Regex Pattern Matching in Programming Languages

By Xingyu Wang

Outline

- Definition of Regular Expressions
- Brief History of Development of Regular Expressions
- Introduction to Regular Expressions in Perl

What a Regular Expression Is?

A regular expression is a sequence of characters that forms a pattern which can be matched by a certain class of strings.

A character in a regular expression is either a normal character with its literal meaning, or a metacharacter with a special meaning. Some commonly-used metacharacters include:

. ? + * () [] ^ $

For many versions of regular expressions, one can escape a metacharacter by preceding it by \ to get its literal meaning.

For different versions of regular expressions, metacharacters may have different meanings.

Brief History

- It was originated in 1956, when the mathematician Stephen Kleene described regular languages using his mathematical notations called *regular sets*.
- Around 1968, Ken Thompson built Kleene's notations into the text editor called QED ("quick editor"), which formed the basis for the UNIX editor *ed*.
- One of *ed*'s commands, "g/regular expression/p" ("global regular expression print"), which does a global search in the given text file and prints the lines matching the regular expression provided, was made into its own utility *grep*.

- Compared to *grep*, *egrep*, which was developed by Alfred Aho, provided a richer set of metacharacters. For example, + and ? were added and they could be applied to parenthesized expressions. Alternation | was added as well. The line anchors ^ and $ could be used almost anywhere in a regular expression.
- At the same time (1970s), many other programs associated with regular expressions were created and evolved at their own pace, such as *AWK*, *lex*, and *sed*.
- POSIX (Portable Operating System Interface), which appeared first in the 1980s, tried to provide standardization for the various programs dealing with regular expressions. It defined two classes, Basic Regular Expressions (BREs) and Extended Regular Expressions (EREs).

An Overview of POSIX Regular Expression Classes

Regex Features	BREs	EREs
dot, ^, &, [...], [^...]	yes	yes
“any number” quantifier	*	*
+ and ? quantifier		yes
range quantifier	\{min, max\}	{min, max}
grouping	\(...\)	(...)
applying quantifiers to (...)	yes	yes
backreferences	\1 through \9	
alternation		yes

- In 1986, a regular expression package was released by Henry Spencer, which could be freely incorporated by others into their own programs. An enhanced version of the package was used for regular expressions in Perl 2.
- First developed in the late 1980s, Perl is a powerful scripting language. Many of its concepts of text handling and regular expressions were derived from AWK, *sed*, and the Spencer's package. As several new versions were released over the years, Perl provides a much richer set of metacharacters and more regular expression features.
- Regular expressions today are widely used and supported in programming languages, text editors, and text processing tools. Programming languages like Perl and AWK, provides built-in regular expression capabilities, while others like Java and Python, support regular expressions through standard libraries.

Perl Regex Features (version 5.8)

- ❏ Metacharacters
- ❏ Regex-Related Operators
- ❏ Modifiers
- ❏ Grouping, Capturing, and Control
- ❏ After-Match Variables
- ❏ Zero-Width Tests

Metacharacters

Perl supports a rich set of metacharacters, many of which were not supported by other languages previously. Over the years, other systems have adopted many of Perl's innovations.

Some examples of Perl-specific metacharacters include:
\w, \W, \d, \D, \s, \S, \b
(…) and \1, \2, etc
(?:)

Metacharacter may have different meanings inside or outside character class […]. For example, \b, and ^.

Regex-Related Operators

- m/regular expression/optional modifiers

  The regular-expression match operator takes two operands, a target string operand and a regex operand. The m indicates “try a regular expression match”.

  A basic match looks like:

  $target_string =~ m/regex/

  The =~ links m/.../ with the target_string to be searched. If the match is successful, the entire expression will return true; otherwise, false.

- s/regular expression/replacement/optional modifiers

  Compared to the match operator, the substitution operator takes a further step. It is usually used in the form of:

  $target_string =~ s/regex/replacement/

  If the regex is able to match the text somewhere in the target_string, the text matched will be replaced by the string stored in the replacement operand.

- qr/regular expression/optional modifiers

  The qr/.../ operator is a unary operator that takes a regular expression operand and returns a regular expression object. The returned object can be used as a regular expression operand of match, substitution, and split. It can also be used as a subexpression of a large regular expression.

  The qr/…/ operator supports the five core modifiers. Once a regex object is created using qr/…/ with some modifiers specifying some match modes, the match modes of the regular expression represented by the object cannot be changed, even if later the regex object is used in m/…/ with its own modifiers.

- split(...)

  The split operator is often used as the converse of the m/…/g operator.

  Example:

  $text = “10/30/2014”;

  $text ~= m/\//g results in two successful match.

  split(/\//, $text) returns a list (‘10’, ‘30’, ‘35’). The ‘/’ is matched twice in $text, and it partitions $text into three parts.

Modifiers

Five Core Modifiers Available to All Regex Operators

- i

  Ignore letter case during matching.

  Example: $var =~ m/ab/i

  The expression above will return true if the text in var contains ab, aB, Ab, or AB.

- x

  Free-spacing and comments regex mode.

  In this mode, whitespace outside character classes is ignored, and whitespace within a character class still counts. Comments are allowed between a # and a newline.

- o

  Compile only once. It is associated with efficiency issue.

- s

  Dot-matches-all match mode.

  Usually, a dot does not match a newline.
- m

  Enhanced line-anchor match mode.

  It influences where the line anchors, ^ and $, match.

More Modifiers

- g

  Global replacement. After the first match and replacement, try to find more matches and to make more replacements.

  Example: % perl –p –i –e ‘s/sysread/read/g’ filename

When more than one modifiers are needed, just combine them in any order and append them to the closing delimiter of the regex operator.

Example:

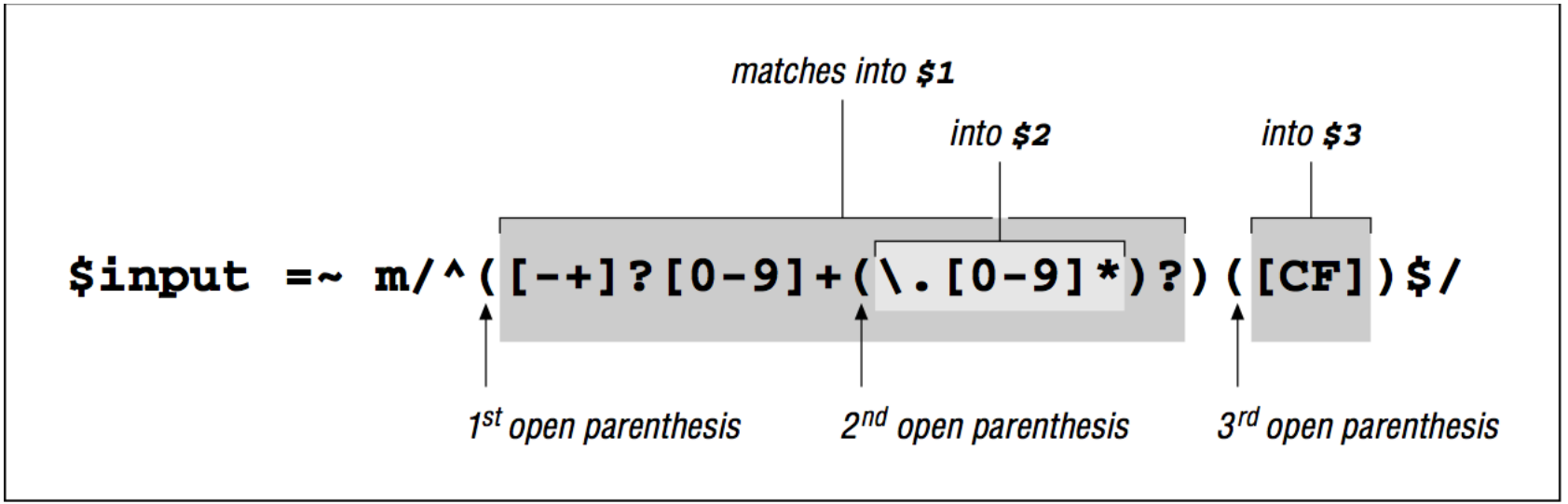
```
$var =~ s/Mike/Michael/ig
```

Grouping, Capturing, and Control

- Capturing Parentheses
  - (...) and \1, \2, etc

    Capturing parentheses are numbered by counting their opening parentheses from the left. Since *backreferences* are available in Perl, the text matched via an enclosed subexpression can itself be matched later in the same regular expression with \1, \2, etc. \1, \2 can only be used in regular expressions.
  - $1, $2, etc

    After a successful match, Perl provides the variables $1, $2, $3, etc., which hold the text matched by their respective parenthesized subexpressions in the regex.


matches into $1
into $2
into $3
$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)([CF])$/
1st open parenthesis
2nd open parenthesis
3rd open parenthesis

- Grouping-Only Parentheses
  (?: ...)
- Alternation
  regex_1|regex-2|…
- Greedy Quantifiers
  ?, +, *, {min, max}
- Lazy Quantifiers
  ??, +?, *?, {min, max}?
  Normally, the quantifiers are greedy, so they try to match as match as possible. Conversely, the lazy quantifiers try to match as little as possible.

After-Match Variables

- Grouping and Capturing
  Discussed previously.
- $^N
  It stores a copy of the most-recently-closed $1, $2, etc. It is explicitly set during the match.
- $+
  It stores a copy of the highest numbered $1, $2, etc. It is explicitly set during the match. It is helpful to avoid use of undefined variables.

- $&
  A copy of the text successfully matched by the regex. After a successful match, it is never undefined.

- $`
  A copy of the target text to the left of the match's start.
- $'
  A copy of the target text to the right of the matched text.

After a successful match, the string "$`$&$'" is always a copy of the original target string.

Zero-Width Tests

- Perl supports the enhanced line-anchor match mode. It influences where the line anchors, ^ and $, match. The anchor ^ usually does not match at embedded newlines.
- Start of line/string
  - ^
    It matches at the beginning of the text being searched; if in the enhanced line-anchor match mode, it matches after any newline.
  - \A
    Regardless of match mode, it only matches at the start of the text being searched.

- End of line/string
  - $

    Normally, it matches at the end of the target string. It matches before string-ending newline as well.

    In the enhanced line-anchor mode, it matches at the end of string, or before any newline.
  - \Z

    It matches the end of the target string, or before string-ending newline. It always matches like normal.
  - \z

    It matches only the end of the string, without regard to any newline.

- End of previous match: \G
  It is the anchor to where the previous match ends.
- Word boundaries
  - word boundary
    \b
  - not word boundary
    \B
  - by look around
    start of word: (?<! \w)(?= \w)
    end of word: (?<= \w)(?! \w)

- Look around
  - Lookbehind
    - Positive
      (?<= ...), successful if the subexpression can match to the left
    - Negative
      (?<! ...), successful if the subexpression cannot match to the left
  - Lookahead
    - Positive
      (?= ...), successful if the subexpression can match to the right
    - Negative
      (?! ...), successful if the subexpression cannot match to the right

Future Work

- More Regex features in Perl including related pragmas, related functions, and related variables.
- More advanced use of regex operators in Perl.
- Dynamic scoping and expression context in Perl.
- How the regex engine in Perl works.
- New regex features added to newer versions of Perl.
- Regex pattern matching in other programming languages such as Java and Python.

References

1. Friedl, J. (2006). *Mastering Regular Expressions* (Third ed., p. 517). Sebastopol: O'Reilly Media.
2. www.regular-expressions.info
3. http://en.wikipedia.org/wiki/Regular_expression#History
4. www.cs.cmu.edu/afs/cs/usr/rgs/mosaic/pl-regex.html