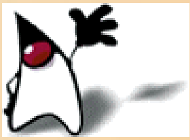
Regular Expressions



Regular Expressions

- A regular expression is a kind of pattern that can be applied to text (`String`s)
- A regular expression either matches the text (or part of the text), or it fails to match
  - If a regular expression matches a part of the text, then you can easily **find** out which part
  - If a regular expression is complex, then you can easily find out which parts of the regular expression match which parts of the text
  - With this information, you can readily extract parts of the text, or do substitutions in the text
- Regular expressions are extremely useful for manipulating text
  - Regular expressions are used in the automatic generation of Web pages

Perl

- The Perl programming language is heavily used in server-side programming, because
  - Much server-side programming is text manipulation
  - Regular expressions are built into the syntax of Perl

- Regular expressions are easier and more convenient in Perl

A first example

- The regular expression "[a-z]+" will match a sequence of one or more lowercase letters
  - [a-z] means any character from a through z, inclusive
  - + means "one or more"
- Suppose we **apply this pattern** to the String "Now is the time"
  - There are *three ways* we can apply this pattern:
    - To the *entire string:* it fails to match because the string contains characters other than lowercase letters
    - To the *beginning of the string:* it fails to match because the string does not begin with a lowercase letter
    - To *search the string:* it will succeed and match ow
      - If the pattern is applied a second time, it will find is
      - Further applications will find is, then the, then time
      - After time, another application will fail

Some simple patterns

abc	exactly this sequence of three letters
[abc]	any *one* of the letters a, b, or c
[^abc]	any character *except* one of the letters a, b, or c (immediately within an open bracket, ^ means "not," but anywhere else it just means the character ^)
[a-z]	any *one* character from a through z, inclusive
[a-zA-Z0-9]	any *one* letter or digit

Sequences and alternatives

- If one pattern is followed by another, the two patterns must match consecutively
  - For example, `[A-Za-z]+[0-9]` will match one or more letters immediately followed by one digit
- The vertical bar, `|`, is used to separate alternatives
  - For example, the pattern `abc|xyz` will match either `abc` or `xyz`

Some predefined character classes

.	any one character except a line terminator
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\x0B\f\r]
\S	a non-whitespace character: [^\s]
\w	a word character: [a-zA-Z_0-9]
\W	a non-word character: [^\w]

Notice the space.
Spaces are **significant**
in regular expressions!

Boundary matchers

- These patterns match the *empty string* if at the specified position:

^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	not a word boundary
\A	the beginning of the input (can be multiple lines)
\Z	the end of the input except for the final terminator, if any
\z	the end of the input
\G	the end of the previous match

Greedy quantifiers

(The term “**greedy**” will be explained later)

Assume *X* represents some pattern

X?	optional, *X* occurs once or not at all
*X**	*X* occurs zero or more times
X+	*X* occurs one or more times
X{*n*}	*X* occurs exactly *n* times
X{*n*,}	*X* occurs *n* or more times
X{*n*,*m*}	*X* occurs at least *n* but not more than *m* times

Note that these are all *postfix* operators, that is, they come *after* the operand

Types of quantifiers

- A greedy quantifier will match as much as it can, and back off if it needs to
  - We’ll do examples in a moment
- A reluctant quantifier will match as little as possible, then take more if it needs to
  - You make a quantifier reluctant by appending a ?:
    X?? *X**? *X*+? *X*{*n*}? *X*{*n*,}? *X*{*n*,*m*}?
- A possessive quantifier will match as much as it can, and never let go
  - You make a quantifier possessive by appending a +:
    X?+ *X**+ *X*++ *X*{*n*}+ *X*{*n*,}+ *X*{*n*,*m*}+

Types of quantifiers

- In some implementations, a quantifier in regular expressions corresponds to the maximum line length is possible. This can be a significant problem.
  - For example, often expect that the expression (<.*>) find in the text tag HTML. However, if the text is more than one HTML-tag, this expression matches the entire string containing a set of tags.

```
<p><b>Beginning with bold text</b> next, text body,
<i>italic text</i> end of text.</p>
```

- This problem can be solved in two ways
  - Take into account characters that are not relevant to the desired **pattern** (**<[^>]*>** for the above case).
  - Determine how non-greedy quantifier (**lazy**) - most implementations allow you to do this by adding after the question mark.

  * **->** *?   + **->** +?   {n,} **->** {n,}?

Quantifier examples

- Suppose your text is aardvark
  - Using the pattern a*ardvark (a* is greedy):
    - The a* will first match aa, but then ardvark won't match
    - The a* then "backs off" and matches only a single a, allowing the rest of the pattern (ardvark) to succeed
  - Using the pattern a*?ardvark (a*? is reluctant):
    - The a*? will first match zero characters (the null string), but then ardvark won't match
    - The a*? then extends and matches the first a, allowing the rest of the pattern (ardvark) to succeed
  - Using the pattern a*+ardvark (a*+ is possessive):
    - The a*+ will match the aa, and will not back off, so ardvark never matches and the pattern match fails

Capturing groups

- In regular expressions, parentheses are used for **grouping**, but they also capture (keep for later use) anything matched by that part of the pattern
  - Example: `([a-zA-Z]*)([0-9]*)` matches any number of letters followed by any number of digits
  - If the match succeeds, `\1` holds the matched letters and `\2` holds the matched digits
  - In addition, `\0` holds everything matched by the entire pattern
- Capturing groups are numbered by counting their *opening parentheses* from left to right:
  - ```
    ( ( A ) ( B ( C ) ) )
    1 2     3   4
    \0 = \1 = ((A)(B(C))),    \2 = (A),    \3 = (B(C)),    \4 = (C)
    ```
- Example: `([a-zA-Z])\1` will match a double letter, such as letter
    ```

Pig Latin

- Pig Latin is a spoken “secret code” that many English-speaking children learn
  - There are some minor variations (regional dialects?)
- The rules for (written) Pig Latin are:
  - If a word begins with a consonant cluster, move it to the end and add “ay”
  - If a word begins with a vowel, add “hay” to the end
  - Example:
    regular expressions are fun! →
    egularray expressionshay arehay unfay!

Example use of capturing groups

- Suppose word holds a word in English
- Also suppose we want to move all the consonants at the beginning of word (if any) to the end of the word (so string becomes ingstr)
  - ```
    Pattern p = Pattern.compile("([^aeiou]*)(.*)");
    Matcher m = p.matcher(word);
    if (m.matches()) {
      System.out.println(m.group(2) + m.group(1));
    }
    ```
- Note the use of (.*) to indicate "all the rest of the characters"
    ```

Pig Latin translator

- ```
  Pattern wordPlusStuff =
       Pattern.compile("([a-zA-Z]+)([^a-zA-Z]*)");
  Pattern consonantsPlusRest =
       Pattern.compile("([^aeiouAEIOU]+)([a-zA-Z]*)");
  ```

- ```
  public String translate(String text) {
     Matcher m = wordPlusStuff.matcher(text);
     String translatedText = "";

     while (m.find()) {
        translatedText += translateWord(m.group(1)) + m.group(2);
     }
     return translatedText;
  }
  ```

- ```
  private String translateWord(String word) {
     Matcher m = consonantsPlusRest.matcher(word);
     if (m.matches()) {
        return m.group(2) + m.group(1) + "ay";
     }
     else return word + "hay";
  }
  ```
  ```

Double backslashes

- Backslashes have a special meaning in regular expressions; for example, `\b` means a word boundary
- The compiler treats backslashes specially; for example, `\b` in a String or as a char means the backspace character
- Syntax rules apply first!
  - If you write `"\b[a-z]+\b"` you get a string with backspace characters in it--this is *not* what you want!
  - Remember, you can quote a backslash with another backslash, so `"\\b[a-z]+\\b"` gives the correct string
- Note: if you *read in* a String from somewhere, you are not *compiling* it, so you get whatever characters are actually there

Escaping metacharacters

- A lot of special characters--parentheses, brackets, braces, stars, plus signs, etc.--are used in defining regular expressions; these are called metacharacters
- Suppose you want to search for the character sequence `a*` (an `a` followed by a star)
  - `"a*"`; doesn't work; that means "zero or more `a`s"
  - `"a\*"`; doesn't work; since a star doesn't *need* to be escaped (String constants), Code just ignores the `\`
  - `"a\\*"` *does* work; it's the three-character string `a`, `\`, `*`
- Just to make things even more difficult, it's *illegal* to escape a *non*-metacharacter in a regular expression
  - Hence, you can't backslash special characters "just in case"

Spaces

- There is only one thing to be said about spaces (blanks) in regular expressions, but it's important:
  - *Spaces are significant!*
- A space stands for a *space*--when you put a space in a pattern, that means to match a space in the text string
- It's a *really bad idea* to put spaces in a regular expression just to make it look better

Regular expressions are a language

- Regular expressions are *not* easy to use at first
  - It's a bunch of punctuation, not words
  - The individual pieces are not hard, but it takes practice to learn to put them together correctly
  - Regular expressions form a miniature programming language
    - It's a different kind of programming language, and requires you to learn new thought patterns
  - You have to first create Patterns and Matchers
  - Syntax for String constants doesn't help, either
- Despite all this, regular expressions bring so much power and convenience to String manipulation that they are well worth the effort of learning

Thinking in regular expressions

- The fundamental concept in regular expressions is automatic backtracking
  - You match the parts of a pattern left to right
    - Some pattern parts, such as `x` (the letter "x"), `.` (any one character), and `^` (the beginning of the string) are deterministic: they either match or don't match; there are no other alternatives to try
    - Other pattern parts are nondeterministic: they have alternatives, such as `x*` (zero or more letter "x"s), `x+` (one or more letter "x"s), `[aeiou]` (any vowel), and `yes|no` (either "yes" or "no")
  - If some part fails to match, you backtrack to the most recent nondeterministic part and look for a different match for that part

Backtracking examples

- Search cases for a [aeiou]s$, that is, a vowel followed by an "s" at the end of the string
  - [aeiou] doesn't match c
  - [aeiou] matches a, s matches s, $ fails
    - There is no other possible match for s in this position
  - [aeiou] doesn't match s
  - [aeiou] matches a, s matches s, $ succeeds

Hazards of regular expressions

- Regular expressions are complex
  - They are often used when you cannot guarantee “good” input, so you have to make them fail-safe
- Backtracking can be extremely expensive
  - Avoid **.*** and other highly nondeterministic patterns
  - Test with non-trivial data to make sure your patterns scale
- Test thoroughly!
  - Break a complex regular expression into its components, and test each separately
    - Every pattern is a *program*, and needs to be treated with respect
  - Pay special attention to edge cases
- Consider alternatives
  - Regular expressions are powerful, **but**... If you can get the job done with a few **simple String methods**, you probably are better off doing it that way