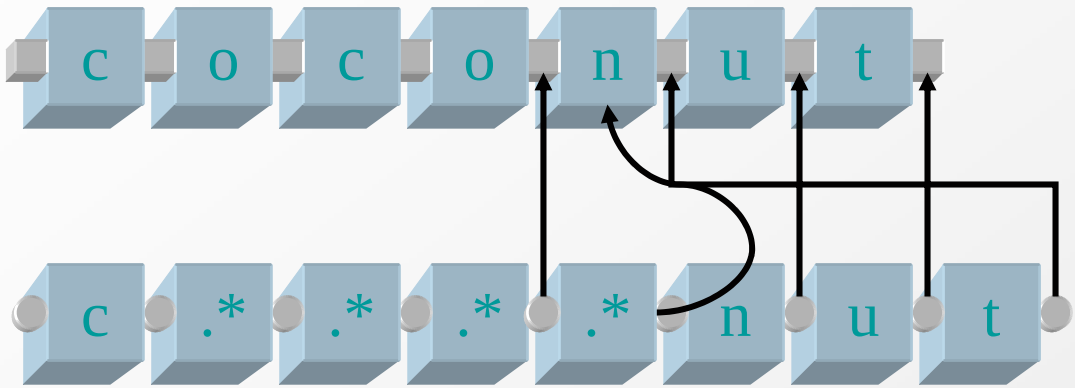
Regular Expressions

- Working with iterations: `/c.*nut/`



Regular Expressions

- Working with iterations: `/c.*nut/`



TRUE

Iterators Are Greedy

- The star operator really works by consuming the longest string it can

```
$_ = ‘A coconut is a nut that only nuts eat.’
/c.*nut/;
print “$&\n”;  # prints ‘coconut is a nut that only nut’
```

- All iteration operators work this way: `*`, `+`, `{m,n}`
- The `?` operator is “zero or one” so even though it’s greedy, it can only match up to one element, so there is no difference
- There are “non-greedy” operators

```
$_ = ‘A coconut is a nut that only nuts eat.”
/n.*?nut/;
print “$&\n”;  # prints ‘coconut’
```

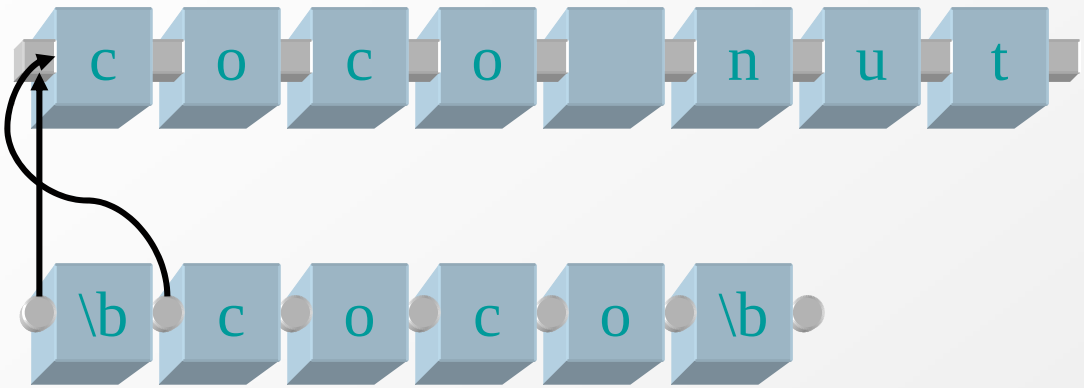
Anchors

- We have already seen two anchors:
  - `^` matches the start of a line
  - `$` matches the end of a line
- An anchor is a zero width assertion, meaning that it does not match a character, but the space between characters
  - `^` matches just before the first character of the string
  - `$` matches just after the last character of the string
- The `\b` anchor matches between a word (`\w`) to non-word (`\W`) character
  - It matches the “boundary” of a word

Working with Word Boundary Anchors

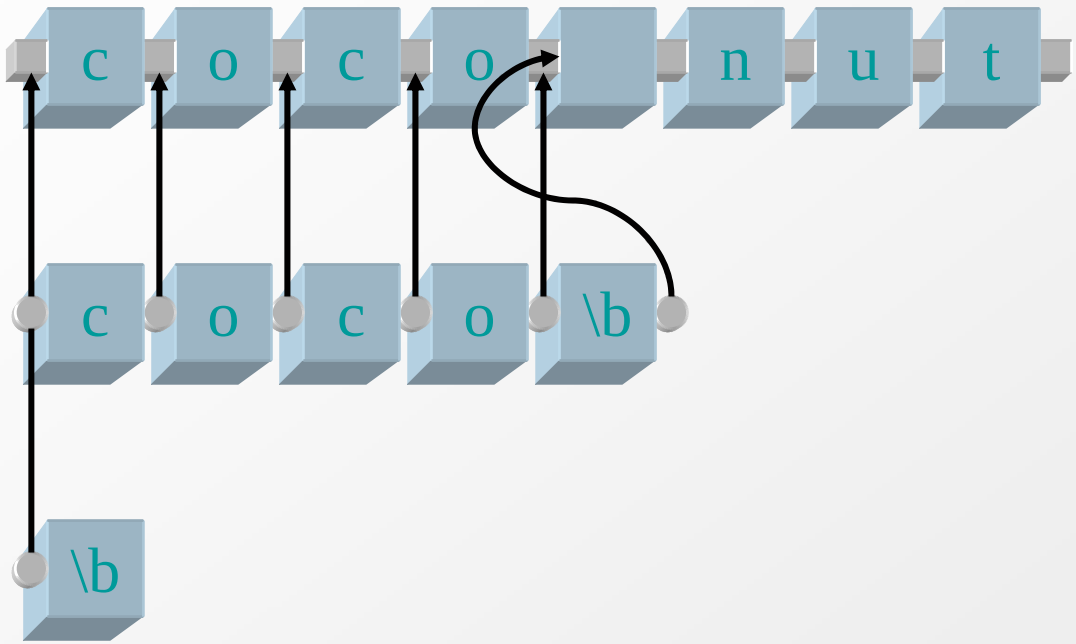
- Let's look at the pattern `/\bcoco\b/` working over the string

```
coco nut
```



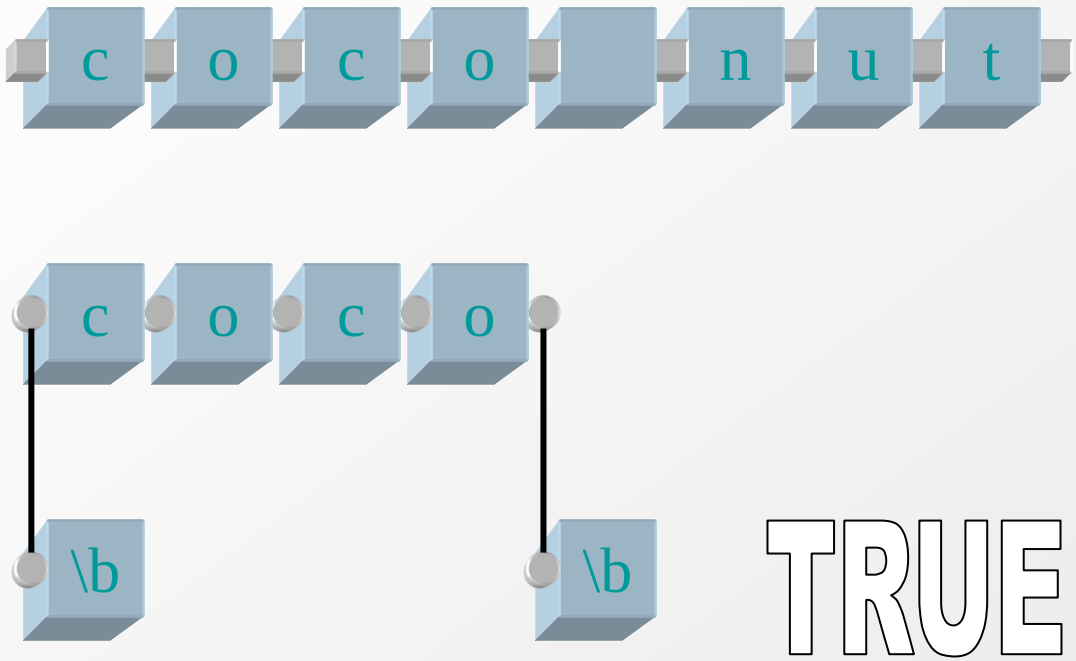
Working with Word Boundary Anchors

- Let's look at the pattern `/\bcoco\b/` over the string `coco nut`



Working with Word Boundary Anchors

- Let's look at the pattern `/\bcoco\b/` over the string `coco nut`



More anchors

- Like the character classes (`\w`, `\s`, etc.) there complement ((`\W`, `\S`, etc.) for the word boundary anchor: `\B`
  - It matches the boundary between two word characters or non-word characters

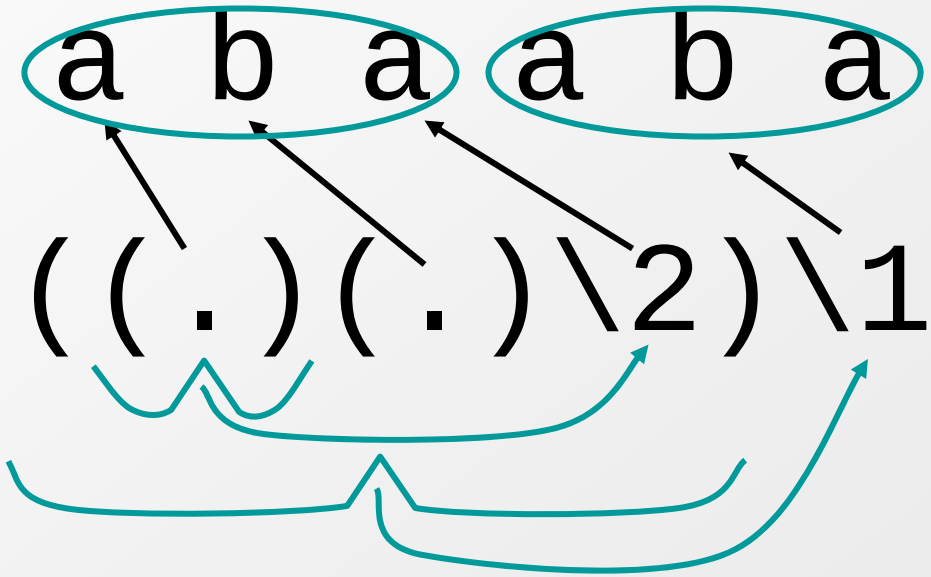
```
$_ = ‘Room B123 is the 4th door on the left’;
/Room.*\d+/; # finds ‘Room B123 is the 4’
/Room.*\B\d+/; # finds ‘Room B123’
```

Regular expression precedence

- Ambiguous patterns can use parentheses for clarification (parentheses count for memory, though.)
  - `a|b|c{3,5}` (is `a` or `b` or 3 to 5 `c`'s, because curlies are higher than bars)
  - `(a|b|c){3,5}` (to get 3 to 5 of any of them)
  - `^a|b|c` (an `a` at the beginning, or `b` or `c` anywhere)
  - `(^a)|b|c` (same thing)
  - `^(a|b|c)` (`a` or `b` or `c` only at the beginning)
- If you don't want to trigger memory, but still need to group, use non-memory form of parentheses:
  `(?:subexpression)`

Parentheses Memory

- Parentheses Memory is triggered when the first opening parentheses is found, working from left to right
  - `((.)(.)\2)\1` (any two chars, followed by first char twice, i.e. abaaba)



Matching Modifiers

- Append a “`i`” to ignore case

```
print "do you like Perl?";
$_ = <STDIN>;
if (/^y/i) { # begins with y
    print "good answer!\n";
}
```

Multi-line matches

- By default, Perl treats a string as a single line and a `$` will skip over embedded `\n`'s (but "`.`" will not match one)
- The "`m`" modifier put the regex engine into "multi-line mode" where the "`^`" and "`$`" will match the start and end of an embedded line

```
$_= "This is\na string\nwith embedded newlines";
/^a string$/  # Will not match
/^a string$/m # Will match
```

- You can use the `\A` and `\Z` to always match the start end end of the string

Single Line Mode

- The “`s`” modifier starts the “single line mode”. It changes the definition of the dot operator so that it will match a “`\n`”
- Mostly, you will see the “`m`” and “`s`” operators used together
- You should only use the “`m`” and “`s`” modifiers if you know that you will have string that will contain multi-lines.

Extended Mode

- Extended Mode, using the “x” modifier, will make the regular expression engine ignore any unescaped white space and recognizes the “#” as the start of a comment

```
m/
    (
        (.)         # Remember a character in \2
        (.)         # Remember the next character in \3
        \2          # Match the the first character again
    )               # Remember the that in \1
    \1              # and match it again
/x;
```

Global Mode

- The “g” modifier puts you into “global mode” where all the matches in a string are found
- For the match operator (m//g), in scalar context, the operator will stop after it finds a match, and start up at that same spot the next time the match is called

```
$_ = ‘12 and 3.1415926 and 130.2’;
my $count = 0;
while (/\d+(?:\.\d+)?/g) {
    $count++;
}
```

- Finds 12 the first time, 3.1415926 the second, and 130.2 last

Global Mode

- In LIST context, and there are parentheses triggering memory, all the memorized matches in a list like
  `($1, $2, $3)`
- The Perl function `pos()` will return the character position after the match in the string starting
- The position is reset if the match fails, or a new regular expression is used
- You can use the “`c`” modifier with “`g`” to stop this feature

Global Mode

- There is another anchor that you can use to mark the position in a pattern: `\G`

```
$_ = ‘foobar bar’;
/foobar/gc          # match foobar
/\Gbar/gc           # match the bar after foobar
```

Global Mode

- The Perl substitution operator can use all of the modifiers as well
- `s///` always returns the number of substitution made, no matter the context
- There is one modifier that is unique to substitution: “e” for execute mode
- It will execute the replacement string as a Perl program

Execute Mode

```
$_ = ‘It is 2.7 miles from here to there.’;
s/(\d+(?:\.\d+)?)\s*miles/miles_to_km($1) . kilometers’/e;
print;

sub miles_to_km {
    my $miles = shift;
    my $km = sprintf(“%.2f”, $miles / 0.6);
    return $km;
}
```

Finding a substring

- Use the `index()` operator to find a sub-string within another string.

```
$where = index($big, $small); # find $small within $big
```

- The returned index is a zero-origin number, or -1 if not found

```
$stuff = “Hello world!”;
$where = index($stuff,”wor”); # $where gets 6
```

- To find a later copy, use the optional third parameter, which gives a minimum value for index.

```
$where1 = index($stuff,”l”); # $where1 gets 2
$where2 = index($stuff,”l”,$where1+1); # $where2 gets 3
$where3 = index($stuff,”l”,$where2+1); # $where3 gets 9
$where4 = index($stuff,”l”,$where3+1); # $where4 gets -1
```

- Search from right to left using `rindex()`:

```
$last = rindex(“/etc/passwd”,”/”); # $last gets 4
```

Extracting and replacing a substring

- The `substr()` operator examines part of a string

```
$part = substr($string, $initial_position, $length);
print ‘j’,substr(“Hello, world!”,1,4); # prints “jello”
```

- The initial position can be negative, counting from the end of the string

```
$x = substr(“some very long string”,-3,2); # $x gets “in”
```

- In no length given, the position to the end of the string is returned

```
$long = “some very very long string”;
$right = substr($long,index($long,”l”))
```

- The selection portion of the string can be changed if the string is a variable

```
$str = “Hello, world!”; substr($str,0,5) = “Goodbye”;
```

- It’s OK to make the string longer or short this way, but it’s more efficient if it stays the same

Exercises Week Five

1. Write a regex that will match a target string only if it contains only an integer number (all digits).
2. Write a regex that will match a string that contains an integer or decimal number.
3. Write a regex that will match a string if it contains a positive or negative number including numbers in scientific format (i.e., `3e12`).
4. Write a function that prints out a summary of the frequencies of each vowel in a string. For example, if passed the string `This is the winter of our discontent`, it would print:

```
a  0
e  2
i  4
o  3
u  1
```