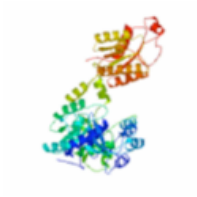
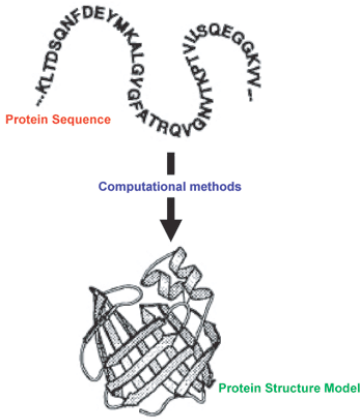
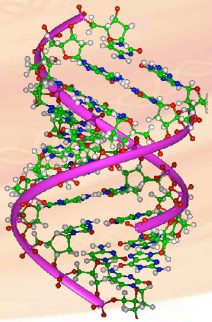




Bioinformatics Computational Methods 1 - BIOL 6308



October 1st 2013

http://155.33.203.128/cleslin/home/teaching6308F2013.php

Last Time

- vim tips
- Additional Linux Tips
- Default Variable
- Writing to an OUTFILE
- Sorting Arrays and Hashes
- Defensive Programming
- Line Separator
- File Tests
- Subroutines
  - Arguments (parameters)
  - Return Values
  - Context
  - Understanding Scope
  - Call By Value
  - Call By *Reference*
- References
- Questions?

Pattern matching in UNIX

adopted from Stewart Weiss

Problem Ahead…

- Much to your disadvantage, there are two different forms of patterns in UNIX
  - Ones used when representing file names
  - Another used by commands, such as `grep, sed, awk,` and `vi`
- **You must remember the two types of patterns are different**

Remember: File Glob

- In card games, a wildcard is a playing card that can be used as if it were any other card, such as the Joker
- Computer science has borrowed the idea of a wildcard, and taken it several steps further
- Patterns are not regular expressions - they look like them
- Patterns are called **file globs**
- The name "glob" comes from the name of the original UNIX program that expanded the pattern and matched filenames to it

File Glob Rules

- **Rule 1:**
  - a character always matches itself, except for the wildcards. So a matches 'a' and 'b' matches 'b' and so on
- **Rule 2:**
  - A sequence of characters that does not contain any wildcards matches itself, so hello matches 'hello'
- **Rule 3:**
  - **? matches exactly one character, including blanks and wildcard characters. It matches itself as well.**
  - **So ?? matches any filename with exactly two characters in it, such as 'aa' or 'bb' or 'b?' or '_t'. ? is an example of a wildcard**
- **Rule 4:**
  - **? will not match a '.' when it is the first character in the file name**

File Glob: Character Classes

- [list-of-characters] matches any single character in the list
- The list-of-characters can be specified as a range, which is of the form c-d, where c and d are characters and no space is between

Examples:

`[a-zA-Z]`	matches any single letter
`[0-9]`	matches any single digit
`[a-zA-Z0-9]`	matches any letter or digit
`[[]`	matches left bracket '['
`[-a]`	matches 'a' or '-'
`[]]`	matches ']'

File Glob: Character Class Complements

- Putting a ! as the first character in the list forms the complement list
- [!list-of-characters] matches any character NOT in the list

Examples:

`[!  ]`	matches any character that is not a space
`[!a-zA-Z]`	matches any character except letters

File Glob Wildcards: *

- '`*`' matches 0 or more characters
- Examples:
  - `s*` matches any filename starting with `s`
  - `bin*` matches any filename starting with `bin`
  - `t*c` matches any filename starting with `t` and ending with `c`
- But
- `*` matches all filenames except those starting with '.'
- `.*` matches only filenames starting with '`.`'

File Glob Examples

`hwk[0-9].???`

- Matches all files whose names start with `hwk` and are followed by a single digit then a '`.`' then 3 characters, such as `hwk1.bak`

`w*.[a-z][a-z][a-z]`

- Matches all filenames starting with '`w`' having a '`.`'somewhere after `w` after which are 3 lowercase letters

`[!a-zA-Z]*_*`

- Matches all files whose names start with a character other than a letter, and have an underscore somewhere in them

Regular Expressions in UNIX

Regular Expressions in Filters

- Let's introduce regular expressions
- These are a special kind of pattern used by `grep`, its two cousins, `egrep` and `fgrep`, as well as `vi`, `sed`, and `awk`
- They are:
  - Used within the `vi` editor for searching and replacement of strings
  - Also partly the foundation of pattern-matching in Perl
  - `grep` is a tool I use all the time
    - Search:
      - output files
      - directory listings
- Therefore, they are of fundamental importance in using UNIX efficiently

Regular Expressions in Filters

- With these slides - you will learn the rules for constructing regular expressions
- Best way to understand them is to see what they do when given as patterns to `grep`
- If you are curious what the regular expression `[acgt][acgt]*` matches, type the command

```
$ grep "[acgt][acgt]*"
```

  - w/o a file name after it - `grep` will use whatever lines you type on the keyboard to find a match
  - If what you **type** matches, then when you press the Enter key, it will `echo` it below
  - If not it will not `echo` it

Regular Expressions: What are They?

- regex is a pattern that represents a set of character strings
- Character string (string for short)
  - Is any sequence of characters, including blanks, newlines, punctuation, and control characters
  - For example:
    - If we invented a rule::
      - that '#' represents any single digit from 0 to 9
      - then the pattern ## would represent all strings consisting of exactly two digits
        - » such as 00, 01, 02, 03, ...,10, 11, ..., 20, ..., 30, ... 97 ,98, and 99
- We say that a **regex** matches a string **s** if **s** is in the set that the **regex** defines
- Thus ## would match 56 in our fictitious regex language

Regular Expression Building Blocks

- Basic regex are built up from operands and operators in much the same way that arithmetic expressions are constructed
- The fundamental building block of a regular expression is a single character
- Most single characters match themselves (not all do!)
- For example

```
a  matches 'a'
b  matches 'b'
1  matches '1'
```

so on…

Basic regex Operations: Concatenation

- Concatenation is the juxtaposition of two strings
- The concatenation of two regular expressions **r** and **s** is the set of all possible strings **xy**, where **r** matches **x** and **s** matches **y**

`aa matches 'aa'`

`11 matches '11'`

- Concatenation is associative:

`abc` is really `(ab)c` and so it matches '`ab`' concatenated with '`c`' which is '`abc`'

- Concatenation is really called product

Basic regex Operations: Closure (*)

- The only explicit, basic operator is * = closure operator
- A regex followed by * matches the concatenation of 0 or more strings each of which is matched by the regular expression
- For example:
- `a*` matches 0 or more a's:, `a`, `aa`, `aaa`, `aaaa` ...
- `ca*` matches `c` followed by whatever matches `a*`, so it matches `c`, `ca`, `caa`, `caaa`, `caaaa`, ...
- `ca*t` matches `cat, ccat, caat, cccat, ccaat, caaat` ...

Basic regex Operations: Closure (*)

- `cc*aa*` is the product of `cc*` and `aa*`
- Matches all strings formed in all possible ways by concatenating a string from `cc*` to one from `aa*`
- Write all strings of length 2, then length 3, then 4 and so on
- `ca, cca, caa, ccca, ccaa, caaa, cccca, cccaa, ccaaa, caaaa` , …
- Because a pattern like `a*` matches zero a's as well as 1 or more a's, if you want to match one or more a's, you need to use the regex `aa*`
  - which matches `a, aa, aaa`, and so on

More Examples of *

- If you want to apply the `*` operator to more than one character, you have to enclose it in `\( \)` brackets
- For example, a regex that matches all strings of the form `abababababab`, i.e., `ab` repeated any number of times, is `\(ab\)*`

```
$ grep "\(aa*\)" /data/METHODS/Fall/LECT6/test.txt

$ grep "\(aaa*\)" /data/METHODS/Fall/LECT6/test.txt
```

Basic Character Classes

- The period '.' matches any single character
- There are other one character regular expressions
- `[list-of-characters]` matches any single character in the list
- This is the same rule as file globs:
- `[a6j&]` matches a, 6, j, or &
- `[0-9]` matches any single digit
- `[a-zA-Z0-9]` matches any letter or digit
- The ^ inside brackets means the complement:
- `[^a6j&]` matches anything BUT a, 6, j, or &
- `[^0-9]` matches anything but a digit

FYI - Character Classes Combined with *

- You can use character classes with the * operator to create useful patterns:
- `\(c[acgt]g\)*` matches 0 or more sequences of cag, ccg, cgg, or ctg
- `[1-9][0-9]*` matches any decimal numeral except 0
- `[A-Z][a-z]*` matches words that start with an uppercase letter
- `\(...\)*` matches any string whose length is a multiple of 3..

FYI - Predefined Character Classes

- Certain character classes have special names
- Some of them are:

`[[:alpha:]]` matches any letter, upper or lowercase
`[[:alnum:]]` matches any letter or digit
`[[:lower:]]` matches lowercase letters
`[[:upper:]]` matches uppercase letters
`[[:punct:]]` matches punctuation
`\w` equivalent to [[:alnum:]]

- **These must be typed exactly as shown here**

Anchors

- Caret ^ anchors a regex - beginning of a line
- Sollar sign, $, anchors it to the end of the line
- For example:
- `^drwx` matches lines whose first 4 characters are drwx
- `^\w` matches lines that begins with a letter or digit
- `abcd$` matches lines whose last 4 characters are abcd
- `^abc$` matches lines that contain only abc
- `^$` matches empty lines
- `^[ ]*$` matches empty lines or lines containing only spaces

Metacharacters

- If you want to match one of the special characters such as *, [,], ., or -, you need to put a backslash in front of it:
- `\.` matches `.`
- `\*` matches `*`
- `\[` matches `[`
- `\]` matches `]`
- `\\` matches `\`
- These characters are called **metacharacters**
- Note: there are other ways to do this
- These are just the easiest to remember

Extended Regular Expressions

- Set of basic regexs - extended to include more powerful operators and has come to be called the **extended regular expression language**
- The `egrep` filter recognizes these expressions
- So does grep if you give it the `-E` option:

`egrep` is the same as `grep -E`

- Other programs recognize the extended regular expression language
  - Most notable are `sed` and `vi`
- The grep man page describes these expressions in sufficient detail
- We will cover a few useful operators

Extended Regular Expressions: | and +

- | **This is the OR-operator**
- If **`r`** and **`s`** are regular expressions, then **`r|s`** matches either strings that **`r`** matches or strings that **`s`** matches:

- `acg|act` matches either `acg` or `act`
- `aa*|bb*` matches either a sequence of 1 or more a's or sequence of 1 or more b's
- + **This is called positive closure**
- It is identical to * except it matches 1 or more instead of 0 or more occurrences:
- `a+` is the same as `aa*`
- `a+|b+` is the same as `aa*|bb*`

Extended Regular Expressions: ?

- `?` This matches 0 or 1 occurrences of its argument

- `a?` matches either the empty string or a
- `ab?a` matches either aa or aba
- `..?` matches any single symbol or two symbols
- `(cc?)+` matches 1 or more combinations of cc and c

- Beware: it is different than the glob ? operator !!!

Backreferences

- When you:
  - Enclose a basic regex in `\(  \)` brackets
  - Or an extended regex in ordinary parentheses `(  )`
  - String matched it is "**remembered**" for future use
- The regular expression backreference, `\1`, matches the first "**remembered string**."
- For example:

`\(aa*\)b\1`

- Any string that matches `aa*` is saved into a memory cell named `\1`
- Therefore the only strings that this expression matches are either **`aba`**, **`aabaa`**, **`aaabaaa`**, **`aaaabaaaa`**, etc

UNIX regex

- This part lecture introduced a very powerful computational tool called regular expressions
- The work that goes on behind the scenes to match them is significant
- Tools like grep can simplify many of the tasks you have to do, so it is worth the time to master regular expressions now

- Now we will Learn regex in Perl

Final Lecture on Perl today!!! ☺

Regular Expressions in Perl

Suppose We Have…

```
my $response;
my $responseIsValid = 0; # set to FALSE
while (!  $responseIsValid ) {
    say $question ,"\n> "; # display question
    $response = <STDIN>;
    chomp($response);
    #<<check if response is valid>>
}
```

- We need the chunk that validates input
- *pseudoCode*

Input Validation Chunk

```
#<<check if response is valid>>
if response is all digits or 'q' {
   set responseIsValid to TRUE
} end of if}
```

- How can we test if `$response` is all digits or equals the letter q?
- With Perl's matching operator and regular expressions
- Perl has regular expressions similar to grep's
- Let's learn more about it now

Perl's Match Operator

- Can check whether a variable, such as **$response, contains a string that matches a pattern using the expression:**

```
$response =~ m/pattern/
```

- Where ***pattern is a regular expression with usually the same syntax as those of grep***
- The **=~ operator is Perl's *binding operator***
- Variable on the left-hand side of **=~ is searched for a match of the pattern on the right-hand side**
- If a match is found, a true value is returned, otherwise a false value is returned

Regular Expressions in Perl

- Allow us to look for patterns in our data
- Use pattern to describe what we're looking for and check a value to see if it matches pattern
- **regex big area in Perl**
  - One of the most powerful features of Perl
    - Basic Patterns
    - Special Character Use
    - Quantifiers, anchors and transforming text using patterns
- In general if you ask Perl something about a piece of text
  - regex are going to be your first method

Patterns

- What constitutes a pattern?
- How do you compare it against something?
- The simplest pattern is a word
  - Simple sequence of characters
  - May want to ask Perl whether a certain string contains that word

```
if ($_ =~ /people/){
    say "Hooray! Found the word 'people'";
}
```

Writing a Regular Expression

- How do I tackle a regex in Perl
- Three Steps:
  - First, describe the pattern in English
  - Second, what part of match do you want to extract, if any?
  - Third, translate into Perl

`[A-Z]`	any capital letter	`\bword\b`	word anchor
`[0-9]*`	>= 0 numbers	`ATG/i`	ATG or atg
`\s+`	>= 1 space chars	`ATG/g`	all ATG's
`[^A]`	anything but 'A'	escaped characters: `\*` `\.` `\+` `\\|` `\\` `\/` `\#` `\"`	
`\d{3}`	3 digit numbers		

Special Characters in Perl RexEx

These are the characters that are given special meaning within a regular expression, which you will need to backslash if you want to use literally:

```
. * ? + [ ] ( ) { } ^ $ | \
```

Any other characters automatically assume their literal meanings.

You can turn off the special meanings using the escape sequence \Q

After Perl sees \Q, the 14 special characters will assume their ordinary, literal meaning

Turn back on using \E

```
if ( /\Q$pattern\E/ ){
```

What happens to the variable **`$pattern`** we checking here?

Interpolation

- RegEx work like a double-quoted string
  - Variables and metacharacters are interpolated
    - So you can store patterns in variables
    - Determine what we are matching when we run the programs
    - Don’t have to have them hard-coded

Let's look at testScriptRegExp1.pl

Anchors

- What if the pattern cannot be anywhere in the string?
  - So far that's how are patterns have been built
  - What if we know specifically the pattern is at:
    - Beginning
    - End
  - Extend our regex by telling Perl where the match must occur
    - This text must be
      - At the beginning of the string
      - At the end of the string
    - Done (just like before) by anchoring the match to either end
      - `^` = must appear at the beginning of the pattern
      - `$` = appears at the end of pattern

Modifiers / /ims

- i = will do a case insensitive pattern matching
- We will come back to m and s
- m = treat string as multiple lines
  - Change "^" and "$" from matching at only the very start or end, to the start or end of any line anywhere within the string
- s = treat string as a single line
  - Change "." Match any character whatsoever
    - Even a newline "\n", which it would **not** normally do
    - When we change $/ this comes into play

Shortcuts and Options in Patterns

- Finding patterns means more than just locating exact pieces of text
  - We may want to find three-digit numbers
  - 1st word on the line
  - Four or more letters all in capitals

Shortcut	Expansion	Description
`\d`	`[0-9]`	Digits 0 to 9.
`\w`	`[0-9A-Za-z_]`	A 'word' character allowable in a Perl variable name.
`\s`	`[ \t\n\r]`	A whitespace character that is, a space, a tab, a newline or a return.

also, the negative forms of the above:

Shortcut	Expansion	Description
`\D`	`[^0-9]`	Any non-digit.
`\W`	`[^0-9A-Za-z_]`	A non-'word' character.
`\S`	`[^ \t\n\r]`	A non-blank character.

Repetition

- What if we want:
  - To match three or more digits in a row
  - Two to four capital letters
- The metacharacters that we use to deal with a number of characters in a row are called quantifiers

`/bea?t/`	Matches either 'beat' or 'bet'
`/bea+t/`	Matches 'beat', 'beaat', 'beaaat'…
`/bea*t/`	Matches 'bet', 'beat', 'beaat'…

Look Familiar?

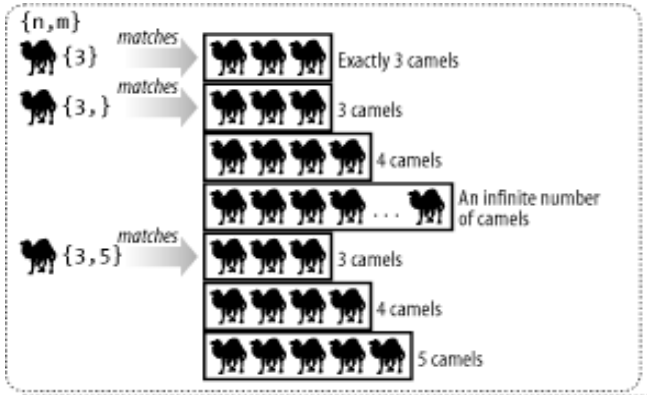
Metacharacters Summed Up

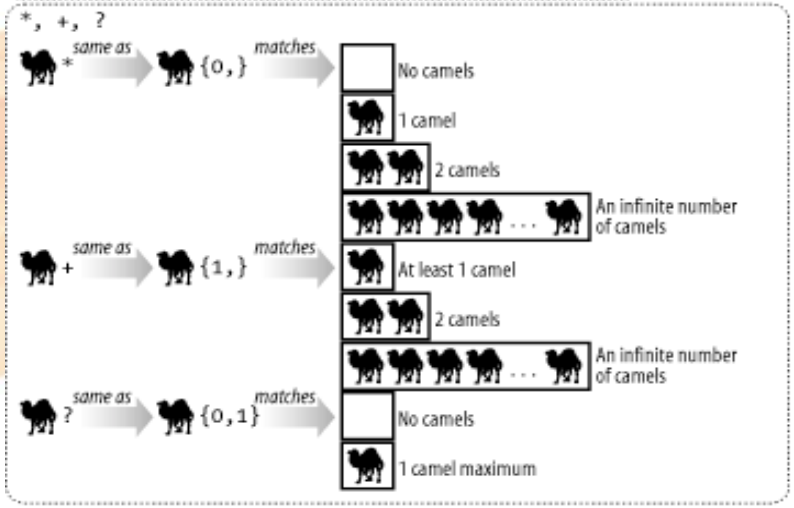
Metacharacter	Meaning
`[abc]`	any one of the characters a, b, or c.
`[^abc]`	any one character other than a, b, or c.
`[a-z]`	any one ASCII character between a and z.
`\d \D`	a digit; a non-digit.
`\w \W`	a 'word' character; a non-'word' character.
`\s \S`	a whitespace character; a non-whitespace character.
`\b`	the boundary between a `\w` character and a `\W` character.
`.`	any character (apart from a new line).
`(abc)`	the phrase 'abc' as a group.
`?`	preceding character or group may be present 0 or 1 times.
`+`	preceding character or group is present 1 or more times.
`*`	preceding character or group may be present 0 or more times.
`{x,y}`	preceding character or group is present between *x* and *y* times.
`{,y}`	preceding character or group is present at most *y* times.
`{x,}`	preceding character or group is present at least *x* times.
`{x}`	preceding character or group is present *x* times.

Matching Multiple Characters

- `x*` matches zero or more x's (**greedily**)
- `x+` matches one or more x's (**greedily**)
- `x{n}` matches n x's
- `x{m,n}` matches from m to n x's

- **greedily** – means match as many as it can





Review: Pattern-Matching

- The following code:

```
if ($_ =~ /ACGCGT/) {
    say "Found MCB binding site!";
}
```

  prints the string "`Found MCB binding site!`" if the pattern "`ACGCGT`" is present in the default variable, `$_`

- Instead of using `$_` we can "bind" the pattern to another variable (e.g. `$dna`) using this syntax:

```
if ($dna =~ /ACGCGT/) {
    say "Found MCB binding site!";
}
```

- We can even use a variable in the pattern, to search for multiple binding sites

```
foreach my $key (keys %bindingSites){
        my $bindingSite = $bindingSites{$key};
        if ($dna =~ /$bindingSite/) {
                say "Found ", $key , " binding site!";
        }
}
```

Here we have a hash: **keys** type of binding site and the **value** is the site

What are we using in the above code

How Can We Substitute?

- We can replace the first occurrence of `ACGCGT` with the string `_MCB_` using the following syntax:

```
$dna =~ s/ACGCGT/_MCB_/;
```

- We can replace all occurrences by appending a '`g`'

```
$dna =~ s/ACGCGT/_MCB_/g;
```

Matching Alternative Characters

- In general square brackets denote a set of alternative possibilities
  - [ACGT] matches **one** A, C, G or T:
-

```
while ($_ = <INFILE>) {
   chomp $_;
   if ($_ =~ /[ACGT]/ ){
      say "Matched: $_"
   }
}
```

  - Use - to match a range of characters: `[A-Z]`
  - `[^X]` matches anything but `X`
    - notice the ^ now has a different meaning

Matching Alternative Strings

- `/(this|that)/` matches "this" or "that"
- ...and is equivalent to `/th(is|at)/`

```
while ($_ = <STDIN>) {
        if ($_ =~ /this|that|other/){
                say "Matched: $_"
        }
        …
        …
        …
}
```

Take look at thisOrThat.pl

```
This loop will match this|that|other, and exit to end the loop
THIS
Will not match THIS
Other
Will not match Other
other
Matched: other
that
Matched: that
exit
Ending program now
```

Remember, regex's are case-sensitive

"Escaping" special characters

- \ is used to "escape" characters that otherwise have meaning in a regex
- so `\[` matches the character "`[`"
  - Remember, if not escaped, "`[`" signifies the start of a list of alternative characters, as in `[ACGT]`

Retrieving What Was Matched

- If parts of the pattern are enclosed by parentheses, then (following the match) those parts can be retrieved from the scalars `$1, $2`...

```
while ($_ = <STDIN>) {
  chomp $_;
  if ($_ =~ /(a|the)\s+(\S+)/i)
{
    say "Noun: " , $2;
  }
}
```

```
Pick up the cup
Noun: cup
Sit on a chair
Noun: chair
Put the milk in the tea
Noun: milk
```

Different Example:

e.g. `/the (\S+) sat on the (\S+) drinking (\S+)/`

matches "the cat sat on the mat drinking milk"

with `$1="cat", $2="mat", $3="milk"`

Note: only the first "the" is picked up by this regex

Back to Our Input Validation Chunk

```
#<<check if response is valid>>
if response is all digits or 'q' {
   set responseIsValid to TRUE
} end of if}
```

- How can we test if `$response` is all digits or equals the letter q?
- With Perl's matching operator and regular expressions
- Perl has regular expressions similar to grep's
- Let's learn more about it now

Back to Our Input Validation Chunk

- For example, to check if **`$response`** matches a line that contains at least one digit and nothing but digits:

```
$response =~ m/^[0-9]+$/
```

- or equivalently:

```
$response =~ m/^\d+$/
```

- Since `\d` is the pattern that matches any digit, and + is the "1 or more occurrences" operator
- Remember that `^` and `$` are anchors to the beginning and end of a line in grep?
- In Perl they anchor to the beginning and end of a string

Almost There

- To check if **`$response`** **is equal to a particular string, we can use the string comparison operator, eq:**

```
$response eq 'q'
```

- This returns true if and only if the string stored in **`$response`** **is exactly 'q'. Putting this together, we have:**

```
if ( $response =~ m/^[0-9]+$/ || $response eq 'q' ){
      responseIsValid = 1; #breaks the while loop
}
else {
      print "Invalid Input: Enter an integer ";
      say "or 'q' quit";
}
```

Finishing Up

- The last chunk to convert is the chunk that compares the response to the solution and displays the appropriate message.

```
<<check correctness of user's response>>
if response equals solution {
   display correct_response_message
}
else {
   display incorrect_response_message
} end if
```

```
<<check correctness of user's response>>
if ($response == $solution) {
   say "Correct!";
}
else {
   say "Incorrect: " , $question , " " ,  $solution;
}
```

Questions?

- Sounds like a ton of information
  - And it is!
  - This is only the beginning
    - Much more in 6200
    - This is enough to get you started
- Can be very hard to get right at first
- Even after years of programming you will constantly have to go back and look at some rules
  - Constant source of **frustration** and **bugs**
  - But regex's are so powerful, you just can't afford to ignore them
  - Good thing is: Many languages have adopted Perl regex
- Lets take a look at some bioinformatics examples

Not a Given Residue

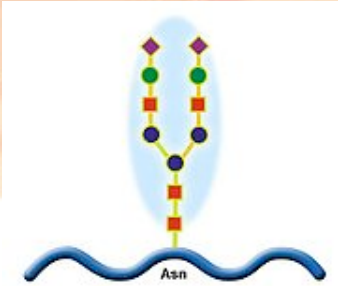
- Suppose you know from structural reasons that a residue cannot be a Proline
- You could write: `[ACDEFGHIKLMNQRSTVWY]`
- That's tedious, so what notation would we use?

`[^P]`

This matches anything which is *not* a Proline
(Yes, using the ^ is strange. That's the way it is)

N-glycosylation Motif

- Common post-translational modification in ER
  - Membrane & secreted proteins
  - Purpose: folding, stability, cell-cell adhesion
- Attachment of a 14-sugar oligosaccharide
- Occurs at asparagine residues with the consensus sequence:
  - `N{P}[ST]{P}` - This is the way it's shown publications, **not in Perl**!!!!!
  - `{X}` means any amino acid except `X`; and `[XY]` means either `X` or `Y`
- Can we detect potential N-glycosylation sites in a protein sequence?



N-glycosylation Site Detector

Convert to upper case

```
while ($_ = <STDIN>) {
  chomp $_;
  $_ = uc $_;
  while (/(N[^P][ST][^P])/g) {
    say "Potential N-glycosylation sequence ",
            $1, " at residue ", pos() - 3;
  }
}
```

using the 'g' modifier to get all matches in sequence

`pos()` is index of first residue after the match, indexes **start at zero**;

so, `pos() - 3` is the index of first residue of four-residue match, starting at one, b/c biologist think strings start at 1! -)

See testScriptRegExp2.pl

Sequence Suggests Structure/Function

- When working with tumors you find the p53 tumor antigen, which is found in increased amounts in transformed cells.
- After looking at many p53's you find that the substring:

  `MCNSSCMGGMNRR`

  is well conserved and has few mismatches

  `MCNSSC`**`V`**`GGMNRR`

- If you have a new protein sequence and it has this substring then it is likely to be a p53 tumor antigen

Exercise: Write a script that will check for the p53 motif, hint: use a regular expression
–why?

p53 Tumor Antigen Protein

- Many contain the string:
  - MCNSSCMGGMNRR
- Others contain the string:
  - MCNSSC**V**GGMNRR

Can you write the regular expression?

Aspartic Acid and Asparagine Hydroxylation site

- Consensus pattern:

  `C - C - x(13) - C - x(2) - [GN] - x(12) - C - x - C - x(2,4) - C`

- As regular expression:

  `CC.{13}C.{2}[GN].{12}C.C.{2,4}C`

Lets use this format here on:

- . is same as `.{1}`

- Special repeat ranges:
  - Optional: `?` is same as `{0,1}`
  - 0 or more: `*` is same as `{0,}`
  - 1 or more: + is same as `{1,}`

Repeated Residues

- Sometimes you'll repeat yourself. For example, a pattern may require 3 hydrophobic residues between two well conserved regions
- You could write it as

  `[FILAPVM][FILAPVM][FILAPVM]`
- But we're programmers, we're lazy, so what should we do?

Regular expression:

```
[FILAPVM]{3}          - exactly 3 hydrophobic residues
[FILAPVM]{3,5}        - between 3 and 5
[FILAPVM]{,3}         - at most 3
[FILAPVM]{3,}         - at least 3
```

`.{10}` matches exactly 10 residues

- domain signatures often have spacers

EGF-like domain signature 2

PS01186 is: `C.C.{2}[GP][FYW].{4,8}C`

Use a spacer of at least 4 residues
and up to (and including) 8 residues

```
RHCYCEEGWAPPDCTTQLKA
RHCYCEEGWAPPDECTTQLKA
RHCYCEEGWAPPDEQCTTQLKA
RHCYCEEGWAPPDEQWCTTQLKA
RHCYCEEGWAPPDEQWICTTQLKA
```

^Examples$

^A	start with an A
^[MPK]	start with an M, P, or K
E$	end with an E
[QSN]$	end with a Q, S, or N
^[^P]	start with anything except P
^A.*E$	start with an A and end with an E

Perl and Greediness

- When a regular expression uses the '*' wild card operator to match text:
  - regex will attempt to match as much as possible when applying the regular expression (greedy)
- Given the the following Perl code with the regular expression “(Some.*text)”:

```
my $string = "Some chunk of text that has text";
if ($string =~ /(Some.*text)(.*)/){
   say "One: ", $1, "\nTwo: ", $2;
}
```

```
One: Some chunk of text that has text
Two:
```

greedy1.pl

Perl and Greediness

- In the previous slide:
  - This is considered a "greedy" regex since it attempts to match as much as possible
  - This is not ideal in most situations, and is easily fixed with Perl's '?' operator:

```
my $string = "Some chunk of text that has text";
if ($string =~ /(Some.*?text)(.*)/){
   say "One: ", $1, "\nTwo: ", $2;
}
```

```
One: Some chunk of text
Two:  that has text
```

- Perl – no longer greedy when evaluating the expression, and will attempt to match up to the left-most occurrence of the string element prefaced by '?'

greedy2.pl

Putting All the Regex Rules Together

GenBank Flat File Format

- The GenBank Flat File sequence format is a rich format for storing sequences and associated annotations
  - It shares a feature table vocabulary and format with the EMBL and DDJB formats
- 3 Parts:
  - Header
  - Features
  - Sequence

GenBank Flat File Format – Header (2)

- LOCUS - A short mnemonic name for the entry. The line contains the Accession number, length of molecule, type of molecule (DNA or RNA), a three letter reference to possibly Taxonomy, and the date that the data was made public
- DEFINITION - A concise description of the sequence
- ACCESSION - The primary accession number is a unique, unchanging code assigned to each entry. Used often when citing sequence in journals
- VERSION - The primary accession number and a numeric version number associated with the current version of the sequence data in the record. This is followed by an integer key (a "GI") assigned to the sequence by NCBI
- KEYWORDS - Short phrases describing gene products and other information about an entry
- SOURCE - Common name of the organism or the name most frequently used in the literature

GenBank Flat File Format – Header (2)

- ORGANISM - Formal scientific name of the organism (first line) and taxonomic classification levels (second and subsequent lines)
- REFERENCE - Citations for all articles containing data reported in this entry.
- AUTHORS - Lists the authors of the citation
- TITLE - Full title of citation
- JOURNAL - Lists the journal name, volume, year, and page numbers of the citation
- MEDLINE - Provides the Medline unique identifier for a citation
- PUBMED - Provides the PubMed unique identifier for a citation
- REMARK - Specifies the relevance of a citation to an entry
- COMMENT - Cross-references to other sequence entries, comparisons to other collections, notes of changes in LOCUS names, and other remarks

GenBank Flat File Format – Features

- SOURCE: contains information about organism, mapping, chromosome, tissue alignment, clone identification
- CDS: instructions on how to join sequences together to make an amino acid sequence from the given coordinates. Includes cross references to other databases
- GENE Feature: a segment of DNA identified by a name
- RNA Feature: used to annotate RNA on genomic sequence (for example: mRNA, tRNA, rRNA)
- Much more can be found at the GenBank Feature Table Definition

GenBank Flat File Parsing

- Lets look at the file format - sampleGenbank.gb
- Let's use Perl to Parse GenBank files
  - There are BioPerl modules to do this:
    - Bio::SeqIO system using the Bio::SeqIO::genbank
    - But we'll use our own to learn more about parsing
- Use regular expressions to extract information from the GenBank Flat File
  - Accession
  - DNA Sequence
  - Protein Sequence
  - Gene
  - Organism

See testScriptRegExp3.pl

GenBank Flat File Parsing

- Using regular expressions to grab the Accession number from file

```
ACCESSION   BC013459
```

```
sub getAccession {
      my ($GB_file) = @_;
      if($GB_file =~ /ACCESSION\s*(\w+)/){
                return $1;
      }
      else{
                return 'error';
      }
}
```

The world is out to get you!

GenBank Flat File Parsing

- Using regular expressions to grab the gene name from file

```
/gene="PB2"
```

```
sub getGene {
        my ($GB_file) = @_;
        if($GB_file=~/gene="(.*?)"/s){
                return $1;
        }
        else{
                return 'unknown';
        }
}
```

What does this do?

What does this do?

GenBank Flat File Parsing

- Using regular expressions to grab the DNA sequence from file

```
ORIGIN
        1 atggagagaa taaaagaact gagagatcta atgtcgcagt cccgcactcg cgagatactc
       61 actaagacca ctgtggacca tatggccata atcaaaaagt acacatcagg aaggcaagag
                                   ........
                                   ........
//
```

```
sub getDnaSequence {
      my ($GB_file) = @_;
      my $seq;
      if($GB_file=~/ORIGIN\s*(.*)\/\//s){
                $seq = $1;
      }
      else{
                return "unknown";
      }

      $seq =~ s/[\s\d]//g;
      return uc($seq);
```

What does this do?

GenBank Flat File Parsing

- Using regular expressions to grab the protein sequence from file

```
/translation="MIPGNRMLMVVLLCQVLLGGASHASLIPETGKKKVAEIQGHAGG
RRSGQSHELLRDFEATLLQMFGLRRRPQPSKSAVIPDYMRDLYRLQSGEEEEEQSQG
TGLEYPERPASRANTVRSFHHEEHLENIPGTSESSAFRFLFNLSSIPENEVISSAELR
LFREQVDQGPDWEQGFHRINIYEVMKPPAEMVPGHLITRLLDTRLVHHNVTRWETFDV
SPAVLRWTREKQPNYGLAIEVTHLHQTRTHQGQHVRISRSLPQGSGDWAQLRPLLVTF
GHDGRGHTLTRRRAKRSPKHHPQRSRKKNKNCRRHSLYVDFSDVGWNDWIVAPPGYQA
FYCHGDCPFPLADHLNSTNHAIVQTLVNSVNSSIPKACCVPTELSAISMLYLDEYDKV
VLKNYQEMVVEGCGCR"
```

```
sub getProteinSequence {
    my ($GB_file) = @_;
    my $pro;
    if($GB_file=~/translation="(.*?)"/s){
          $pro = $1;
    }
    else{
          return "unknown";
    }

    $pro =~ s/[\s]//g;
    return uc($pro);
}
```

What does this do?

GenBank Flat File Parsing

- Using regular expressions to grab the organism name from file

```
/organism="Influenza A virus (A/Quebec/144147/2009(H1N1))”
```

```
sub getOrganism ($){
     my ($GB_file) = @_;
     if($GB_file=~/organism="(.*?)"/s){
         return $1;
     }else{
        return 'unknown';
     }
}
```

Filters in UNIX

adopted from Stewart Weiss

What's a Filter in UNIX

- `awk`, `sort`, & `uniq` - examples of a class of UNIX programs called filters
- A filter is a UNIX command
  - Input and output are ordinary text
  - Expects:
    - its input from standard input
    - puts its output on standard output
- Filters transform their input in some way:
  - i.e sorting it
  - Removing words or lines based on a pattern or on their position in the line or file
    - remove every 3rd word in a line
    - or remove every 4th line
    - or remove any line that has a certain word

Filtering Standard Input

- Filters may have file name arguments on the command line, but when they have no arguments, they read from standard input (the keyboard) instead:

```
$ grep 'a clue' thehouse.txt
```

- Searches for 'a clue' in `thehouse.txt`, whereas in

```
$ cat thehouse | grep 'a clue'
```

- `grep` searches through its standard input stream for lines with 'a clue'

Some Useful Filters

Filter	description
`grep`	global regular expression parsers
`sort`	sorts based on several criteria
`uniq`	removes adjacent identical lines
`awk`	full-fledged programming language for field-oriented pattern matching
`cut`	removes pieces of each line based on positions

http://en.wikipedia.org/wiki/Filter_%28Unix%29

Some Additional Filters

Filter	description
`head, tail`	display just top or bottom lines of files
`cat`	null filter -- shows everything in order
`tac`	shows lines in reverse order
`fold -w<N>`	display output in width of N columns
`sed`	stream editor -- very powerful filter
`wc`	not exactly a filter, display count of chars, words, and lines

Selected Filters: `sort`

- `sort` program can be used for sorting one or more text files in sequence
- In simplest form

```
$ sort filename
```

- sort `filename`
  - Using the first line field (first chars up to the first white space) in ASCII collating order
  - Displaying the sorted file on the screen (standard output)
- `sort`
  - Will ignore case by default in some versions of UNIX
  - Whereas in others, uppercase and lowercase letters will be treated as different
  - Try it on your Mac then on fisher

Selected Filters: `sort`

- `sort` program will not sort numbers properly unless you tell it to sort numerically
- It treats them like letters, by default
- For example, if `scores` contains the two lines

```
5
10
100
```

- Then we get

```
$ sort scores
10
100
5
```

- Because "1" precedes "5", so "10" & "100" precedes "5"

More on `sort`

- To sort numerically, you give sort the `-n` flag:

```
sort -n filename
```

- as in:

```
$ sort -n scores
5
10
100
```

- To reverse the order of the sort use the `-r` flag

`sort` by Fields

- To sort using the second field of the line

```
sort -k2 filename
```

- To sort using the third field of the line

```
sort -k3 filename
```

- To use the first field as the primary key, then the second field as secondary key using numeric sorint, use

```
sort -k1 -k2n filename
```

/data/METHODS/Fall/LECT6/test.txt

Selected Filters: `uniq`

- `uniq` removes a line from a file if it is identical to the one preceding it
- If you `sort` a file that has duplicate lines, and pipe it through `uniq`, the duplicates will be removed
- You could do the same thing by using the `-u` option with `sort` though, so this is not why `uniq` is unique
  - Sometimes there are files that are not sorted but have "runs" of the same lines
  - You could sort them using `sort -u`, but it is fast to just run `uniq` on them
- You can also do some chaining:

```
$ sort test.txt | uniq -c | sort -k1nr
```

Selected Filters: `fold`

- `fold` filter breaks each line at a fixed number of characters, so that each line is at most a certain length

- Suppose dnastring has the line

```
agatggcggc

$ fold -c4 /data/METHODS/Fall/LECT5/dnastring #produces
agat
ggcg
gc
```

Selected Filters: `wc`

- `wc` command, by default, displays the numbers of lines, words, and characters in one or more files given on the command line, or in its input stream if not given any arguments

```
$ wc /data/METHODS/Fall/LECT5/*
    1      1     11 /data/METHODS/Fall/LECT5/dnastring
    5     56    331 /data/METHODS/Fall/LECT5/nlexample.txt
   10     15    591 /data/METHODS/Fall/LECT5/sequences.fasta
```

- Tells me how many lines, words, and characters are in the fasta file
- I can give it -m, -w, or -l for chars, words, and lines to restrict its output

- Very nice to find the length of a sequence:

```
$ echo -n 'CCGGGTCGCGGGCCCCGGGCTCGGGGCCGCCCTCCGCGT' | wc
    0       1      39
```

Selected Filters: `fold`

- For example:

```
$ who | wc -l
2
```

displays the number of users currently logged in

```
$ ps -ef | grep '/bin/bash' | wc -l
5
```

displays how many people are running bash at the moment

Things to Do for Thursday

- Perl Readings
  - **References**
  - **Subroutines**
- Look over the example scripts provided today
- Bring questions on any topics we covered in Perl
  - Check out posted solutions to Lab 2.
  - Make sure to complete Perl questions from last week
  - Thursday we will be doing our last Perl coding lab
- **Get you article approved by next Tuesday!**