Bab 3

# Regex di Berbagai Bahasa

Bab ini membahas fasilitas regex di berbagai bahasa, lingkungan, dan aplikasi. Termasuk perbedaan dan fitur unik masing-masing dibandingkan dengan regex ala Perl.

# Perl

Regex amat sentral perannya di Perl. Nyaris semua program Perl mengandung atau menggunakan regex, dan jika seorang programer Perl tidak akrab atau fasih dengan regex, bisa diragukan apakah ia seorang programer Perl yang baik atau bukan.

Di Perl, ada lima operator yang berhubungan dengan regex. Yakni `m//`, `qr//`, `s///`, `=~`, `!~`. Dua operator pertama (`m//` dan `qr//`) adalah operator yang dipakai untuk mengapit dan mengkompilasi sebuah pola regex, operator ketiga dan keempat (`=~` dan `!~`) digunakan untuk pencocokan atau substitusi regex dengan sebuah string, sementara operator terakhir (`s///`) digunakan untuk substitusi regex.

Perlu dicatat bahwa pola regex tidak ditulis dalam bentuk string biasa, misalnya `"ka[km]u"`). Melainkan harus dikutip oleh salah satu dari kedua operator pengutip regex yang telah disebutkan di atas. Misalnya `m/ka[km]u/` atau `qr/ka[km]u/`) karena operator-operator pengapit regex tersebut sekaligus berfungsi untuk mengkompilasi pola regex sehingga dapat dipakai.

Selain keenam operator tersebut, ada fungsi-fungsi lain yang berhubungan dengan regex yaitu `quotemeta()`, `split()`, dan `grep()`.

# Pencocokan Regex

Untuk melakukan pencocokan regex, pertama-tama kita menulis regex dengan `m//` atau `qr//`. `m//` biasanya digunakan jika kita ingin langsung melakukan pencocokan bersama dengan operator `=~` atau `!~`, sementara `qr//` biasanya digunakan jika kita ingin menyimpan dulu pola regex ke dalam variabel.

```
if ($str =~ m/(\w+)-\1/) { print "String mengandung kata
ulang!\n"; }

$pola = qr/(\w+)-\1/;
if ($str =~ $pola) { print "String mengandung kata ulang!\n"; }
```

Perl memberi fleksibilitas yaitu tanda `//` dapat kita ganti dengan pasangan karakter lain. Karakter lain ini bukan angka, huruf, atau spasi. Misalnya: `##`, `{}`, `[]`, `<>`, `!!`, `@@`, `()`, dan lain sebagainya.

"`m`" pada `m//` bersifat opsional jika karakter pengapitnya adalah `//`, sehingga `m//` dapat ditulis `//` saja, tapi `m##` atau `m{}` tidak dapat ditulis dengan `##` atau `{}` saja.





```
if ($str =~ /(\w+)-\1/) { print "String mengandung kata
ulang!\n"; }
if ($str =~ m#(\w+)-\1#) { print "String mengandung kata
ulang!\n"; }
if ($str =~ m{(\w+)-\1}) { print "String mengandung kata
ulang!\n"; }
```

Karakter pengapit alternatif berguna jika pola regex Anda mengandung salah satu karakter pengapit. Misalnya, dalam mencocokkan HTML atau nama path atau URL, akan sering ditemukan karakter / (dalam tag penutup `</foo>` atau dalam karakter pemisah path/URL `/home/steven/foo.txt`).

Karena itu dalam mencocokkan pola-pola seperti ini, jika kita menggunakan karakter pengapit `//` maka / dalam pola harus di-escape menjadi `\/` . Meskipun / bukanlah karakter meta, tetapi jika tidak di-escape, maka Perl akan kebingungan membedakan mana karakter penutup pola:

```
if ($path !~ /^\//) {
  die "Path bukan path absolut! Path absolut diawali '/'\n";
}
```

Tapi jika menggunakan karakter pengapit `m##` atau `qr##`, / tidak perlu di-escape.

```
if ($path !~ m#^/#) {
  die "Path bukan path absolut! Path absolut diawali '/'\n";
}
```

Terlalu banyaknya escape pada / bisa menyebabkan munculnya banyaknya string seperti `\/` atau `\/\/`. Hal ini akan mengganggu pandangan dan membuat regex semakin sulit dibaca—hal ini kadang-kadang dijuluki *toothpick syndrome* atau *backslashism.*

```
/^(https?|ftp):\/\/([A-Za-z0-9-\.]+(:\d+)?\//
```

Dengan menggunakan karakter pengapit alternatif, kita dapat menghindari hal ini.

```
m#^(https?|ftp)://([A-Za-z0-9-\.]+(:\d+)?/#
```

Tentu saja, jika pola regex mengandung `()` Anda harus menghindari karakter pengapit `()`, jika pola mengandung `{}` Anda harus menghindari karakter pengapit `{}`, dan lain sebagainya.

Masih ingat modifier? Modifier dapat kita sebutkan di belakang karakter pengapit:

```
# hasilnya "Tidak cocok"
if ("Hello world!" =~ /HELLO/) { print "Cocok" } else { print
"Tidak cocok" }

# hasilnya "Cocok"
if ("Hello world!" =~ /HELLO/i) { print "Cocok" } else { print
"Tidak cocok" }

# hasilnya "Tidak cocok"
if ("Satu\nDua" =~ /^Dua/) { print "Cocok" } else { print "Tidak
cocok" }

# hasilnya "Cocok"
if ("Satu\nDua" =~ /^Dua/m) { print "Cocok" } else { print "Tidak
cocok" }

# keduanya hasilnya "Cocok"
if ("Satu\nDua" =~ /^DUA/mi) { print "Cocok" } else { print
"Tidak cocok" }
if ("Satu\nDua" =~ /^DUA/im) { print "Cocok" } else { print
"Tidak cocok" }
```

Seperti telah muncul beberapa di atas, operator `=~` dan `!~` berguna untuk mengkaitkan string pada pola regex. Sintaks penggunaan operator ini:

```
STRING =~ POLA
STRING !~ POLA
```

Hasil operator `=~` adalah true jika *STRING* cocok dengan *POLA* dan false jika tidak, sementara operator `!~` kebalikannya. Dengan kata lain, `!~` ekivalen dan shortcut dari:

```
not (STRING =~ POLA)
```

Salah satu kesalahan pemula adalah kadang-kadang terbalik meletakkan *STRING* dan *POLA* (misalnya `/\d\d\d/ =~ "abc123"` padahal seharusnya `"abc123" =~ /\d\d\d/`). Perl tidak mengeluarkan pesan kesalahan atau warning, malah sebetulnya operator `=~` mau menerima string biasa di sebelah kanan dan otomatis mengubahnya menjadi regex, tapi Anda perlu mengingat bahwa di sebelah kanan operator haruslah selalu regex.





## Beberapa contoh pencocokan

```
print "Masukkan password: ";
$_ = <STDIN>;

if (/^$/) {
  print "Password tidak boleh kosong!\n";
}
if (/^\w{5,8}$/) {
  print "Password harus 5-8 karakter, terdiri dari hanya huruf,
angka, dan _!\n";
}
```

Sesuai sifat Perl yang amat ringkas, `STRING =~` dapat tidak disebutkan jika kita ingin mencocokkan regex pada string di dalam variabel khusus bernama `$_`.

Pada baris pertama dan kedua kita meminta user mengetikkan password dan menangkapnya di variabel `$_`. Jika kita hanya menekan Enter, maka `$_` akan berisi `"\n"` saja, sehingga pengecekan regex pertama akan benar (ingat, `$` akan cocok dengan akhir string atau posisi sebelum newline). Pengecekan regex untuk memastikan bahwa password berisi 5-8 karakter kata dan hanya itu saja. Jika kita tidak memberi jangkar, maka string seperti `–abcd1234567––` pun akan cocok. Pola akan cocok dengan `abcd1234`. Tapi jika digunakan jangkar, maka pola harus dicocokkan di awal string, sehingga keseluruhan string tidak akan cocok.

```
$teks = "saya bilang juga apa, ".
        "dengan regex pemrosesan teks jadi mudah!";

$n = 0;
while ($teks =~ /\w+/g) { $n++ }
print "Total = $n kata.\n";
```

Hasilnya adalah:

```
Total = 10 kata.
```

Kode di atas menghitung jumlah kata yang terdapat pada string. Perhatikan modifier `g` pada regex. Ini untuk memberitahu Perl agar mengingat proses pencocokan string sebelumnya sehingga dapat memulai dari titik sebelumnya. Pada kasus di atas, pertama-tama pola akan cocok dengan kata `"saya"`. Lalu `$n` ditambah satu dari 0 menjadi 1. Saat loop berikutnya, pencocokan akan mulai dari titik

terakhir dan tidak dari awal string sehingga kini cocok dengan "bilang", dan seterusnya. Setelah kata terakhir "mudah" dicocokkan, ketika loop berikutnya tidak ada lagi yang cocok dengan \w+ sehingga hasil pencocokan menjadi false dan loop berakhir. Jika kita tidak memakai modifier g, maka loop tidak akan pernah berakhir karena $teks selalu cocok dengan pola (tepatnya dengan "saya").

Sebagai catatan pinggir, dulu saat saya belajar Turbo Pascal yang tidak mengenal regex, terdapat contoh program untuk menghitung jumlah kata dalam teks/file. Panjangnya puluhan hingga seratusan baris, berisi loop yang menelusuri huruf per huruf. Kini dengan regex hal ini dapat dilakukan dalam beberapa baris saja, seperti yang didemonstrasikan pada contoh di atas!

```
1|#!/usr/bin/perl
 |
3|### ekstrak-heading.pl
 |
5|while (<>) {
6|  if (m#<h(\d+)>(.+?)</h\1>#i) {
7|    print "Found heading $1: $2\n";
8|  }
9|}
```

Kode di atas dapat digunakan untuk mengekstrak dan mencetak heading yang ada pada dokumen HTML. Perhatikan baris 6. Kita ingin mencocokkan regex dengan isi variabel $_ (yang diisi dengan baris yang diambil di baris 5). Kita menggunakan modifier i karena tag heading dapat ditulis dengan huruf besar (<H1>Ini Heading</H1>) atau huruf kecil (<h2>Ini subjudul</h2>). Kita juga menangkap dua buah match group. Selain mengeset \1, \2, dan seterusnya. yang dapat digunakan di dalam pola regex itu sendiri, Perl juga akan mengeset $1, $2, dan seterusnya. yang dapat digunakan di luar pola regex. Pada kasus di atas, kita menggunakan $1 dan $2 di baris 7.

Jika kita berikan dokumen HTML html1.html seperti berikut:

```
<html>
  <title>Test document 1</title>
  <h1>Subjudul 1</h1>
  bla bla bla bla ...
    <h2>Subsubjudul 1</h2>
    bla bla bla ...
    <h2>Subsubjudul 2</h2>
    bla bla bla ...
  <h1>Subjudul 2</h1>
  bla bla bla bla ...
</html>
```





Maka hasilnya adalah sebagai berikut:

```
$ ./ekstrak-heading.pl html1.pl
Found heading 1: Subjudul 1
Found heading 2: Subsubjudul 1
Found heading 2: Subsubjudul 2
Found heading 1: Subjudul 2
$ _
```

Perhatikan lagi baris 6:

```
if (m#<h(\d+)>(.+?)</h\1>#i) {
```

di Perl kita dapat langsung mengambil match group ke variabel dengan cara:

```
if (($level, $name) = m#<h(\d+)>(.+?)</h\1>#i) {
```

sehingga kita tidak perlu memakai $1 dan $2 lagi melainkan langsung $level dan $name. Jika string yang ingin dicocokkan dengan pola bukan $_:

```
if (($level, $name) = $line =~ m#<h(\d+)>(.+?)</h\1>#i) {
```

## Mengetahui posisi pencocokan

Setelah berhasil melakukan sebuah pencocokan regex pada sebuah string, Perl akan mengeset array @- dan @+ dengan posisi/ofset awal pola dan tiap match group yang ditangkap. @- akan berisi offset awal dan @+ offset akhir. Elemen pertama (indeks array 0) berisi offset keseluruhan pola, elemen kedua (indeks array 1) berisi offset match group pertama (jika ada), elemen ketiga (indeks array 2) berisi offset match group kedua (jika ada), dan seterusnya. Contoh:

```
$string = "Subratawangsapraditamaraja";
print "Pada '$string', ditemukan konsonan yang dijepit vokal a-a
sebagai berikut:\n";
while ($string =~ /a(.)a/g) {
  print "- pada posisi $-[0] s.d. $+[0]: huruf $1\n";
}
```

hasilnya:

```
Pada 'Subratawangsapraditamaraja', ditemukan konsonan yang
dijepit vokal a-a sebagai berikut:
- pada posisi 4 s.d. 7: huruf t
- pada posisi 19 s.d. 22: huruf m
- pada posisi 23 s.d. 26: huruf j
```

## Mengambil string sebelum dan sesudah pola

Selain $1, $2, $3, dan seterusnya. dan `@-`, `@+`, Perl juga menyediakan variabel-variabel khusus lain yaitu:

`$&` berisi keseluruhan string yang cocok dengan pola.
`$'` (dolar diikuti kutip terbalik) berisi string sebelum pola.
`$'` (dolar diikuti kutip tunggal) berisi string sesudah pola.

Contoh:

```
$string = "1234567890";
$string =~ /5../; # pencocokan tentu saja berhasil
print "\$& = $&\n";
print "\$` = $`\n";
print "\$' = $'\n";
```

hasilnya:

```
$& = 567
$' = 1234
$' = 890
```

Catatan: `$&`, `$'`, dan `$'` dapat berpotensi memperlambat eksekusi program Perl Anda, karena Perl harus menangkap string yang mungkin saja amat panjang jika string yang ingin Anda cocokkan panjang (isi file besar tertentu misalnya). Karena itu kalau bisa hindari menggunakan kedua variabel khusus ini. Misalnya, jika Anda ingin menangkap keseluruhan pola, Anda bisa mengapit pola dengan `()` dan mengambil hasilnya di $1.

```
/(5..)/
```

Kemudian, jika Anda butuh mengambil string sebelum pola, Anda bisa melihat variabel `$-[0]` dan menggunakan `substr()` untuk mengambil substring dari `$string`:





```
substr($string, 0, $-[0])
```

Untuk mengambil string setelah pola, Anda bisa menggunakan `$+[0]` dan `substr()`:

```
substr($string, $+[0])
```

Catatan 2: Pada pencocokan berikutnya, tentu saja variabel-variabel khusus ini akan disesuaikan atau direset. Misalnya:

```
"becak rusak" =~ /(.)(ak)/; # $1 akan berisi 'c', $2 'ak', $-[0]
2, dan seterusnya.
"satu sepatu" =~ /(.at)/;   # $1 akan berisi 'pat', $2 kosong, $-
[0] 7, dan seterusnya.
```

Karena itu jika masih butuh nilai lama, Anda perlu menyimpannya ke variabel biasa.

## Substitusi regex

Operator `s///` digunakan untuk substitusi regex. Sintaks operator ini:

```
STRING =~ s/POLA/SUBSTITUSI/MODIFIER
```

Seperti dapat dilihat, operator `=~` juga digunakan di sini untuk mengkaitkan string dengan pola regex. `STRING =~` dapat tidak usah disebutkan jika kita ingin melakukan substitusi string pada variabel khusus $.

Sama dengan operator `m//` dan `qr//`, karakter pengapit pada `s///` juga dapat diganti, misalnya:

```
s#...#...#g

s!...!...!is

s{...}{...}

s(...)
 (...)e
```

Jika karakter pengapit merupakan karakter berpasangan (seperti {} atau []), maka kita dapat memisahkan antara pasangan yang pertama dan kedua, seperti pada contoh terakhir.

Berikut ini beberapa contoh substitusi:

```
# 1. hilangkan whitespace di awal string $_
s/^\s+//;

# 2. hilangkan whitespace di akhir string $teks
$teks =~ s/^\s+$//;

# 3. tukar dua huruf pertama
s/(.)(.)/$2$1/;

# 4. hilangkan spasi ganda
s/\s{2,}/ /g;

# 5. buat satu spasi menjadi dua spasi
s/\n/\n\n/g;

# 6. hilangkan komentar HTML
s#<!–.*?–>##sg;

# 7. hilangkan nama direktori (/home/steven/foo.txt menjadi
foo.txt)
s#.+/##;
```

Seperti bisa Anda lihat pada contoh 3, di bagian sebelah kiri (*POLA*) Anda dapat menggunakan \1, \2, dan seterusnya seperti biasa, sementara di sebelah kanan (*SUBSTITUSI*) Anda dapat menggunakan $1, $2, dan lain sebagainya untuk menggunakan match group.

Pada contoh 4, 5, 6 kita menggunakan modifier g karena kita ingin mengganti *semua* pola yang ditemukan. Sementara pada contoh 1, 2, 3 kita hanya akan melakukan penggantian satu kali saja, yaitu pada pola yang pertama kali ditemukan mesin regex.

Pada contoh 5, meskipun bekerja dengan string multibaris kita tidak membutuhkan modifier s karena kita toh tidak menggunakan karakter meta . (titik). Modifier s hanya berurusan/relevan dengan karakter meta titik, seperti bisa kita lihat pada contoh 6.

Karena komentar HTML dapat terdiri dari beberapa baris, kita menggunakan modifier s. Jika tidak, komentar seperti:





```
<!– ini komentar.
     ini baris kedua komentar.
     ini baris ketiga –>
```

tidak akan cocok dengan pola.

Jika *POLA* tidak cocok, tentu saja tidak akan ada substitusi yang dilakukan. Jadi pada contoh 1 dan 2, jika sebuah string tidak mengandung spasi di awal atau di akhir maka tidak akan ada substitusi yang dilakukan (sebagaimana mestinya).

## Substitusi dengan eval (modifier e)

*SUBTITUSI* pada operator s/// tidak hanya dapat berupa string biasa, tapi kode Perl. Jika kita menggunakan modifier e (eval), maka Perl akan menganggap bagian kanan pada operator s/// adalah kode, dan mengkompile serta mengeksekusinya. Contoh:

```
$teks =~ s/(\d+)\+(\d+)/$2+$1/e;
print $teks;
```

Jika teks berisi:

```
Contoh ekspresi: 12+3.
```

maka setelah substitusi hasilnya:

```
Contoh ekspresi: 15.
```

Mengapa demikian? Karena pada bagian kanan substitusi, kita memiliki kode $2+$1 yang jika dikompilasi akan menghasilkan 15 karena $1 berisi 12 dan $2 berisi 3. Tanpa modifier e, hasilnya adalah:

```
Contoh ekspresi: 3+12.
```

Ini karena pada bagian kanan $2+$1 dianggap sebagai string biasa, yang setelah interpolasi $1 dan $2 hasilnya 3+12.
Kode di bagian kanan dapat berupa sembarang kode Perl, karena itu kita dapat menggunakan penggantian yang amat fleksibel dan ampuh. Contoh lain, misalnya kita ingin menandai setiap kata-kata yang potensial salah eja:

```
s[(\w+)]
  [cek_kata($1) ? $1 : "<font color=red><i>$1</i></font>" ]eg;
```

`cek_kata()` ceritanya adalah sebuah fungsi Perl yang akan mengecek kata dalam kamus dan mengembalikan true jika kata terdapat dalam kamus dan false jika kata tidak dikenali. Jika kata tidak dikenali, kita menambahkan font merah dan cetak miring pada kata tersebut.

## Split regex

Selain mencocokkan, mengekstrak match group, dan melakukan substitusi, ada satu operasi lagi yang sering dilakukan dengan regex, yaitu splitting (membelah-belah string). Sintaks fungsi `split()` di Perl:

```
split POLA, STRING
```

Hasilnya adalah array string hasil split. jika *STRING* tidak disebutkan, maka dianggap kita ingin men-split variabel `$_`. Jika *POLA* tidak disebutkan, maka dianggap *POLA* adalah `/\s+/s`. Jika *POLA* mengandung match group, maka match group akan disisipkan juga ke dalam hasil split. Jika *POLA* adalah regex kosong (`//`), maka dianggap kita ingin membelah per huruf. Beberapa contoh:

```
# belah kalimat menjadi kata
$kalimat = "Aku cinta padamu.";
@kata = split /\s+/, $kalimat;      # "Aku", "cinta", "padamu."

# belah kalimat menjadi kata, sertakan spasinya
$kalimat = "Aku cinta padamu.";
@kata = split /(\s+)/, $kalimat;    # "Aku", " ", "cinta", " ",
"padamu."

# belah kata menjadi per huruf
$kata = "Cinta";
@huruf = split //, $kata;           # "C", "i", "n", "t", "a"

# belah teks menjadi per baris
@lines = split /\n/, $teks;
```

## Mengescape string

Jika Anda ingin mencari substring seperti `M*A*S*H` pada string, maka Anda harus menggunakan pola seperti ini: `M\*A\*S\*H`. Dengan kata lain, karakter-karakter meta harus diescape dulu. Jika tidak, pola regex dapat saja cocok hanya dengan huruf `H` (karena `M*A*S*H` berarti "0 atau lebih huruf M, diikuti 0 atau lebih huruf A, diikuti 0 atau lebih huruf S, diikuti huruf H"). Fungsi `quotemeta()` dapat melakukan escaping ini untuk Anda:





```
$unquoted = "M*A*S*H";
$quoted = quotemeta($unquoted);
$string = "Acara TV M*A*S*H popular tahun 1970-an";
print $quoted, "\n";
```

Hasilnya:

```
M\*A\*S\*H
```

Jika Anda tidak ingin menggunakan fungsi quotemeta(), Anda dapat juga menggunakan \Q dan \E untuk mengapit string yang ingin dianggap literal. Misalnya:

```
$string =~ /\Q$unquoted\E/;
```

## Named capture

Seperti telah disebutkan pada Bab 2, Perl belum mendukung named capture. Namun jika Anda menginstal modul `Regexp::Fields`, maka Anda dapat pula menikmati named capture.
Contoh penggunaan:

```
use Regexp::Fields 'my';

my $rx = qr/Time: (?<jam>..):(?<menit>..):(?<detik>..)/;
if (/$rx/) {
  print "Waktu menunjukkan: $&{jam}:$&{menit}:$&{detik}\n";
      # atau: "Waktu menunjukkan : $jam:$menit:$detik\n";
      # atau: "The time was: $1:$2:$3\n";
}
```

Seperti dapat dilihat di baris 3, jika kita menggunakan pola tanpa *named capture*:

```
Time: (..):(..):(..)
```

Maka jam, menit, dan detik dapat kita ekstrak hanya dari variabel $1, $2, $3. Seandainya suatu hari pola kita ubah menjadi:

```
Date and time: (..)-(..)-(....) (..):(..):(..)
```

Kita harus juga mencari dan menyesuaikan baris-baris lain yang mengambil $1, $2, dan $3. Karena sekarang jam, menit, dan detik ada di $4, $5, $6.

Dengan named capture, kita dapat menulis pola sebagai berikut:

```
Time:  (?<jam>..):(?<menit>..):(?<detik>..)
```

Selain memperjelas pola dengan memberi label pada apa yang sebetulnya ingin ditangkap, jika suatu hari kita memodifikasi pola menjadi:

```
Date and time: (?<d>..)-(?<m>..)-(?<y>....)
(?<jam>..):(?<menit>..):(?<detik>..)
```

kita tidak perlu mengubah kode yang menangkap dan memanfaatkan match group karena jam misalnya tetap dapat diambil di `$&{'jam'}` atau juga pada `$jam` kita kita menggunakan `my` saat meload `Regexp::Fields`.
Bagi mereka yang sudah 'kepincut' dengan named capture, misalnya yang sudah merasakan enaknya named capture di .NET atau Mono, bisa menggunakan modul Perl ini untuk bisa menggunakan named capture juga di Perl.

### Modifier o

Perl menganggap pola dalam `m//` seperti string berkutip dua biasa dan menerima interpolasi variabel di dalamnya, misalnya:

```
$jam = "..";
$menit = "..";
$detik = "..";

# cari pola hh:mm:ss di file
while (<>) {
  @matches = /($jam:$menit:$detik)/g
    and print join ", ", @matches, "\n";
}
```

Kode di atas akan mencari pola `/(..:..:..)/` di dalam tiap baris file. Namun karena pola regex mengandung variabel (`$jam`, `$menit`, `$detik`) maka Perl akan melakukan kompilasi ulang variabel di tiap putaran loop karena Perl tidak tahu apakah variabel `$jam`, `$menit`, atau `$detik` telah berubah isinya (jika ya, maka pola tentu berubah dan harus dikompilasi ulang).

Anda dapat membantu Perl menghindari kompilasi regex berulang kali dengan menambahkan modifier `o` (Once).

```
/($jam:$menit:$detik)/og
```





Dengan modifier ini, maka Perl hanya mengkompilasi pola sekali saja. Di putaran loop berikutnya Perl akan menggunakan hasil kompilasi pertama kali, tak peduli apakah `$jam`, `$menit`, atau `$detik` telah berubah isinya. Karena itu Anda perlu berhati-hati jika menggunakan variabel dalam regex dan modifier `o`. Saya pernah terkena bug yang melibatkan ini saat mengubah skrip CGI menjadi FastCGI. Karena pola regex menggunakan modifier `o`, maka pola regex dari satu request web ke request berikutnya masih menggunakan variabel dari request yang pertama. Untuk kasus seperti ini, modifier `o` perlu dihilangkan atau pola regex perlu dibuat ulang di setiap awal request baru.

## PHP

PHP memiliki dua set fungsi regex. Yang pertama adalah fungsi-fungsi dengan prefiks `ereg_` seperti `ereg_match()`, `eregi_replace()`, dan lain sebagainya. Huruf e pada reg artinya extended. Artinya, fungsi-fungsi regex ini adalah fungsi regex flavor POSIX.  Buku ini sengaja tidak membahas fungsi-fungsi `ereg_*()` karena keterbatasn sintaks regex POSIX; melainkan membahas fungsi-fungsi dengan prefiks `preg_`, seperti `preg_match()` dan `preg_replace()`.  Fungsi-fungsi `preg_*()` menggunakan library PCRE yang mendekati sintaks Perl. Berikut ini fungsi-fungsi yang disediakan PHP untuk menggunakan PCRE beserta sintaksnya:

```
int preg_match(str pola, str subjek[, array matches[, flag]]);
int preg_match_all(str pola, str subjek, array matches[, int
order]);
mixed preg_replace(mixed pola, mixed pengganti, mixed subjek[,
int limit]);
mixed preg_replace_callback(mixed pola, callback callback,
                            mixed subjek[, int limit]);
array preg_split(str pola, str subjek[, int limit[, int flags]]);
string preg_quote(str s[, str kutip]);
array preg_grep(str pola, array input);
```

Catatan: Versi PHP yang saya gunakan adalah 4.2.2 dan 4.3.2. Setting `register_globals` On.

### Mencocokkan string

Fungsi `preg_match()` dan `preg_match_all()` digunakan untuk melakukan pencocokan regex. Fungsi ini setara dengan `m//` dan `m//g` di Perl. Contoh:

```
<?
if (preg_match("/(siemens|motorola) ([^,]+)/i",
```

```
                "Nokia 8310, Siemens S45, Samsung SGH 100",
                $matches)) {
    echo "HP pilihanku = $matches[1] $matches[2]<br>\n";
    echo "Sekali lagi, pilihanku = $matches[0]<br>\n";
}
?>
```

Hasilnya:

```
HP pilihanku = Siemens S45
Sekali lagi, pilihanku = Siemens S45
```

Pola regex disebutkan di argumen pertama. Anda dapat menggunakan string berkutip tunggal maupun kutip ganda; hanya saja, hati-hati menggunakan string kutip ganda karena variabel seperti `$foo` akan diinterpolasi oleh PHP di dalam string berkutip ganda. Sama seperti di Perl, pola regex dapat diapit dengan // atau ##, {}, dan lain sebagainya. sementara modifier disebutkan setelah pengapit.

`$matches[1]` sama dengan $1 di Perl, `$matches[2]` sama dengan $2 di Perl, dan seterusnya. Dengan kata lain, array `$matches` digunakan untuk menampung hasil tangkapan match group. `$matches[0]` sama dengan $& di Perl yaitu untuk menangkap isi string yang cocok dengan keseluruhan pola. Berbeda dengan di Perl, `$matches[0]` selalu disediakan jadi tidak relevan dengan memperlamban program seperti kasus $& pada Perl.

Anda dapat tidak menyebutkan argumen `$matches` jika tidak ingin mengambil hasil match group (misalnya, Anda hanya ingin mengetes apakah `$string` cocok dengan `$pola`; dengan kata lain, hanya membutuhkan jawaban true atau false):

```
if (preg_match($pola, $string)) echo "Cocok!";
```

Untuk mengambil semua pola yang cocok seperti fungsi `m//g` pada Perl, Anda menggunakan `preg_match_all()`. Contoh:

```
<pre>
<?
$string = "(1,2), (3,4), (7,1), (11,4), (5,7), (10,0)";
preg_match_all("/\((\d+),(\d+)\)/", $string, $matches);
print_r($matches);
?>
</pre>
```





Hasilnya:

```
Array
(
    [0] => Array
        (
            [0] => (1,2)
            [1] => (3,4)
            [2] => (7,1)
            [3] => (11,4)
            [4] => (5,7)
            [5] => (10,0)
        )

    [1] => Array
        (
            [0] => 1
            [1] => 3
            [2] => 7
            [3] => 11
            [4] => 5
            [5] => 10
        )

    [2] => Array
        (
            [0] => 2
            [1] => 4
            [2] => 1
            [3] => 4
            [4] => 7
            [5] => 0
        )

)
```

Bisa Anda lihat, pada bagian `$matches[1]` selalu berisi match group pertama (dalam contoh yang kita bahas, `x` pada `(x,y)`). sedangkan pada `$matches[1][0]` berisi match group pertama untuk pola yang pertama kali cocok, demikian pula `$matches[1][1]` untuk pola kedua, `$matches[1][2]` untuk pola ketiga, dan seterusnya.

Untuk mengetahui berapa pola (koordinat) yang ditemukan, dapat digunakan:

```
count($matches[0])
```

atau

```
count($matches[1])
```

dan seterusnya.

Kalau Anda kurang suka dengan cara penyusunan seperti ini, Anda dapat menggunakan flag PREG_SET_ORDER di argumen keempat:

```
preg_match_all("/\((\d+),(\d+)\)/", $string, $matches,
PREG_SET_ORDER);
```

sehingga hasilnya:

```
Array
(
    [0] => Array
        (
            [0] => (1,2)
            [1] => 1
            [2] => 2
        )

    [1] => Array
        (
            [0] => (3,4)
            [1] => 3
            [2] => 4
        )

    [2] => Array
        (
            [0] => (7,1)
            [1] => 7
            [2] => 1
        )
```





```
    [3] => Array
        (
            [0] => (11,4)
            [1] => 11
            [2] => 4
        )

    [4] => Array
        (
            [0] => (5,7)
            [1] => 5
            [2] => 7
        )

    [5] => Array
        (
            [0] => (10,0)
            [1] => 10
            [2] => 0
        )

)
```

dengan kata lain, $matches[i][1] berisi match group pertama, $matches[i][2] berisi match group kedua, dan seterusnya. Untuk mengetahui ada berapa pola yang ditemukan:

```
count($matches);
```

## Mengetahui posisi pencocokan

Sebelum versi 4.3.0, PHP tidak menyediakan offset seperti halnya @- dan @+ pada Perl. Namun sejak PHP 4.3.0, Anda dapat menyertakan argumen ketiga bernilai PREG_OFFSET_CAPTURE untuk mengembalikan juga offset di argumen kedua $matches. Tapi perlu diperhatikan bahwa flag ini membuat tiap elemen $matches menjadi array 2 elemen.

Elemen yang pertama berisi isi stringnya, elemen kedua berisi offsetnya. Jadi $matches[0][0] menangkap string yang cocok dengan seluruh pola (sama dengan $& pada Perl), $matches[0][1] berisi offsetnya (sama dengan $-[0] pada Perl), $matches[1][0] berisi match group pertama (sama dengan $1 pada Perl), $matches[1][1] offsetnya (sama dengan $-[1] pada Perl), dan seterusnya.

Catatan: Anda juga bisa menggunakan flag ini di `preg_match_all()` bersama flag lain. Gabungkan antarflag dengan operator | (bitwise OR). Contoh:

```
<pre>
<?
$string = "(1,2), (3,4), (7,1), (11,4), (5,7), (10,0)";
preg_match_all("/\((\d+),(\d+)\)/", $string, $matches,
               PREG_SET_ORDER | PREG_OFFSET_CAPTURE);
print_r($matches);
?>
</pre>
```

Hasilnya:

```
Array
(
    [0] => Array
        (
            [0] => Array
                (
                    [0] => (1,2)
                    [1] => 0
                )

            [1] => Array
                (
                    [0] => 1
                    [1] => 1
                )

            [2] => Array
                (
                    [0] => 2
                    [1] => 3
                )

        )

    [1] => Array
        (
            [0] => Array
                (
                    [0] => (3,4)
                    [1] => 7
                )

            [1] => Array
                (
                    [0] => 3
                    [1] => 8
                )

            [2] => Array
                (
                    [0] => 4
                    [1] => 10
                )

        )

    [2] => Array
        (
            [0] => Array
                (
                    [0] => (7,1)
                    [1] => 14
                )

            [1] => Array
                (
                    [0] => 7
                    [1] => 15
                )

            [2] => Array
                (
                    [0] => 1
                    [1] => 17
                )

        )
```

```
    [3] => Array
        (
            [0] => Array
                (
                    [0] => (11,4)
                    [1] => 21
                )

            [1] => Array
                (
                    [0] => 11
                    [1] => 22
                )

            [2] => Array
                (
                    [0] => 4
                    [1] => 25
                )

        )

    [4] => Array
        (
            [0] => Array
                (
                    [0] => (5,7)
                    [1] => 29
                )

            [1] => Array
                (
                    [0] => 5
                    [1] => 30
                )

            [2] => Array
                (
                    [0] => 7
                    [1] => 32
                )
```





```
        )

    [5] => Array
        (
            [0] => Array
                (
                    [0] => (10,0)
                    [1] => 36
                )

            [1] => Array
                (
                    [0] => 10
                    [1] => 37
                )

            [2] => Array
                (
                    [0] => 0
                    [1] => 40
                )

        )

)
```

## Substitusi Regex

Untuk melakukan substitusi regex, gunakan fungsi `preg_replace()` dan `preg_replace_callback()`. Contoh kode berikut adalah implementasi `nl2br()`:

```
 1|<?
  |
 3|//
 4|// my-nl2br.php
 5|//
  |
 7|function my_nl2br($string) {
 8|  return preg_replace('/\n/', "<br/>\n", $string);
 9|}
  |
```

```
11|// test
12|echo my_nl2br("Halo, ini baris 1.\nIni baris 2.\nBaris
   3.\n");
  |
14|?>
```

Hasilnya (setelah Anda *View Source* di browser):

```
Halo, ini baris 1.<br/>
Ini baris 2.<br/>
Baris 3.<br/>
```

Argumen pertama fungsi `preg_replace()` adalah pola regex. Argumen kedua string pengganti. Argumen ketiga string yang ingin dicocokkan. `preg_replace()` sama dengan `s///g di Perl`, yang akan mengganti semua substring yang cocok dengan pola regex dengan string pengganti. Jika Anda hanya ingin mengganti pola yang pertama cocok saja, berikan argumen keempat (*limit*) bernilai 1. Anda juga dapat mengisi *limit* dengan 2, 3, dan seterusnya. untuk mengganti *N* buah pola yang pertama cocok.

Berbeda dengan `s///` pada Perl, `preg_replace()` tidak mengubah langsung isi string (in-place) melainkan mengembalikan string baru yang telah disubstitusi. Jika ingin mengganti $STRING, Anda tentu saja bisa melakukan:

```
$STRING = preg_replace($pola, $pengganti, $STRING);
```

Sama dengan Perl, backtracking disebutkan dengan \1, \2, dan seterusnya. Tapi jika Anda menggunakan string berkutip ganda, Anda perlu menuliskannya sebagai \\1. Sementara jika menggunakan string berkutip tunggal Anda bisa tetap menuliskannya sebagai \1. Di string pengganti, Anda juga bisa menyebutkan backtracking ini tapi dengan tetap \1, \2, dan lain sebagainya dan bukannya $1, $2, dan lain sebagainya seperti pada Perl. Contoh:

```
 1|<?
  |
 3|//
 4|// kata-ulang.php
 5|//
  |
 7|if (!isset($q)) $q = "";
  |
```





```
 9|echo "
10|  <form>
11|    Masukkan sebuah kalimat berisi kata ulang:<br>
12|    <input name=q size=60 value=\"",htmlentities($q),"\"><p>
13|    <input type=submit>
14|  </form>";
  |
16|if ($q)
17|  echo preg_replace("#\b(\w+)-\\1\b#", "\\1<sup>2</sup>",
   $q);
  |
19|?>
```

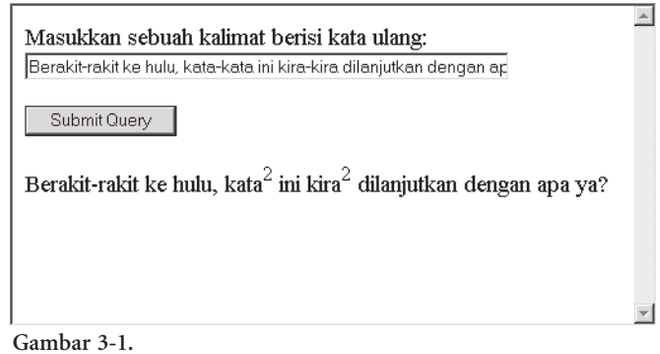
Seperti bisa dilihat pada contoh hasil keluaran di Gambar 3-1, program ini akan mengganti kata ulang seperti "kupu-kupu" menjadi "kupu²". Karena kita menambahkan pembatas kata \b pada pola regex, maka kata ulang berimbuhan seperti "Berakit-rakit" atau "selama-lamanya" tidak akan cocok dengan regex dan tidak tersubsitusi.

Jika ditulis dengan string berkutip tunggal, baris 17 menjadi:

```
echo preg_replace('#\b(\w+)-\1\b#', '\1<sup>2</sup>', $q);
```

Bagaimana jika kita ingin melakukan penggantian yang bersifat kompleks, bukan hanya dengan sebuah string statik? Ada dua cara. Pertama, kita dapat memberi modifier `e` pada pola regex. Modifier ini sama artinya seperti pada Perl, yaitu string pengganti dianggap adalah sebuah ekspresi PHP, bukan string statik. Contoh:

```
echo preg_replace("/(\d+)-(\d+)/e", "\\1-\\2", "238-2304");
```




Gambar 3-1.

hasilnya:

```
-2066
```

sementara tanpa modifier e hasilnya menjadi:

```
238-2304
```

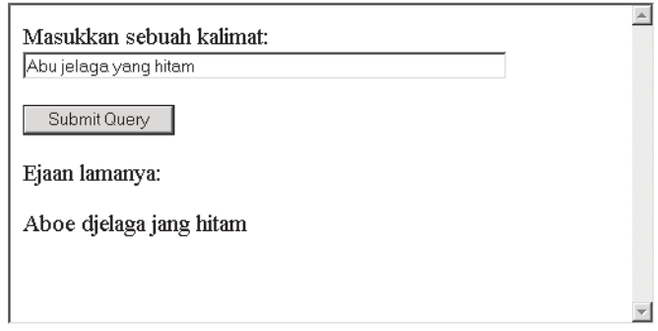
Cara kedua, untuk ekspresi atau penggantian yang lebih kompleks dengan menggunakan `preg_replace_callback()`. Fungsi ini sama dengan `preg_replace()`, tapi argumen kedua bukanlah string pengganti melainkan nama fungsi yang akan dipanggil setiap kali ditemukan substring yang cocok dengan pola. Contoh:

```
 1|<?
  |
 3|//
 4|// ejaan-lama.php
 5|//
  |
 7|function my_func1($m) {
 8|  if     ($m[1] == 'j') return 'dj';
 9|  elseif ($m[1] == 'u') return 'oe';
10|  elseif ($m[1] == 'y') return 'j';
11|}
  |
13|if (!isset($q)) $q = "";
  |
15|echo "
16|  <form>
17|    Masukkan sebuah kalimat:<br>
18|    <input name=q size=60 value=\"",htmlentities($q),"\"><p>
19|    <input type=submit>
20|  </form>";
  |
22|if ($q)
23|  echo "<p>Ejaan lamanya:";
24|  echo "<p>", preg_replace_callback('/([juy])/', "my_func1",
   $q);
  |
26|?>
```





Contoh keluaran program seperti di Gambar 3-2.
Perhatikan baris 24. Setiap kali ditemukan pola (yaitu huruf j, u, atau y), maka `preg_replace_callback()` akan memanggil fungsi buatan kita yaitu `my_func1()`. Fungsi `my_func1()` akan menerima argumen `$m` sama seperti `$matches` yang dikembalikan `preg_match()`. Fungsi ini lalu bebas mengembalikan nilai apa saja, yang akan dipakai sebagai string pengganti. Jadi, pada contoh string "Abu jelaga yang hitam", `my_func1()` dipanggil sebanyak 3 kali.




Gambar 3-2.

## Split regex

Untuk memecah string dengan pembatas regex, digunakan `preg_split()`. Fungsi ini sama dengan fungsi `split()` di Perl. Contoh ini (dicontek dari manual PHP):

```
$keywords = preg_split ("/[\s,]+/", "hypertext language, program
ming");
```

Baris di atas akan memecah string setiap kali ada spasi atau tanda koma.

## Mengescape string

Fungsi `preg_quote()` di PHP sama fungsinya dengan `quotemeta()` di Perl:

```
echo preg_quote("...");
```

Hasilnya:

```
\.\.\.
```

Mulai PHP 4.3.0 (yang menggunakan PCRE 4.x), PHP juga mendukung karakter meta `\Q` dan `\E` sebagai alternatif `preg_quote()`:

```
if (preg_match("/\Q...\E/", $string))
  echo "String mengandung tiga buah titik";
```

### Named capture dan fitur lain

Seperti dijelaskan di Bab 2, PCRE mendukung beberapa fitur yang tidak ada di regex Perl yaitu named capture dan beberapa modifier seperti `A`, `D`, `U`.

## Bahasa-Bahasa Lain

Mulai dari subbab ini kita akan menyinggung bahasa-bahasa seperti Python, Ruby, dan lain sebagainya. dengan lebih sepintas lalu saja, karena sebetulnya fokus utama buku ini adalah regex menggunakan Perl dan PHP. Sintaks fungsi/kelas regex pada bahasa-bahasa lain ini disinggung semata-mata demi kelengkapan dan yang terutama dibahas adalah perbedaannya dengan sintaks regex Perl atau fitur-fitur khusus yang dimilikinya. Tentu saja, mayoritas bahasa-bahasa ini menggunakan varian flavor regex Perl sehingga pada dasarnya pembahasan Bab 2 dan bagian awal Bab 3 berlaku juga untuk bahasa-bahasa ini.

### Python

Tidak seperti Perl, budaya di Python tidak terlalu "tergila-gila" dengan regex. Tapi seiring mottonya "batteries included", sejak versi 1.x Python sudah mendukung regex lewat modul `regex`, yang tidak seampuh regex Perl. Lalu muncul modul `re` yang menggunakan library PCRE yang setara kemampuannya dengan regex Perl. Seiring perkembangan, di Python 2.x Fredrik Lundh juga mengembangkan modul `sre` dengan kelebihan utama dukungan terhadap Unicode (karena pada saat itu PCRE 3.x belum mendukung Unicode). Modul `re` kemudian direname menjadi `pre` dan nama modul `re` sendiri menjadi interface standar dan "gateway" di mana kita dapat memilih ingin menggunakan implementasi `pre` atau `sre`. Defaultnya kini (Python 2.2 ke atas) adalah `sre`. `sre` mendukung mayoritas sintaks regex Perl dengan beberapa perbedaan sebagai berikut.:

1. `sre` mendukung named capture dengan sintaks `(?P<nama>...)`. Untuk menyebutkan capture atau match group (analogi dengan `\1`, `\2`, dan lain sebagainya) digunakan sintaks `(?P=nama)`.
2. Hanya mendukung positive dan negative look-ahead, tidak mendukung look-behind.
3. Tidak mendukung pola kondisional: `(?(condition)...)` atau `(?(condition)...|...)`.
4. Memiliki modifier `L` dan `U`. Modifier `L` berarti karakter meta seperti `\w`, `\W`, `\b`, dan `\B` menggunakan locale setempat untuk menentukan mana karakter yang termasuk dalam kata atau bukan. `U` berarti menggunakan Unicode.
5. Tidak mengenal modifier `e` (Eval).





### Bekerja dengan regex di Python

Untuk bekerja dengan regex, mula-mula kita mengimpor modul `re`. Kita lalu menggunakan metode `compile()` untuk mengubah sebuah string menjadi objek regexp. Objek regex inilah yang nanti bisa digunakan untuk melakukan pencocokan dan substitusi. Objek regex memiliki metode seperti `search()`, `sub()`, dan `split()`. Beberapa contoh:

```
re.compile(r'^abc$').search("abc")          # berhasil
re.compile(r'^abc$').search("abcd")         # gagal, hasilnya None
re.compile(r'^abc$', re.I).search("ABC")    # berhasil
```

Metode match menerima pola di argumen pertama dan modifier di argumen kedua. Modifier yang tersedia antara lain `re.I`, `re.M`, `re.X`, `re.S` (jika menyebutkan lebih dari satu modifier sekaligus, lakukan penjumlahan atau bitwise OR, misalnya: `re.I|re.M`). Biasakan untuk menggunakan raw string (string yang dikutip dengan `r''`) sebagai pengutip string pola untuk menghindari kesalahpahaman karakter meta dengan interpretasi karakter `\` yang dilakukan oleh Python terhadap kutip tunggal maupun kutip ganda. Jika pencocokan berhasil, yang dikembalikan oleh metode `search()` adalah objek matchgroup. Contoh:

```
$ python2
Python 2.2.2 (#1, Jan 30 2003, 21:26:22)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type "help", "copyright", "credits" or "license" for more infor
mation.
>>> match = re.compile(r'(..):(..):(..)').search("Waktu saat ini:
11:03:57")
>>> match.group()
11:03:57
>>> match.group(0)
11:03:57
>>> match.group(1)
11
```

Untuk mencari semua pola yang ada, Anda bisa menggunakan `findall()`:

```
>>> re.compile(r'(\w+)').findall("Waktu saat ini: 11:03:57")
['Waktu', 'saat', 'ini', '11', '03', '57']
```

tapi ini hanya akan mengembalikan string yang cocok, bukan objek match. Agar Anda bisa memperoleh objek match untuk setiap kecocokan, gunakan `finditer()`.

Hasilnya adalah iterator yang dapat Anda gunakan pada loop `for`.

```
>>> for i in re.compile(r'(\w+)').finditer("Waktu saat ini:
11:03:57"):
...   print i.group(1)
...
Waktu
saat
ini
11
03
57
```

Untuk melakukan substitusi, gunakan metode `sub()`:

```
>>> re.compile(r'[aeiou]').sub("*", "Jangan pipis sembarangan")
'J*ng*n p*p*s s*mb*r*ng*n'
```

Argumen pertama `sub()` adalah string pengganti, argumen kedua adalah subjek (string yang ingin diganti), dan argumen ketiga bersifat opsional yaitu jumlah maksimum penggantian:

```
>>> re.compile(r'[aeiou]').sub("*", "Jangan pipis sembarangan", 3)
'J*ng*n p*pis sembarangan'
```

`sub()` tidak mengubah string langsung melainkan mengembalikan string baru. Ini karena string di Python bersifat immutable (tidak dapat diganti).

Untuk melakukan penggantian regex yang kompleks memang agak tidak nyaman di Python, karena saat ini tidak ada metode seperti `subfunc()` yang menerima fungsi. Tapi Anda bisa menggunakan trik misalnya membelah string menjadi array dulu lalu melakukan pengubahan yang lebih sederhana pada tiap elemen string. Untuk melakukan split dengan regex, gunakan metode `split()`:

```
>>> re.compile(r',\s+').split("1, 2, 3, 4, 5")
['1', '2', '3', '4', '5']
```

Untuk mengescape string, gunakan fungsi `escape()`:

```
>>> re.escape("...")
'\\.\\.\\.'
```





Untuk penjelasan lebih mendetil mengenai regex di Python, silakan merujuk ke dokumentasi Python (disertakan di CD).

## Ruby

Ruby adalah bahasa skripting OO produk Asia, diciptakan oleh Yukihiro Matsumoto yang akrab dipanggil "Matz". Di negara asalnya, Ruby amat popular melebihi popularitas Perl maupun Python. Dalam banyak hal, Ruby mengikuti Perl. Termasuk di antaranya mayoritas sintaks regex Perl dan beberapa operator regex.

Mesin regex Ruby hingga versi 1.8 (tahun 2003) adalah mesin regex POSIX milik GNU (bagian dari glibc) yang dimodifikasi oleh Matz dkk agar mendukung sintaks regex Perl, Unicode/UTF8, dan multibyte (yang digunakan untuk memroses teks Jepang). Masalahnya, library ini berlisensi LGPL yang lebih restriktif ketimbang lisensi Ruby itu sendiri (lisensi Ruby adalah *dual license*; pengguna dapat memilih GPL atau lisensi Ruby yang mirip-mirip Artistic License ala Perl). Restriksi ini mempersulit sebagian pengguna Ruby yang ingin menggunakan Ruby untuk membuat aplikasi komersial. Karena itu dikembangkanlah mesin regex baru, yang *made in Japan* juga, bernama Oniguruma yang akan dipakai mulai Ruby 1.9 ke atas. (Saat buku ini ditulis, development Ruby 1.9 baru akan dimulai.) Mesin regex Oniguruma selain lisensinya sama dengan Ruby juga akan mengatasi beberapa keterbatasan mesin regex lama (dijelaskan di bawah) dan memperkenalkan beberapa fitur baru.

Dibandingkan regex Perl, mesin regex Ruby 1.8 memiliki perbedaan sebagai berikut :

1. Hanya mendukung positive dan negative look-ahead. Tidak mendukung look-behind.
2. Modifier `m` di Ruby sama fungsinya dengan modifier `s` di Perl/PHP, yaitu agar `.` (titik) juga cocok dengan newline. Sementara modifier `s` di Ruby artinya berbeda sama sekali, yaitu untuk mengaktifkan fasilitas multibyte. `s` berarti menganggap string berupa set karakter SJIS. `s` juga boleh ditulis `S`.
3. Tidak mengenal modifier `e` (Eval), melainkan `e` dipakai untuk menganggap string ada dalam set karakter EUC. `e` boleh ditulis `E`.
4. Memiliki modifier `u`/`U`. Dengan modifier `u` atau `U` ini, string dianggap ada dalam set karakter Unicode UTF8.

Pada dasarnya mesin regex di Ruby saat ini tidaklah secepat atau seteroptimasi mesin regex Perl. Pola-pola kompleks dapat saja berjalan amat lambat di Ruby, sementara cepat di Perl. Mudah-mudahan kondisi ini diperbaiki oleh Ruby di versi mendatang dengan Oniguruma-nya. Oniguruma juga menjanjikan semua fitur-fitur regex yang kurang saat ini, yaitu look-behind, named capture, atomic group, dan possessive quantifier.

### Bekerja dengan regex di Ruby

Sama seperti Perl, Ruby pun memiliki operator // untuk mengapit regex dan operator =~. Perbedaannya, untuk menggunakan karakter pengapit alternatif, digunakan sintaks %r(). Contoh:

```
"Satu dua tiga" =~ %r{(ti..)}i
```

Anda juga dapat menggunakan %r#...#, %r{...}, %r!...!, dan lain sebagainya. sama seperti m// di Perl.

Operator =~ akan mengembalikan nil jika tidak cocok atau bilangan mulai dari 0 yang merupakan offset di mana terjadi kecocokan (sama dengan $-[0] di Perl). Contohnya, ekspresi di atas akan menghasilkan 9.

Ruby juga mengenal operator !~ sama seperti di Perl, di mana operator ini akan menghasilkan true jika pola tidak cocok atau false jika cocok. Karena sifatnya yang lebih murni OO, sebetulnya // dan %r() akan membuat sebuah objek regex (instans kelas Regexp):

```
$ ruby -v
ruby 1.8.0 (2003-08-04) [i686-linux-gnu]
$ irb
irb(main):001:0> /123/
=> /123/
irb(main):002:0> /123/.class
=> Regexp
irb(main):003:0> %r(^abc)
=> /^abc/
irb(main):004:0> %r(^abc).class
=> Regexp
```

Selain menggunakan operator =~, Anda juga dapat menggunakan metode match() pada objek Regexp, mirip seperti di Python:

```
irb(main):001:0> re = /(\d+)/
=> /(\d+)/
irb(main):002:0> re.match("Satu domba, dua domba, 3 domba.")
=> #<MatchData:0x401efdac>
```

Hasil pencocokan adalah objek MatchData. Anda dapat mengambil match group dengan menggunakan metode []:





```
irb(main):003:0> m = re.match("Satu domba, dua domba, 3 domba.")
=> #<MatchData:0x401eae10>
irb(main):004:0> m[1]
=> "3"
```

Di Ruby banyak digunakan blok (subrutin anonim). Anda dapat menggunakan metode scan() pada string:

```
"kata-kata indah dari buku-buku sastra".scan(/(\w+)-\1/) { |m|
puts "kata ulang: #{m}2x" }
```

Hasilnya:

```
kata ulang: kata2x
kata ulang: buku2x
```

Untuk mengganti string, digunakan metode gsub() pada string:

```
irb(main):001:0> string = "kata-kata indah dari buku-buku sastra"
=> "kata-kata indah dari buku-buku sastra"
irb(main):002:0> string.gsub(/a\b/, "e")
=> "kate-kate indah dari buku-buku sastre"
irb(main):003:0> string
=> "kata-kata indah dari buku-buku sastra"
```

Jika hanya ingin mengganti pola pertama yang ditemukan, gunakan sub(). Jika ingin mengganti string secara langsung (in-place), gunakan gsub!() dan sub!():

```
irb(main):004:0> string.sub!(/a\b/, "e")
=> "kate-kata indah dari buku-buku sastra"
irb(main):004:0> string
=> "kate-kata indah dari buku-buku sastra"
```

Untuk melakukan pergantian yang kompleks, juga digunakan blok. gsub() dapat menerima blok:

```
irb(main):001:0> benar = "5"
=> "5"
irb(main):002:0> "6, 4, 5, 1".gsub(/\d+/) { |x|
  x==benar ? "benar" : "SALAH!" }
=> "SALAH!, SALAH!, benar, SALAH!"
```

Untuk mensplit string dengan regex, gunakan metode `split()` pada string.

```
irb(main):001:0> "1|2|3".split /\|/
=> ["1", "2", "3"]
```

Untuk mengescape string, gunakan metode `escape()` pada kelas `Regexp`:

```
irb(main):001:0> Regexp.escape("...")
=> "\\.\\.\\."
```

## Tcl

Tcl (dibaca "tickle", singkatan dari Tool Command Language) adalah sebuah bahasa pemrograman yang sintaksnya sederhana dan mudah dipelajari—lebih mudah dari pada Perl, Python, atau Ruby. Beberapa hal yang membuat Tcl menarik adalah dukungan Unicode yang solid, kemudahan di-embed, thread safety, dan toolkit GUI builtinnya yang bernama Tk. Tcl banyak diembed di sofware server, seperti di **OpenACS**, **AOLServer,** atau **PostgreSQL**. Atau bahkan di program-program robot IRC. Tk sendiri banyak dipakai oleh bahasa lain seperti Perl (TkPerl) dan Python (Tkinter).

Versi pertengahan 1990-an, Tcl bisa dibilang merupakan bahasa skripting yang kurang kompetitif dan skalabel untuk pemrograman serius: lambat, hanya mendukung tipe data string, bersifat interpreter (tidak terkompilasi menjadi bytecode), tidak mendukung paket/namespace, dan lain sebagainya. Namun sejak versi 8.x (tahun 1997), Tcl telah berubah menjadi sebuah bahasa yang patut diperhitungkan dengan memperbaiki hampir semua kekurangan seriusnya. Meskipun Tcl bukanlah bahasa termodern atau murni OO (demikian juga dengan toolkit GUI-nya), kesederhanaan dan implementasinya yang solid tetap membuat Tcl/Tk menarik untuk banyak aplikasi.

Salah satu kekuatan Tcl adalah kemampuan pemrosesan string. Karena itu Tcl pun memiliki fitur regex. Bahkan tool **Expect** berbasis Tcl yang banyak dipakai/dikloning oleh bahasa lain seperti Perl, menggunakan regex secara intensif.

Regex di Tcl mendukung flavor POSIX dan juga sebagian besar regex Perl. Beberapa perbedaan dibandingkan regex Perl:

1. `\b` bukanlah karakter meta untuk posisi batas antarkata, melainkan melambangkan backspace.
2. `\B` bukanlah karakter meta untuk posisi di dalam kata, melainkan untuk melambangkan `\` literal (dengan kata lain, sinonim `\\`).
3. Sebagai pengganti `\b` dan `\B` di Perl, Tcl memiliki `\m` (posisi awal kata), `\M` (posisi akhir kata), `\y` (di awal atau akhir kata, ini sama dengan `\b` di Perl), dan `\Y` (bukan di awal ataupun akhir kata, ini sama dengan `\B` di Perl).
4. Tidak mengenal modifier `e` (Eval). Modifier lain rata-rata ada seperti `i` (`-nocase`), `x` (`-expanded`), `m` (`-lineanchor`), dan lain sebagainya.
5. Tidak mendukung look-behind, hanya look-ahead dengan sintaks yang sama di Perl.
6. Tidak mendukung pola kondisional, atomic group, named capture.

### Bekerja dengan regex di Tcl

Perintah di Tcl yang berhubungan dengan regex adalah `regexp` dan `regsub`. Untuk mencocokkan sebuah string dengan pola regex, digunakan perintah `regexp` yang memiliki sintaks:

```
regexp ?switches? pat str ?matchVar? ?subMatchVar1 subMatchVar2
...?
```

`switches` adalah modifier regex, bersifat opsional dan dimulai dengan tanda minus. Terdapat beberapa switch antara lain `-nocase` (sama dengan `i` di Perl/PCRE), -expanded (sama dengan `x` dengan Perl/PCRE), `-lineanchor` (sama dengan modifier `m` di Perl/PCRE), `-all` (sama dengan modifier `g` di Perl). `pat` adalah pola regex. Kalau pola regex Anda dimulai dengan `-`, maka terlebih dahulu Anda harus memberi – sebagai tanda akhir `switches`. `str` adalah string yang ingin dicocokkan. `matchVar` dan `subMatchVar` menangkap match group (`matchVar` sama dengan `$&` di Perl, subMatchVar1 sama dengan `$1` di Perl, `subMatchVar2` dengan $2, dan seterusnya.) Perintah ini akan mengembalikan *N* jika berhasil/cocok (di mana *N* adalah bilangan bulat positif yang menyatakan jumlah pola yang cocok) atau 0 jika tidak ada yang cocok. Contoh:

```
$ tclsh
% regexp abc "abcde"
1
% regexp abc "ABC"
0
% regexp -nocase a\wc "ABC"
1
```

Untuk mengambil match group, sebutkan argumen ketiga atau keempat:

```
% set kalimat "Di sini senang di sana senang di mana-mana hatiku
senang"
Di sini senang di sana senang di mana-mana hatiku senang
```

```
% regexp {s([aeoiu])n\1} $kalimat kata huruf
1
% puts $kata
sini
% puts $huruf
i
```

Untuk mengambil semua pola, gunakan switch `-inline` sehingga perintah `regexp` akan mengembalikan list berisi match group dan `-all` yang akan menggabungkan semua match group dari semua pola yang ditemukan menjadi satu rangkaian:

```
% regexp -inline -all {s([aeoiu])n\1} $kalimat
sini i sana a
```

Perhatikan bahwa kita tidak menyebutkan lagi argumen ketiga dan keempat (match/submatch) karena match group langsung dikembalikan sebagai list oleh `regexp`. Untuk substitusi string, digunakan perintah `regsub`. Sintaks perintah di bawah ini:

```
regsub ?switches? pat str replace_pat var
```

`switches` adalah daftar modifier. `pat` adalah pola regex. `str` adalah subjek yang ingin diganti. `replace_pat` adalah pola pengganti (yang dapat mengandung `&` untuk substitusi seluruh string yang cocok dengan pola, `\1` untuk match group pertama, `\2` match group kedua, dan seterusnya). `var` adalah nama variabel untuk menampung hasil penggantian. Contoh:

```
% regsub aku "aku cinta kamu" kamu hasil; puts $hasil
kamu cinta kamu
% regsub -all {[aeiou]} "aku cinta kamu" * hasil; puts $hasil
*k* c*nt* k*m*
% regsub {(\d\d)-(\d\d)-(\d\d\d\d)} 31-12-2003 \\3-\\2-\\1 hasil
1
% puts $hasil
2003-12-31
```

## Javascript

Regex telah didukung sejak Javascript 1.2, yaitu versi Javascript yang ada di Netscape 4.x dan IE4. Javascript 1.5 (Netscape 6 ke atas dan Mozilla) mendukung sebagian besar sintaks Perl, bahkan memiliki operator `//` untuk mengapit pola regex seperti pada Perl dan Ruby.





Tapi regex Javascript memiliki beberapa perbedaan dibandingkan dengan yang lain. Di antaranya:

1. Tidak bisa mendukung modifier `s` (single line), `x` (expanded mode), maupun `e` (eval). Modifier yang dikenal hanya `i`, `g`, dan `m`. Semuanya juga memiliki arti yang sama dengan di Perl.
2. Tidak support atau mendukung look-behind, terbatas hanya mendukung look-ahead.
3. Tidak mendukung named capture, atomic group, atau possessive quantifier.
4. Tidak mendukung pola kondisional maupun komentar di dalam regex.

### Bekerja dengan regex di Javascript

Untuk menyatakan regex, kita menggunakan `//` atau membuat objek `RegExp`:

```
re = /^abc$/i
re = new RegExp("^abc$", "i"
```

Setelah objek `RegExp` dibuat, Anda dapat melihat modifier apa saja yang telah disebutkan melalui beberapa properti objek:

```
re.ignoreCase  // true jika regex dibuat dengan modifier "i"
re.global      // true jika regex dibuat dengan modifier "g"
re.multiline   // true jika regex dibuat dengan modifier "m"
```

Untuk mencocokkan string, digunakan metode `test()` atau `exec()` pada objek `RegExp`. `test()`. Hasil yang muncul sebagai jawaban hanya berupa `true` atau `false`:

```
str = "sun shines brighter if you'll see it with your heart.";
if (/\w+'\w+/.test(str)) alert("Kalimat mengandung kata
berapostrof");
```

Sementara dengan `exec()` Anda dapat mengambil match group:

```
<script>
str = "sun shines brighter if you'll see it with your heart.";
re = /\b[Ss]\w+\b/g;
while (match = re.exec(str)) {
  document.writeln("Ditemukan kata berawalan 's':
"+match[0]+"<br>\n");
}
</script>
```

Hasilnya di browser adalah:

```
Ditemukan kata berawalan 's': sun
Ditemukan kata berawalan 's': shines
Ditemukan kata berawalan 's': see
```

`exec()` akan mengembalikan array, elemen ke-0 sama seperti $& pada Perl, elemen ke-1 seperti $1 pada Perl, dan seterusnya. Perhatikan bahwa seperti di Perl, modifier `g` diperlukan untuk mencari semua pola yang cocok. Dengan `g`, setiap selesai mencocokkan satu kali, Javascript akan mengupdate properti `re.lastIndex` (posisi awal string yang ingin dicocokkan) agar saat berikutnya tidak perlu mengulang dari awal string. Jika tidak ada `g`, `re.lastIndex` tidak diupdate dan Anda akan terjebak dalam loop karena regex terus-menerus menemukan kata pertama `'sun'`.

Untuk mengganti string melalui regex, gunakan metode replace pada objek `String`. `replace()` menerima dua argumen, pertama adalah pola regex dan kedua adalah string pengganti. String pengganti dapat mengandung $1, $2, dan lain sebagainya. Metode ini mengembalikan hasil berupa string yang telah diganti. untuk mengambil match group. Contoh:

```
<script>
valid = false;
while (!valid) {
  name = prompt("Masukkan nama lengkap Anda");
  dir = name.replace(/[^A-Za-z]+/g, "_").toLowerCase();
  if (dir.length > 0) valid = true;
}
alert("Nama database untuk Anda: "+dir+".db");
</script>
```

Jika dimasukkan sebuah nama, misalnya: `"Prof. Dr. Steven Haryanto"`, maka kode di atas akan mengubahnya menjadi `"prof_dr_steven_haryanto.db"`.

Regex dalam Javascript amat berguna terutama dalam validasi form. Contoh:

```
 1|<!– login.html –>
  |
 3|<script>
 4|function checkUserName(v) {
 5|    re = /^[A-Za-z]{8}\d{4}$/;
 6|    return re.test(v);
 7|}
 8|function checkPin(v) {
 9|    re = /^\d{6}$/;
10|    return re.test(v);
11|}
12|function checkForm(f) {
13|    if (!checkUserName(f.username.value)) {
14|        alert("Please enter a valid username!");
15|        f.username.focus();
16|        return false;
17|    }
18|    if (!checkPin(f.pin.value)) {
19|        alert("Please enter a valid PIN!");
20|        f.pin.focus();
21|        return false;
22|    }
23|    return true;
24|}
25|</script>
  |
27|<h2>Internet Banking</h2>
28|<form method=GET onSubmit="return checkForm(this)">
29|   Masukkan username: <input name=username><br>
30|   Masukkan PIN: <input type=password name=pin><br>
31|  <input type=submit value=Login>
32|</form>
```

Listing file di atas merupakan contoh halaman login. Sebelum form dapat disubmit, kode Javascript akan mewajibkan dulu username berupa 8 huruf diikuti 4 angka dan PIN berupa 6 angka (seperti format username dan PIN KlikBCA).

## Java

Java agak terlambat dalam memasukkan fasilitas regex ke dalam library kelasnya. Baru di Java 1.4 tahun 2002 Sun memasukkan `java.util.regex`. Namun sebelum 2002 para programer Java telah memanfaatkan beberapa library yang tersedia, misalnya dari IBM (`com.ibm.regex`), Apache Jakarta Project (`org.apache.oro.text.regex` dan `org.apache.regexp`), dan GNU (`gnu.regexp`).

Regex di Java (`java.util.regex`) cukup solid dan memiliki hampir semua fitur regex Perl. Berikut beberapa perbedaannya:

1. Regex Java Mendukung possessive quantifier, tapi tidak mendukung atomic group. Perl sendiri mendukung atomic group tapi tidak memiliki sintaks shortcut possessive quantifier. Seperti dijelaskan di Bab 2, possessive quantifier adalah bentuk shortcut dari atomic group.
2. Tidak mendukung komentar dalam regex (`?#...`). Tapi mendukung modifier `x`.
3. Tidak mendukung pola kondisional.

### Bekerja dengan regex di Java

Untuk mulai menggunakan regex, pertama kita membuat objek `java.util.regex.Pattern` dari string menggunakan konstruktor `compile()`. Metode ini menerima dua argumen, yang pertama adalah string pola dan kedua adalah flag-flag modifier seperti `Pattern.CASE_INSENSITIVE` (sama dengan `i` di Perl), `Pattern.DOTALL` (sama dengan `s` di Perl), dan lain sebagainya. Untuk menggabungkan beberapa flag, gunakan operator `|`. Contoh:

```
import java.util.regex.*;
pat = Pattern.compile("^abc$", Pattern.CASE_INSENSITIVE);
```

Setelah membuat objek Pattern, kita membuat objek `Matcher` dengan memanggil metode `matcher()` pada objek Pattern kita. `matcher()` menerima sebuah argumen berupa string yang ingin dicocokkan.

```
m = pat.matcher("ABC");
```

Selanjutnya kita memanggil metode `find()` untuk mulai mencari pola. Jika berhasil, metode ini akan mengembalikan `true`. Kita lalu dapat mengambil match group dengan menggunakan metode `group()`. `group()` atau `group(0)` sama dengan `$&` di Perl, `group(1)` dengan `$1`, dan seterusnya. Kita juga dapat mengetahui posisi pencocokan dengan metode `start()` dan `end()`.

Untuk melakukan substitusi, digunakan metode `replaceFirst()` dan `replaceAll()` pada objek string. Sesuai namanya, `replaceFirst()` hanya akan mengganti pola pertama yang ditemukan sementara `replaceAll()` semua (sama dengan mode `g` di Perl). Metode ini menerima satu argumen yaitu string pengganti. Di dalam string pengganti boleh terdapat `$0`, `$1`, `$2`, dan lain sebagainya yang akan diganti dengan match group (`$0` sama dengan `$&` di Perl).

## .NET

Platform Windows sebetulnya secara tradisi tidak terlalu akrab dengan regex. Misalnya, baru di versi-versi terbaru MS ASP dan VB 6.0 saja terdapat fasilitas regex. Tapi di framework .NET terdapat kelas regex yang cukup ampuh,





mendukung hampir semua sintaks regex Perl dan memiliki fitur-fitur tambahan. Dengan demikian, para programer C#, VB.NET, dan ASP.NET dapat menikmati fasilitas regex yang sama. Berikut ini beberapa perbedaan dan fitur unik regex di .NET:

1. Mendukung named capture dengan sintaks (`?<NAMA>...`). Sintaks ini agak berbeda dengan di Python dan PCRE yaitu (`?P<NAMA>...`). Perl sendiri dapat mendukung named capture dengan sintaks .NET melalui modul `Regexp::Fields`.
2. Regex yang telah dikompile dapat disimpan ke disk untuk dipakai program lain.
3. Mendukung pencocokan dari kanan ke kiri. Fitur ini unik dan mungkin cocok digunakan untuk teks bahasa Arab misalnya. Tapi menurut penulis buku Jeffrey Friedl, mode ini saat ini masih memiliki kejanggalan atau bug dan sedang diperbaiki untuk .NET versi 2.0.

### Bekerja dengan regex di .NET

Ada beberapa bahasa yang datang bersama .NET, tapi contoh-contoh pada subbab ini menggunakan VB.

Untuk mulai menggunakan regex, pertama kita mengimpor namespace `System.Text.RegularExpressions`. Di dalam namespace ini terdapat beberapa kelas; yang pertama adalah kelas `Regex`. Kelas `Regex` dapat dipakai untuk mengkompilasi string atau langsung mencocokkan string dengan pola. Misalnya, untuk mengetes apakah sebuah pola cocok dengan string:

```
if Regex.isMatch("5 roti 2 ikan", "\b\d+\b")
  Console.WriteLine("Teks mengandung bilangan")
```

`isMatch()` menerima argumen ketiga berupa modifier. Beberapa modifier yang tersedia: `RegexOptions.IgnoreCase` (sama dengan `i` di Perl), `RegexOptions.IgnorePatternWhitespace` (sama dengan `x` di Perl), `RegexOptions.Multiline` (sama dengan `m` di Perl), dan `RegexOptions.Singleline` (sama dengan `s` di Perl).

Untuk menangkap match group, digunakan metode `Match()` pada kelas `Regex`. Jika terjadi kecocokan, `Match()` akan mengembalikan objek Match yang berisi metode `Groups()` yang merupakan match group. Ambil properti `Value` pada objek `Groups` untuk memperoleh nilai match group. Contoh:

```
Dim bil as String = _
 Regex.Match("5 roti 2 ikan", "\b(\d+)\b").Groups(1).Value
Console.WriteLine(bil)
```

Hasilnya adalah 5. Perhatikan bahwa jika kita membuat pola sebagai berikut:

```
\b(?<bil>\d+)\b
```

maka kita dapat mengambil group dengan menyebut nama capturenya sebagai berikut:

```
Groups("bil").value
```

Untuk mengganti string dengan regex, gunakan metode `Replace()` pada kelas `Regex`. `Replace()` menerima empat argumen: pertama string yang ingin diganti, kedua pola regex, ketiga string pengganti, dan keempat (opsional) modifier. String pengganti dapat mengandung `$&`, `$1`, `$2`, dan seterusnya. yang akan diganti oleh match group:

```
' Tukar-tukar kata pertama, kedua, ketiga
Dim str2 as String = _
  Regex.Replace("orang tidak kaya", _
                "(\w+) (\w+) (\w+)", _
                "$2 $3 $1")
```

Hasilnya adalah `str2` menjadi `"tidak kaya orang"`.