Pragmatic Regular Expressions

Matt Sparks

ACM Student Lecture Series, UIUC
December 4, 2008

Pattern Matching

... just applying a *pattern* against a string or block of text. The match can succeed or fail.

- Substring matching in simple text editors
  → find `bar` in `foo bar baz`

Pattern Matching

... just applying a *pattern* against a string or block of text. The match can succeed or fail.

- Substring matching in simple text editors
  → find `bar` in `foo bar baz`
- Wildcard matching (shell globbing)
  → `ls *.c`

Pattern Matching

... just applying a *pattern* against a string or block of text. The match can succeed or fail.

- Substring matching in simple text editors
  → find `bar` in `foo bar baz`
- Wildcard matching (shell globbing)
  → `ls *.c`
- Regular expressions

Regular Expressions

Powerful pattern matching beyond wildcards:

- `bar`
- `.*?\.c$`
- `^[cC]at(?:fish)? (?:dinner|meal)$`
- `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b`

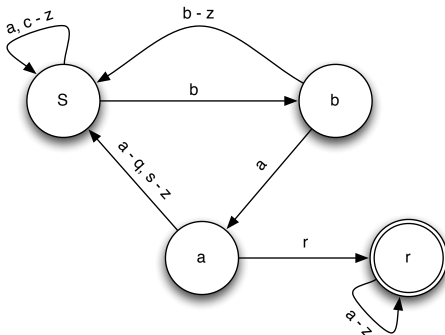
Why are regexes useful?

Regular expressions allow precise definition of complex patterns easily.

- Good for validation and data extraction
  - form input
  - log parsing
- Substantial presence
  - programming languages
  - text editors
  - Google Code Search
- Not just for programmers

Theory

Pattern matching engines are just fancy state machines.



.. but *modern* regexes are more powerful (backreferencing)

Regex Flavors

- POSIX Basic Regular Expressions (BRE)
  - grep
  - sed
- POSIX Extended Regular Expressions (ERE)
  - grep -E / egrep
  - sed -E
  - PHP ereg()
- Perl-Compatible Regular Expressions (PCRE) / Perl-like
  - Perl
  - Python
  - Ruby
  - PHP preg()

Lots of gory details in the differences between these. For this talk, I'll be discussing a *Perl-like* syntax.

Regex Powertoy

For interactive experiments:

`http://regex.powertoy.org/`

Character Classes

A state machine transitions on input characters. A character class define the characters allowed on the transition.

- Match a single character
- Specified with square brackets: `[aeiou]`
- Ranges: `[a-fG-Z0-9+]` (only special characters are `] - ^ \`)
- Negation with caret: `[^ ]`
- Shorthand classes: `\w` = `[a-zA-Z0-9_]`, `\d` = `[0-9]`, `\s` = `[ \t\r\n]` (negations: `\W`, `\D`, `\S`)
- period (`.`) matches any character except newlines (by default)
- Example: `gr[ae]y` matches only `gray` and `grey`

Anchors

'Anchor' the regex to the beginning or end.

- `^`: match beginning of line
- `$`: match end of line
- "Multiline mode" alters behavior slightly
- `\A`: match beginning of input
- `\Z`: match end of input

Examples:

- strings ending in '.c': `\.c$`
- input line is 'foo': `^foo$`
- input line begins with 'mp2', 'mp6', 'mp0', . . . : `^mp\d`

Word boundaries

A word boundary occurs at the beginning and end of every 'word'.

- A 'word' is `[a-zA-Z0-9_]+`
- `\b` matches a *word boundary*
- `\b` is a zero-length match!
- Match whole word: `\bfoo\b`

```
\b         \b           \b
  regular  expressions
```

Alternation: logical OR

- Match one in a set: `cat|dog|mouse|fish`
- Match whole word, 'cat' or 'dog': `\b(cat|dog)\b`
- Lowest precedence: `^gr[ae]y|black$` matches lines that begin with `gray`/`grey` OR lines that end with `black`.
- `^(gr[ae]y|black)$` matches whole lines: `gray`, `grey`, `black`.
- `gr[ae]y` = `gr(a|e)y`

Quantification and repetition: how many?

- Quantifiers: `*`, `+`, `?`
- Quantifiers are placed after a regex *token* to specify how the token may be present
- Star (`*`): zero or more times
- Plus (`+`): one or more times
- Question mark (`?`): exactly zero or one times
- Limiters: `{`*min*`, `*[max]*`}`

Examples:

- `foo*` = `fo+`: `f` followed by any number of `o`s
- `cat(fish)?`: `cat` or `catfish`
- `qu{2,4}`: `quux`, `quuux`, `quuuux`

Greediness

Regexes are greedy. They match the longest string possible.

For string the string `one [two] [three]`,

- `\[.+\]` will match `[two] [three]`, not `[two]`
- Add question mark to a quantifier to make it *non-greedy* or *lazy*: `\[.+?\]` will match `[two]`
- Laziness is often desired.

Grouping, capturing, and backreferences

- Using parentheses creates a group:
  `the (.+) jumped over the (.+) fence`
- Demo
- Use `?:` to prevent capturing: `(?:foo|bar)`
- Capturing creates backreferences
- Demo
- Another example: `\b(\w+)\s+\1\b` matches doubled words (`hello hello`)

Modifiers: matching modes

- Modes change how matching is done
- `i`, `s`, `m`, `x`: specified after a pattern or with function arguments (e.g., Python, Java)
- `i`: case insensitivity
- `s`: single-line mode; periods match newlines
- `m`: multi-line mode; affects `^` and `$`
- `x`: free-spacing mode; ignore whitespace between tokens, allow comments
- selective modes: `(?i)te(?-i)st` = `(?i:te)st`: match `Test`, `TEst`, but not `teST`.
- Syntax is like non-capturing groups: `(?:foo)`

Lookahead and lookbehind

- Make assertions about what has come or will come
- Zero-width assertions: lookarounds do not consume characters!
- Positive lookahead: `q(?=u)`
- Negative lookahead: `q(?!u)`
- Positive lookbehind: `(?<!a)b`
- Negative lookbehind: `(?<=a)b`

Example: 6-letter word containing ‘cat’

- 6-letter word: `\b\w{6}\b`
- word containing cat: `\b\w*cat\w*\b`
- combine with lookahead: `(?=\b\w{6}\b)\b\w*cat\w*\b`

Conclusion and further reading

- Regexes are powerful, and (mostly) easy!
- . . . but sometimes tricky
- Outline and several examples from:
  http://www.regular-expressions.info