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

4 A regular expression (RE) is a mapping from strings to sets of strings
4 Domain: a string consisting of
   - literals
   - metacharacters -- characters that represent sets of strings
4 Range: strings of characters from a specified alphabet, e.g. ASCII or UNICODE
4 Metacharacters are sometimes called “operators”
Assume the alphabet consists of the 26 lower case letters

The \metacharacter \texttt{.} metacharacter means “any letter” and the expression \texttt{x*} metacharacter means “zero or more \texttt{x}’s”

Parentheses can be used to group strings

Examples of REs and their values:

- \texttt{abc} \hspace{1cm} abc
- \texttt{a.c} \hspace{1cm} aac, abc, acc, adc, aec, ...
- \texttt{a*bc*} \hspace{1cm} b, ab, aab, ..., bc, bcc, aabc, ...
- \texttt{(ab)*c} \hspace{1cm} c, abc, ababc, abababc, ...
- \texttt{.*} \hspace{1cm} [any string of lower case letters]
Running Example: Yeast Gene IDs

4 To illustrate Perl regular expressions, assume we’re writing a script that will be used in an analysis of yeast chromosomes

4 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

4 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
4 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 . *


Examples

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

Pattern | Example | Matching Strings
--- | --- | ---
/../ | aa, ab, ac, ... | 
/ab*/ | "", a, aa, aaa, ... | 
/ab+c/ | ac, abc, abbc, ... | 
/a.+c/ | aac, abc, axc, ayc, ... |
Multipliers

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

4 Pattern        Example Matching Strings
  – /a{1,3}/     a, aa, aaa
  – /.{1,5}/     alpha, beta, gamma, ... 
  – /.{5}/       alpha, gamma, omega, ...
Optional Items

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

4 Pattern       Example Matching Strings
   /ab?c/       ac, abc
   /ab{0,1}c/   ac, abc
Yeast IDs with Multipliers

4 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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a.b/</td>
<td>a.b</td>
</tr>
<tr>
<td>/a*.*/</td>
<td>a*, a<em>a, a</em>bba, …</td>
</tr>
<tr>
<td>/2+3+/</td>
<td>2+3, 2+33, 2+333, …</td>
</tr>
<tr>
<td>/2/3/</td>
<td>2/3</td>
</tr>
<tr>
<td>/a**/</td>
<td>a, a*, a**, a***, …</td>
</tr>
<tr>
<td>/\a\/</td>
<td>\a\</td>
</tr>
</tbody>
</table>
Parentheses

4 Use parentheses to group substrings

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/(ab)+(cd)/+</td>
<td>ababcd, abcdcdcd, ...</td>
</tr>
<tr>
<td>/(..)*/</td>
<td>&quot;&quot;, aa, aaaa, abcdef</td>
</tr>
<tr>
<td>/(a.*)/</td>
<td>(a), (abba), (alfred), (a star)</td>
</tr>
<tr>
<td>/(.+ .+)/</td>
<td>(one two), (a b), (zz top)</td>
</tr>
<tr>
<td>/.+(..+)/*</td>
<td>1, 1.1, 1.1.1, one.two.three</td>
</tr>
</tbody>
</table>
Alternatives

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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a</td>
<td>b/</td>
</tr>
<tr>
<td>/a</td>
<td>b+/</td>
</tr>
<tr>
<td>/(a</td>
<td>b]+)/</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a</td>
<td>b</td>
</tr>
<tr>
<td>/[abcde]/</td>
<td>a, b, c, d, e</td>
</tr>
<tr>
<td>/[a-e]/</td>
<td>a, b, c, d, e</td>
</tr>
<tr>
<td>/[a-z]*/</td>
<td>a, z, zzygy</td>
</tr>
<tr>
<td>/[0-9]+/</td>
<td>1, 2, 226, 191239891</td>
</tr>
<tr>
<td>/ [+-*/] /</td>
<td>+, -, *, /</td>
</tr>
</tbody>
</table>
Yeast IDs with Alternatives

Examples of using sets and alternatives in yeast gene IDs:
- `/YA(L|R).{3}(C|W)/`
  - any gene on chromosome 1
- `/Y.(L|R).{3}(C|W)/`
  - any gene
- `/Y[A–P][LR].{3}[CW]/`
  - any gene; a “tighter” specification since it doesn’t match YZR... etc
- `/Y[A–P][LR][0–9]{3}[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]`

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Example Matching Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/\w+/</code></td>
<td>Able, Baker, fred, …</td>
</tr>
<tr>
<td><code>/\d*/</code></td>
<td>&quot;&quot;, 12, 007, …</td>
</tr>
<tr>
<td><code>/\(\d+\s\d+/</code></td>
<td>(12 26), (0 1), (100 345), …</td>
</tr>
</tbody>
</table>
Character Classes (cont’d)

4 If \(^\) is the first character in a set, it negates the set, i.e. the construct means “anything but these characters”

4 Pattern Example Matching Strings
   - /a[^bc]x/ aax, adx, aex, ...
   - /[^[A-Z]]*/ abc, 123, ...

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

4 So far we’ve relied on intuition to define precedence
4 Most expressions we’ve seen are actually ambiguous
4 Examples:
   – /ab*/
   – “a followed by zero or more b’s” or “zero or more pairs of ab”?
   – /one|two/
   – “one or two” or “onewo or ontwo”?
   – /ab?c/
   – is it b that is optional, or ab?
4 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>* + ?</td>
<td>{n,m}</td>
</tr>
<tr>
<td>Characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alternative</td>
<td></td>
<td>lowest</td>
</tr>
</tbody>
</table>

4 Use parentheses (groups) to alter precedence

- `/ (ab)+/` one or more ab pairs
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.
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”

```plaintext
if ($line !~ /pattern/) {
    ...
}
```

is the same as

```plaintext
if (!$line =~ /pattern/) {
    ...
}
```

is the same as

```plaintext
if (!$($line =~ /pattern/)) {
    ...
}
```
=~ Matches Substrings

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

4
- "abc" =~ /abc/
- "ababcab" =~ /abc/
- "abacaba" =~ /abc/
- "abbababc" =~ /abc/
Extracting Values

4 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

4 Example:
- $line = "gi|6319950  YCR106W ...";
- ($id) = ($line =~ /(Y[A-P][LR]\d{3}[CW])/);
- The variable $id is assigned the value "YCR106W"
Use Groups to Define Extracted Values

4 Perl uses groups to define the parts of patterns that are extracted
4 Each time Perl matches a pattern inside parentheses, it assigns the portion of the string that matches the pattern to a special variable
4 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:

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

```perl
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.

```perl
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";
}
```
Value of the =~ Operator (cont’d)

4 If the match fails, the list is empty
4 An empty list is the same as undef, or false
4

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

4
Anchors

4 Use ^ or $ to “anchor” a pattern at the beginning or ending of a string

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

4 Anchor symbols have the same precedence as literal characters
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.

```perl
$s = "gi\|22652\|ref\|NP_00001.1 hypothetical";

$id = ($s =~ /\|(.*?)\|/);
$id == "22652\|ref"
$id = ($s =~ /\|(.*?))\|/);
$id == "22652"
```
Variables in Patterns

4 So far all the pattern examples have been literal strings
4 It is possible to include variables in patterns, e.g.
   - $target = 226;
   - while ($s = <IN>) {
       if ($s =~ /^gi|$target/) {
         ...
       }
   }
4 These patterns can be inefficient, since the Perl compiler preprocesses patterns, and these must be processed at runtime
Substitutions

4 Regular expressions can be used to modify strings
4 The expression
   – $s =~ s/pat/rep/
4 means “replace pat by rep in the string $s”
4 Example:
   – $s = "hello, world";
   – $s =~ s/hello/goodbye/;
   – print "s = "$s"
   – s = 'goodbye, world'
Further Reading

4 The topics covered in this lecture will get you through the Perl projects this term
4 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)