Perl Regular Expressions

Today's lecture: Regular Expressions in Perl
- Pattern matching via regular expressions is very useful in processing the output of other programs
- An essential part of Perl's ability to act as a scripting language

Next Lecture:
- launching and controlling other programs
- common libraries

Regular Expressions

- A regular expression (RE) is a mapping from strings to sets of strings
- Domain: a string consisting of
  - literals
  - metacharacters -- characters that represent sets of strings
- Range: strings of characters from a specified alphabet, e.g. ASCII or UNICODE
- Metacharacters are sometimes called "operators"

Regular Expression Examples

- Assume the alphabet consists of the 26 lower case letters
- The . metacharacter means "any letter" and the expression a*bc* means "zero or more a's"
- Parentheses can be used to group strings
- Examples of REs and their values:
  - abe
  - a.e
  - a*be*
  - (ab)*e
  - .* [any string of lower case letters]

Running Example: Yeast Gene IDs

- To illustrate Perl regular expressions, assume we’re writing a script that will be used in an analysis of yeast chromosomes
- Gene IDs have the following format:
  - the letter Y
  - a chromosome ID (A through P)
  - the letter L or R (left or right “arm” of the chromosome)
  - a three-digit integer
  - the letter C (gene is on complementary strand) or W

The ., *, and + Operators

- The ., *, and + operators used in earlier examples have the same meanings in Perl:
  - . any single character
  - * zero or more of the previous item in the pattern
- Another operator commonly used in REs and also in Perl:
  - + one or more of the previous item
- Note:
  - in Unix shells * means “zero or more characters”
  - in Perl, write “zero or more characters” as .*

. , *, and + Examples

- In Perl, programmers typically use / to delimit patterns in pattern matching operations

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a/</td>
<td>a, ab, abc, ...</td>
</tr>
<tr>
<td>/a+/</td>
<td>&quot;&quot;, a, ab, abc, ...</td>
</tr>
<tr>
<td>/ab+e/</td>
<td>ac, abe, abbe, ...</td>
</tr>
<tr>
<td>/ab+/</td>
<td>abe, abbe, ...</td>
</tr>
<tr>
<td>/a.+e/</td>
<td>aac, abe, aee, aye, ...</td>
</tr>
</tbody>
</table>

Note:
Multipliers

- Perl extends the idea of “zero or more” or “one or more” by letting you set numeric limits
- Use \{n,m\} to mean “between n and m occurrences” of the previous item
- \(\{n\}\) means “exactly n”
- \(\{n,\}\) means “at least n”

Pattern | Example Matching Strings
--------|-----------------------------
/a(1,3)/ | a, aa, aaa
/a(1,5)/ | alpha, beta, gamma, ...
/a(5)/ | alpha, gamma, omega, ...

Optional Items

- Use ? to show an item is optional
- ? is just a shorthand for \{0,1\}

Pattern | Example Matching Strings
--------|-----------------------------
/ab?c/ | ac, abc
/ab(0,1)c/ | ac, abc

Yeast IDs with Multipliers

- Here are some examples of how to use multipliers in the pattern that specifies yeast gene IDs:
  - /YAL.(3)C/ any gene on the complementary strand of the left arm of chromosome 1
  - /Y.....C/ any gene on a complementary strand

Quoting Special Characters

- To include a pattern matching operator in a string, use a backslash to quote it

Pattern | Example Matching Strings
--------|-----------------------------
/a\b/ | a.b
/a\*/ | a, a*, a*b*bb*, ...
/2\+3+/ | 2+3, 2+33, 2+333, ...
/2\+/ | 2/3
/\a\*/ | a, a*, a**, a***, ...
/\a/// | \a\%

Parentheses

- Use parentheses to group substrings

Pattern | Example Matching Strings
--------|-----------------------------
/(ab)+(cd)/ | abab, abcded, ...
/(ab)/ | "", aa, aaaa, ababdef
/\(a.*\)/ | (a), (abba), (alfred), (a star)
/\(a.*\)/ | (one two), (a b), (zz top)
/(\..+)/ | 1, 1.1, 1.1.1, one.two.three

Alternatives

- A vertical bar means “or”
- Use it to specify alternative ways of matching a pattern

Pattern | Example Matching Strings
--------|-----------------------------
/a|b/ | a, b
/a|b\*/ | a, b, bb, bbb, ...
/(a|b\+)/ | a, b, ab, ba, abba, ababaabab
Sets

- A set notation provides an alternative for abbreviations
- A string enclosed in square brackets means "any one of these characters"
- Within a set, - denotes a range of characters

Pattern | Example Matching Strings
--- | ---
/a|b|c|d|e/ | a, b, c, d, e
/\[abcde\]/ | a, b, c, d, e
/\[-a-z\]/ | a, z
/\[a-z\]\+/ | a, z, zzygy
/\[0-9\]+/ | 1, 2, 226, 191239891
/\[\+\-\*\/\]/ | +, -,

Yeast IDs with Alternatives

- Examples of using sets and alternatives in yeast gene IDs:
  - /YA(\L|R).{3}(C|W)/
    - any gene on chromosome L
  - /Y.(\L|R).{3}(C|W)/
    - any gene
  - /Y\[A-P\].{3}\[LR\].{3}\[CW\]/
    - any gene; a "tighter" specification since it doesn't match YER, etc
  - /Y\[A-P\].{3}\[LR\].{3}\[0-9\]+\[CW\]/
    - even tighter yet — only digits allowed after L or R

Character Classes

- Perl has several predefined character sets
  - \w "word character"  \[A-Za-z0-9\]
  - \s "white space"  \[ \t\n\r\]
  - \d "digit"  \[0-9\]

Pattern | Example Matching Strings
--- | ---
/\w+/ | Able, Baker, fred, ...
/\w\|/ | '', 12, 007, ...
/\w\[A-Za-z0-9\]\+/ | (12, 266), (0, 1), (100 345), ...

Character Classes (cont'd)

- If ^ is the first character in a set, it negates the set, i.e. the construct means "anything but these characters"

Pattern | Example Matching Strings
--- | ---
/\w\|/ | aax, adx, aox, ...
/\w\|/ | aax, adx, aox, ...
/\w\|/ | aax, adx, aox, ...
/\w\|\d\s\w\d\s\+/ | (12, 266), (0, 1), (100 345), ...

There are three predefined inverse sets:
  - \W punctuation (i.e. anything but \w)
  - \D non-digit
  - \B printing char (anything but \w)

Precedence Issues

- So far we've relied on intuition to define precedence
- Most expressions we've seen are actually ambiguous

Examples:
  - /ab*/
    - "a followed by zero or more b's" or "zero or more pairs of ab"?
  - /one|two/  "one or two" or "one or two or one two"?
  - /ab?c/  is it b that is optional, or ab?

Precedence Rules

- The precedence of pattern matching operators in Perl is shown in this table:
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups</td>
<td>()</td>
<td>highest</td>
</tr>
<tr>
<td>Multipliers</td>
<td>* + ? {n,m}</td>
<td></td>
</tr>
<tr>
<td>Characters</td>
<td></td>
<td>lowest</td>
</tr>
<tr>
<td>Alternative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use parentheses (groups) to alter precedence</td>
<td>/ (ab)?/</td>
<td>one or more ab pairs</td>
</tr>
</tbody>
</table>
Perl’s =~ Operator

- Now that we know how to construct regular expression patterns, let’s look at how to use them
- The =~ operator is Perl’s pattern matching operator:
  - `$string =~ /pattern/`
    - Here `$string` is typically a scalar variable
    - The pattern to be matched is enclosed in delimiters (slashes in the above example)
- The operator evaluates to 1 (“true”) if the pattern occurs anywhere in the string

File Processing Idiom

- Here is a very common program structure

```perl
open(IN,$filename);
while ($line = <IN>) {
  if ($line =~ /pattern/)
    # line is one we’re looking for --
    # do something
}
close(IN);
```

The ! =~ Operator

- !~ means “does not match”

```perl
if ($line !~ /pattern/)
  #

is the same as

if (!$line =~ /pattern/)
  #
```

=~ Matches Substrings

- The =~ operator evaluates to true if any substring of the left operand matches the pattern

```perl
"abc" =~ /abc/
"ababcab" =~ /abc/
"abacaba" =~ /abc/
"abababab" =~ /abc/
```

Extracting Values

- Up to now the only use for pattern matching has been in boolean expressions, e.g. “if this line matches pattern P”
- Perl regular expressions can be used to extract portions of the string being matched
- Example:
  - `$id = "gi|6319950 YCR106W ...";
    ($id) = ($line =~ /\{A-P\}\{LR\}\{d(3)\}\{CM\}\}/);
  - The variable `$id` is assigned the value “YCR106W”

Use Groups to Define Extracted Values

- Perl uses groups to define the parts of patterns that are extracted
- Each time Perl matches a pattern inside parentheses, it assigns the portion of the string that matches the pattern to a special variable
- The variables are named $1, $2, etc
  - $1 holds the substring matching the first group
  - $2 holds the substring matching the second group
  - etc
Group Value Example

- Example of using groups to extract values:

```
while ($s = <IN>) {
    if ($s =~ /Y([A-P])([LR])(\d{3})([WC])/) {
        print "found gene $3 on $2 arm of";
        print " chromosome $1\n";
    }
}
```

All Groups Are Assigned Values

- A group is always assigned a matching substring
- If you want to use parentheses only for altering precedence, use `?` to tell Perl not to assign it a value:

```
while ($s = <IN>) {
    if ($s =~ /Y([A-P])?:([LR])(\d{3})[WC]/) {
        print "gene $2 on chromosome $1\n";
    }
}
```

Value of the =~ Operator

- The value of the =~ operator is a list of the values assigned by group operations

```
while ($s = <IN>) {
    ($chr,$arm,$num) = ($s =~ /Y([A-P])([LR])(\d{3})[WC]/);
    print "found gene $num on $arm arm of";
    print " chromosome $chr\n";
}
```

Anchors

- Use ^ or $ to "anchor" a pattern at the beginning or ending of a string
- Anchor symbols have the same precedence as literal characters

```
$s = "gi|22652|ref\NP_00001.1 hypothetical";
$s =~ /\^\NP_00001.1/ => 1
$s =~ /ref\$/ => 1
$s =~ /gi\$/ => 0
$s =~ /\NP_00001.1\$/ => 1
$s =~ /gi.*cal\$/ => 1

$\id = "\NP_00001.1\ref";
$\id =~ /\NP_00001.1\ref\$/ => 1
$\id =~ /\NP_00001.1\$/ => 1
$\id =~ /\NP_00001.1\ref\$/ => 1
```

Lazy Matches

- Ordinarily Perl uses as many characters as it can to satisfy a pattern matching operation
- Use ? to tell it to use as few characters as possible

```
$s = "gi|22652|ref\NP_00001.1 hypothetical";
($\id) = ($s =~ /\NP_00001.1\ref\$/);
$id = "22652\ref";
($\id) = ($s =~ /\NP_00001.1\$/);
$id = "22652";
```
Variables in Patterns

- So far all the pattern examples have been literal strings
- It is possible to include variables in patterns, e.g.
  
  ```perl
  $target = 226;
  while ($s = <IN>) {
    if ($s =~ /\G\$target/) {
      ...
    }
  }
  ```

  - These patterns can be inefficient, since the Perl compiler preprocesses patterns, and these must be processed at runtime

Substitutions

- Regular expressions can be used to modify strings
- The expression
  
  ```perl
  $a =~ s/\pat/\rep/)
  ```

  means "replace \pat by \rep in the string $a"

  - Example:
    
    ```perl
    $a = "hello, world";
    $a =~ s/hello/goodbye/;
    print "a = "$a"\n";
    a = 'goodbye, world'
    ```

Further Reading

- The topics covered in this lecture will get you through the Perl projects this term
- Read about regular expressions in a Perl reference to learn more:
  - alternative delimiters
  - additional anchors (e.g. word boundaries)
  - global matches
  - options (e.g. ignore case)
  - translations (e.g. upper to lower case)