Bab 2

Sintaks Regex

Bab ini membahas sintaks regex dan merupakan tutorial bagi pemula. Para pembaca yang telah familiar dengan sintaks regex dapat melewati bab ini, kecuali jika ingin mereview ulang karakter-karakter meta dan modifier regex.

Karakter-Karakter Meta Dasar

Pemilihan (alternation)

Karakter meta | (garis vertikal) disebut karakter meta pemilihan. Sintaksnya adalah:

```
p1|p2
p1|p2|p3
p1|p2|p3|...
```

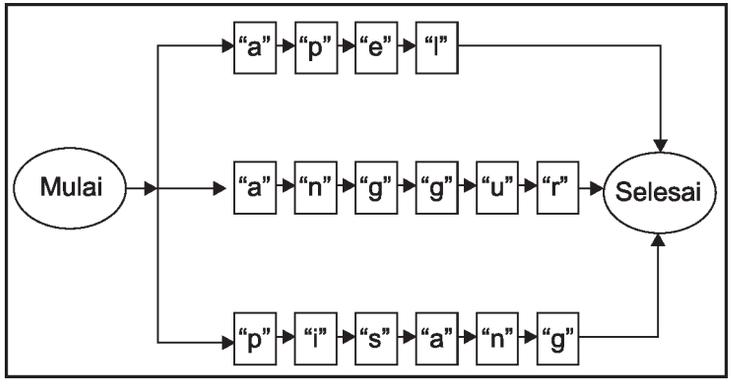
di mana p1, p2, atau p3 adalah subpola. Karakter meta ini dapat dibaca "atau" (salah satu dapat cocok). Beberapa contoh:

```
aku|kamu
```

Regex di atas dapat dibaca "deretan huruf aku *atau* deretan huruf kamu". Pola regex ini akan cocok dengan string aku, akukamu, kamuaku, paku, kamu, atau kamus. Tapi tidak akan cocok dengan a (tidak cocok dengan aku maupun kamu), ak, kam, amu, atau dia (tidak ada dalam pemilihan aku maupun kamu).

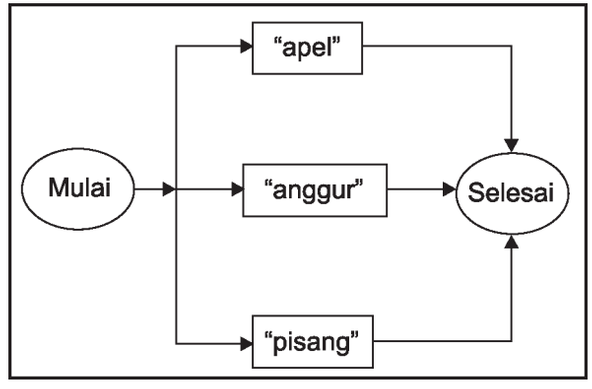
```
apel|anggur|pisang
```

Regex ini dapat dibaca "deretan huruf apel *atau* anggur *atau* pisang". Pola di atas akan cocok dengan string apel, pohon apel, anggur, pisang, pisang ambon, atau apelanggurpisang (mengandung subpola apel). Tapi tidak akan cocok dengan jambu (tidak ada dalam daftar pemilihan apel, anggur, maupun pisang).



Gambar 1-1.

Dalam diagram mesin state, pemilihan dapat digambarkan sebagai percabangan (Gambar 1-1). Dalam mencocokkan string misalnya pohon apel, mesin regex mula-mula akan mencoba karakter pertama p. Karakter ini cocok dengan cabang ketiga (p pada deretan kotak pisang). Mesin regex lalu mencoba karakter kedua pada string o. Karena o tidak cocok dengan kotak kedua i, maka mesin regex melakukan backtracking (mundur) kembali ke kotak pertama. Mesin regex lalu akan mencoba karakter kedua o di titik awal ini. Tidak ada yang cocok (dengan a, a, p). Demikian pula dengan karakter ketiga (h), keempat (o), kelima (n), keenam (" ", spasi). Karakter ketujuh a cocok dengan cabang pertama. Ternyata karakter-karakter berikutnya cocok dengan p, e, l. Maka mesin regex mencapai titik ujung Selesai mesin state. Ini berarti string cocok dengan pola regex. Inilah cara kerja dasar mesin regex. Untuk mempersingkat dan menyederhanakan diagram mesin state berikutnya, dalam diagram-diagram selanjutnya kita kebanyakan akan menggabungkan deretan string menjadi satu kotak saja, seperti di Gambar 1-2.



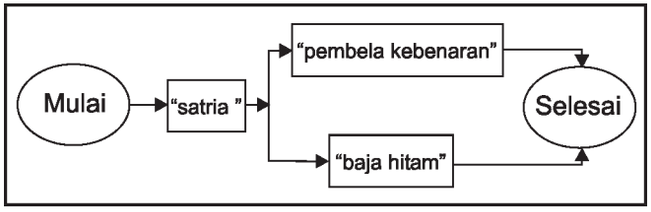
Gambar 1-2.

Pengelompokan (grouping)

Karakter meta (), tanda kurung biasa, merupakan karakter meta untuk pengelompokan. Pengelompokan mirip dengan fungsi tanda kurung di matematika. Umumnya karakter meta ini dipakai bersama dengan karakter meta lain. Contoh:

```
satria (baja hitam|pembela kebenaran)
```

Regex di atas dapat dibaca "deretan huruf satria , diikuti deretan huruf baja hitam atau pembela kebenaran". Dalam diagram mesin state bisa digambarkan seperti di Gambar 1-3.



Gambar 1-3.

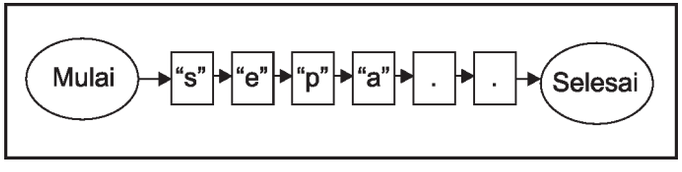
Pola cocok dengan string satria baja hitam, satria pembela kebenaran, satria baja hitam pembela kebenaran (cocok dengan satria diikuti baja hitam), tapi tidak cocok dengan satria tangguh pembela kebenaran (sebab tidak langsung diikuti baja hitam atau pembela kebenaran) atau baja hitam satria atau baja hitam pembela kebenaran.

Karakter apa saja (any character)

Karakter meta . (titik) merupakan karakter meta yang dapat cocok dengan satu karakter "apa saja", baik itu huruf, angka, spasi, dan lain sebagainya. Contoh:

```
sepa..
```

Pola regex di atas dapat dibaca "deretan huruf sepa diikuti oleh dua huruf apa saja." Jika digambarkan dalam bentuk diagram, bentuknya seperti di Gambar 1-4. Regex di atas akan cocok dengan string sepatu, sepak (sepak diikuti spasi), sepak! (sepak diikuti tanda seru), sepak!!! (sepak diikuti tiga tanda seru), sepak terjang, atau kesepakatan. Regex tidak akan cocok dengan string sepa (karena hanya terdiri dari 4 huruf), sepak (hanya terdiri dari 5 huruf), atau email sepam (karena sepa tidak diikuti dua karakter).



Gambar 1-4.

```
bat.k
```

Regex di atas cocok dengan batak, batok, batuk, batpk, bat!k, bat.k (titik itu sendiri), tapi tidak cocok batas (karena tidak ada k) atau batrak (karena ada dua huruf sebelum k).

Perhatikan tanda kutip pada "apa saja" untuk definisi karakter meta ini. Hal ini dikarenakan karakter newline, pada mode tertentu, bisa saja tidak dianggap cocok dengan karakter meta ini. Namun kita akan menyinggung mengenai ini nanti saat membahas mode operasi atau modifier regex.

Kelas karakter (character class, character set)

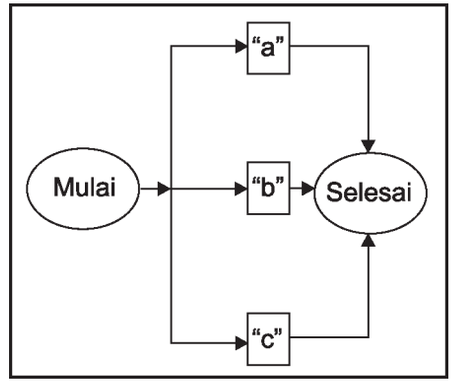
Karakter meta [dan] mengapit sebuah set karakter. Kelas karakter berguna untuk mendefinisikan sekumpulan karakter yang dapat cocok. Pada dasarnya kelas karakter mirip dengan karakter meta alternasi, hanya saja kelas karakter sesuai namanya hanya berlaku untuk karakter tunggal. Namun kelas karakter mengenal sintaks rentang (-, tanda minus) dan negasi (^, tanda pangkat). Berikut ini sintaks kelas karakter:

```
[abc...]
[a-z]
[^a-z]
```

di mana a, b, c, z adalah sebuah karakter. Beberapa contoh kelas karakter:

```
[abc]
```

Regex di atas dapat dibaca: "huruf a, atau huruf b, atau huruf c". Regex dapat cocok dengan string a, ab, ca, abc, tapi tidak cocok dengan d, e, f. Dalam diagram mesin state, kelas karakter juga dapat digambarkan sebagai percabangan (Gambar 1-5).



Gambar 1-5.

```
[0-9a-f][0-9a-f]
```

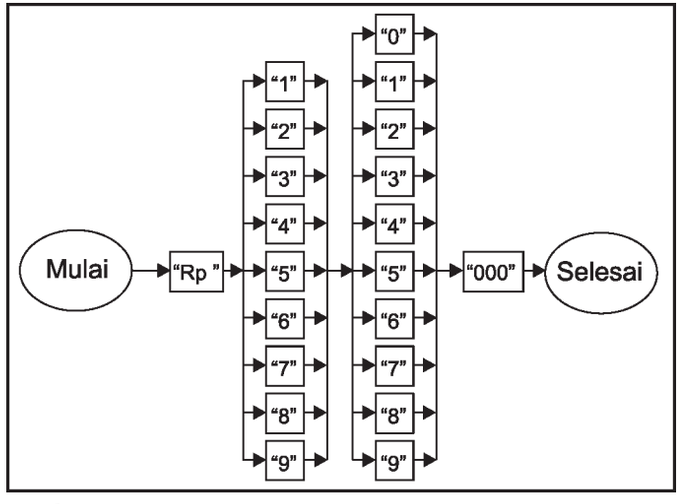
Regex di atas dapat dibaca "dua buah huruf, masing-masing 0 hingga 9 atau a hingga f." Dengan kata lain, regex ini mendeskripsikan string yang mengandung dua buah digit heksadesimal. Tanda minus - menyatakan rentang dan dapat dibaca "hingga." Regex cocok dengan string 01, ff, a5, ffff, zz00zz, tapi tidak cocok dengan string zz, zz0zz. Contoh lain:

```
Rp [1-9][0-9]000,-
```

Regex di atas dapat dibaca "Deretan huruf Rp diikuti sebuah angka 1 hingga 9, diikuti sebuah angka dari 0 hingga 9, diikuti tiga buah angka 0." Dengan kata lain, kita mendeskripsikan sebuah nilai nominal rupiah antara 10 ribu hingga 99 ribu (tanpa titik ribuan). Diagram mesin statenya seperti pada Gambar 1-6. Contoh lain:

```
[a-z][^aeiou][^aeiou][^aeiou][^aeiou][a-z]
```

Regex di atas dapat dibaca sebagai: "huruf yang diikuti empat huruf bukan a/e/i/o/u, diikuti sebuah huruf." Dengan kata lain, kita mencoba mencari empat deretan konsonan atau lebih. Karakter pangkat ^ berarti negasi dan dapat dibaca "bukan", sehingga [^aeiou] berarti bukan a/e/i/o/u (bukan salah satu huruf vokal).



Gambar 1-6.

Beberapa hal yang perlu diperhatikan mengenai kelas karakter. **Pertama**, karakter meta seperti (), |, ., dan lain sebagainya, yang berarti khusus di luar kelas karakter, menjadi karakter biasa di dalam kelas karakter. Artinya, [(a|b)] berarti salah satu karakter dari (, a, |, b,), bukan berarti a atau b. Karakter meta pengelompokan () dan alternasi | menjadi karakter literal biasa. Demikian pula [.] artinya titik, bukan berarti karakter apa saja. Pola regex [.] tidak akan cocok dengan karakter lain selain titik itu sendiri. **Kedua**, karakter tanda minus - dan karakter pangkat ^ menjadi karakter meta di dalam kelas karakter. Di luar kelas karakter, tanda minus bukanlah sebuah karakter meta dan karakter pangkat merupakan karakter meta yang artinya berbeda. Ketiga, tanda minus - pada akhir kelas karakter bersifat literal, contohnya [0-9-] akan cocok dengan salah satu karakter 0 hingga 9 atau dengan tanda minus itu sendiri (11 buah karakter pilihan).

Jangkar (anchor)

Karakter meta ^ berarti awal baris atau awal string. Karakter meta $ berarti akhir baris atau akhir string. Akhir baris atau akhir string bergantung pada mode operasi atau modifier (dijelaskan di subbab berikutnya pada bab ini).
Kedua karakter ini disebut jangkar. Kedua karakter meta ini tidak mewakili sebuah atau sekelompok karakter lain melainkan mewakili posisi. Dengan kata lain kita mewajibkan pengujian string yang ingin kita cocokkan dilakukan pada awal atau akhir string. Beberapa contoh:

```
^satu
```

Pola regex dapat dibaca "baris atau string yang *diawali* dengan deretan huruf satu." Tanpa karakter jangkar ^, regex berbunyi "baris atau string yang mengandung huruf satu." Tanpa jangkar, string satu, kesatuan, satuan, cuma satu akan cocok dengan regex. Dengan jangkar, hanya string satu dan satuan saja yang cocok.

```
satu$
```

Pola regex dapat dibaca "baris atau string yang *diakhiri* dengan deretan huruf satu." Dengan karakter meta jangkar $, hanya satu dan cuma satu yang cocok dengan regex.

```
^satu$
```

Pola regex dapat dibaca "baris atau string yang mengandung deretan huruf satu dan *hanya* deretan huruf satu." Dengan karakter meta jangkar ^ dan $, hanya satu saja yang cocok dengan regex karena string lainnya mengandung deretan huruf satu diikuti dan/atau diawali oleh huruf lain.

Perlu dicatat, $ sebetulnya cocok dengan akhir baris atau "tepat sebelum newline". Jadi jika ada string "satu\n", akan cocok juga dengan pola regex ^satu$.

Quantifier

Karakter meta {} disebut quantifier. Sintaks karakter meta ini:

```
p{m,n}
p{m}
p{m,}
```

di mana p adalah pola atau subpola regex, dan m dan n adalah sebuah bilangan cacah (0, 1, 2, …). Batas m dan n pada banyak implementasi library regex, seperti di Perl atau PCRE, adalah 32767.

Quantifier {m,n} menyatakan bahwa pola p dapat diulangi antara m hingga n kali. {m} menyatakan bahwa pola p harus ada sebanyak tepat m kali. {m,} berarti pola p harus ada *minimal* sebanyak m kali.
Beberapa contohnya:

```
^[0-9]{1,}$
```

Regex dapat dibaca "string atau baris yang merupakan 1 atau lebih angka 0 hingga 9." Regex akan cocok dengan string seperti 1, 123, 00230, 0, 00, tapi tidak cocok dengan 10x (karena mengandung x) atau "" (string kosong, karena quantifier menyatakan minimal 1).

```
^[1-9][0-9]{0,}$
```

Regex dapat dibaca "string atau baris yang merupakan angka 1 hingga 9 diikuti nol atau lebih angka 0 hingga 9." Perbedaan dengan regex sebelumnya adalah, kali ini kita mewajibkan karakter pertama terdiri dari angka 1 hingga 9, sehingga 0 tidak diperbolehkan. Regex tidak akan cocok dengan 00230 (diawali 0), 0 (diawali 0), maupun 10x dan string kosong.

```
b.{3}k
b...k
```

Kedua regex di atas equivalen dan dapat dibaca "string yang mengandung huruf b diikuti tiga buah huruf apa saja dan diikuti huruf k." Regex akan cocok dengan string batuk, b...k, b—k, blocker, tapi tidak akan cocok dengan string bak. Sebab hanya satu huruf di antara b dan k, atau bentrok karena ada 5 huruf mengikuti b sebelum k, atau bisa juga abstract karena tidak ada k mengakhiri tiga buah huruf.

Nol atau satu (optional, zero-or-one)

Karakter meta ? (tanda tanya) berarti quantifier nol atau satu. ? merupakan shortcut untuk {0,1}. Perhatikan bahwa arti ? pada regex berbeda dengan arti ? pada wildcard. ? pada wildcard dapat dinyatakan di regex dengan pola . (titik) yang berarti satu buah karakter apa saja.

Nol atau lebih (zero-or-more)

Karakter meta * (asterisk, bintang) berarti quantifier nol atau lebih. * merupakan shortcut untuk {0,}. Perhatikan bahwa arti * pada regex berbeda dengan arti * pada wildcard. * pada wildcard dapat dinyatakan di regex dengan pola .* yang berarti nol atau lebih karakter apa saja.

Satu atau lebih (one-or-more)

Karakter meta + (tanda plus) berarti quantifier satu atau lebih. + merupakan shortcut untuk {1,}. Beberapa contoh penggunaan ?, *, dan +:

```
[0-9]+
[0-9]{1,}
```

Kedua regex di atas equivalen dan dapat dibaca "deretan digit".

```
.+
.{1,}
```

Kedua regex di atas equivalen dan dapat dibaca "satu atau lebih karakter apa pun." Dengan kata lain, "string yang tidak kosong."

```
.*
.{0,}
```

Kedua regex di atas equivalen dan dapat dibaca "nol atau lebih karakter apa saja." Dengan kata lain, string apa saja cocok dengan regex ini, termasuk string kosong.

```
silah?kan
silah{0,1}kan
```

Kedua regex di atas equivalen dan dapat dibaca "mengandung deretan huruf silakan atau silahkan". Dengan kata lain, huruf h pada silahkan, dinyatakan bersifat opsional.

```
bat.?k
```

Regex di atas cocok dengan batak, batuk, maupun batk. Perhatikan, ada nol karakter antara bat dan k.

```
ma(ri)?lah
```

Regex di atas berbunyi "malah atau marilah". Perhatikan penggunaan karakter meta pengelompokan () untuk membuat ri sebagai satu kesatuan subpola. Tanpa pengelompokan, regex mari?lah berarti "marilah atau marlah". Contoh berikut ini mendemonstrasikan pengelompokan yang lebih kompleks.

```
((do|re|mi|fa|so|la|si)[ -])+(do|re|mi|fa|so|la|si)
```

Regex di atas dapat dibaca “deretan teks nada do re mi yang dipisahkan dengan tanda minus atau spasi, minimal terdiri dari 2 nada.” Contoh string yang cocok dengan regex ini adalah so-so-mi-mi-re-re-do-re-re-mi (kuis trivia: potongan lagu **Phil Collins** manakah deretan nada tersebut?). Contoh string yang tidak cocok dengan regex, misalnya mi, karena hanya terdiri dari satu nada, subpola (do|re|mi|fa|so|la|si) terakhir pada pola regex tidak terpenuhi atau sosomimireredoreremi, karena antarnada tidak dipisahkan dengan spasi atau minus.

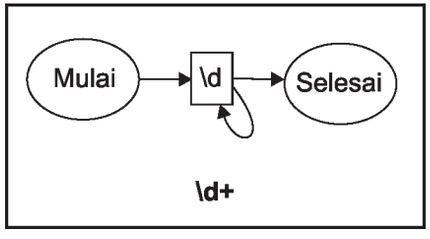
```
h?(ah|eh)+!*
```

Regex di atas berbunyi “boleh diawali huruf h, terdiri dari deretan ah atau eh, dan boleh diakhiri dengan deretan tanda seru.” Cocok dengan string seperti eh, hahahah!, atau ehehahehah!!!. Tidak cocok dengan string seperti haha, hhah!, maupun h!.

```
selama(-lama)*nya
```

Regex di atas cocok dengan string selamanya, selama-lamanya, selama-lama-lamanya, dan seterusnya.

Dalam diagram mesin state, {1,} atau + dapat dilambangkan dengan jalur looping (lihat Gambar 1-7). Sementara {0,1} atau ? dapat dilambangkan dengan jalur shortcut (lihat Gambar 1-8).



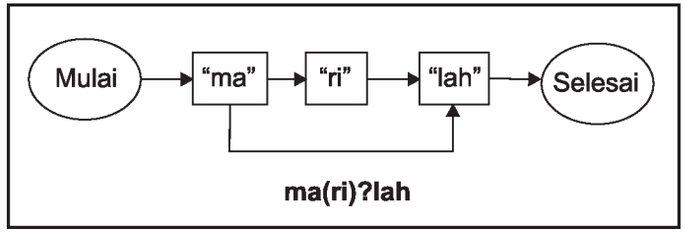
Gambar 1-7.

Kelas karakter: digit

Karakter meta \d berarti digit mana saja. Karakter meta ini equivalen dengan pola [0-9], kecuali jika pada *locale* tertentu yang termasuk dalam digit bukan 0-9 saja.
Karakter meta komplemennya adalah \D, yang berarti bukan karakter digit. Equivalen dengan [^0-9], kecuali jika pada locale tertentu yang digitnya bukan 0-9.

```
\D\d+\D
```

Regex di atas dapat dibaca “deretan angka yang diapit oleh bukan angka”, cocok dengan string seperti x123x atau “ 123 “ (diapit oleh spasi).
Dianjurkan memakai \d dan \D ketimbang equivalennya di atas sebisa mungkin karena lebih singkat dan lebih abstrak sehingga dapat menyesuaikan karakter locale.



Gambar 1-8.

Kelas karakter: karakter dalam kata

Karakter meta \w berarti karakter yang termasuk dalam kata. Dalam locale seperti Inggris dan Indonesia, equivalen dengan [0-9A-Za-z_] (alfanumerik dan underscore). Dalam locale lain bisa saja mencakup karakter kata lain seperti huruf-huruf yang mengandung umlaut atau aksen dalam bahasa Eropa.
Karakter meta komplemennya, \W, berarti karakter yang tidak termasuk dalam kata. Dalam locale seperti Inggris dan Indonesia, equivalen dengan [^0-9A-Za-z_]. Contoh:

```
\w+
```

Regex di atas bisa dipakai untuk mencocokkan sebuah “kata”, meskipun sebetulnya yang dinamakan “kata” dalam bahasa manusia kadang-kadang lebih kompleks dari ini (dibahas di Katalog Pola bagian Kata).

Dianjurkan memakai \w dan \W ketimbang equivalen-nya di atas sebisa mungkin, karena lebih singkat dan abstrak (dapat menyesuaikan terhadap karakter locale).

Kelas karakter: spasi (whitespace)

Karakter meta \s berarti whitespace. Yang termasuk whitespace adalah spasi, tab, dan newline. Namun cocok tidaknya dengan newline bergantung pada mode operasi regex, akan dibahas pada subbab Mode Operasi. Jika newline cocok, maka \s equivalen dengan [\t\n] (spasi, tab, dan newline). Jika newline tidak dianggap cocok, maka \s equivalen dengan [\t].

Komplemen untuk \s adalah \S, yang berarti non-whitespace. Equivalen dengan [^ \t\n] atau [^ \t], bergantung pada mode operasi. Contohnya:

```
^\S
```

Pola regex di atas dapat digunakan untuk mencari baris yang tidak diawali spasi atau tab, dengan kata lain “baris yang tidak kosong dan tidak diindentasi.”

```
aku\s+dan\s+dia
```

Pola regex di atas dapat digunakan untuk mencocokkan tiga buah kata aku, dan, dia yang dipisahkan oleh whitespace. Pola regex ini sedikit lebih fleksibel daripada pola aku dan dia yang hanya mengizinkan satu buah spasi di antara tiap kata.

Batas antarkata (word boundary)

Karakter meta \b tidak mewakili karakter tertentu, melainkan mewakili posisi batas antarkata. Akhir atau awal string dianggap juga merupakan batas antarkata. Kita dapat menggunakan \b di akhir maupun di awal deretan huruf dalam pola untuk menyatakan deretan huruf ini haruslah merupakan bagian akhir dan/atau awal sebuah kata. Beberapa contoh:

```
^Dari\b
```

Pola di atas akan cocok baris yang diawali *kata* Dari. Regex tidak akan cocok dengan Darius atau Darimana, tapi akan cocok dengan Dari mana, Dari Sabang sampai Merauke, Dari (Dari diikuti spasi), maupun Dari (hanya Dari).

```
\bteh\b
```

Pola di atas mencoba mencari kata teh, tapi bukan string teh yang terkandung dalam kata lain seperti teheran atau Whitehall. Biasanya kata teh dalam bahasa Inggris sudah hampir pasti merupakan salah eja dari the. Komplemen dari karakter meta \b adalah \B, yang cocok dengan posisi bukan di batas antarkata. Perhatikan sekali lagi bahwa \b dan \B tidak mewakili karakter melainkan hanya posisi atau antarkarakter. Kadang-kadang karakter meta seperti ini disebut *zero-width assertion*. Karakter jangkar ^ dan $ termasuk contoh lain zero-width assertion.

Awal dan akhir string atau baris

Karakter meta \A, dan \Z, \z mirip dengan karakter jangkar ^ dan $, hanya saja merupakan versi yang tidak bergantung pada mode operasi dan artinya sedikit berbeda. \A selalu cocok dengan awal string, bukan hanya awal baris. \Z akan selalu cocok dengan akhir string (bukan hanya akhir baris) atau sebelum newline—sama dengan kelakuan $ pada mode bukan m—sementara \z hanya cocok dengan akhir string (dan tidak akan cocok dengan sebelum newline di akhir string). Contoh, jika string adalah “baris 1\nbaris 2\nbaris3\n” maka regex-regex berikut akan cocok:

```
\Abaris
\Abaris 1
baris 3\Z
```


Regex-regex berikut tidak cocok:

```
\Abaris 2                # karena "baris 2" bukan di awal string
baris 2\Z                # karena "baris 2" bukan di akhir string
baris 2\z                # karena "baris 2" bukan di akhir string
baris 3\z        # karena "baris 3" mengandung \n seblm akhir
```

Escape Karakter Meta

Satu pertanyaan yang mungkin ada di benak Anda ketika membaca tentang berbagai karakter meta regex yaitu: bagaimana jika kita ingin mencocokkan karakter seperti ^, $, ., [], dsb secara literal? Dengan kata lain, kita ingin mencari karakter itu sendiri, bukan ingin menggunakan karakter metanya. Caranya adalah dengan *meng-escape* karakter meta tersebut, mengawalinya dengan \ (backslash). Beberapa contoh:

```
\(.+\)
\([^)]*\)
```

Kedua regex di atas hampir equivalen, keduanya ingin mencari teks yang diapit oleh tanda kurung, misalnya: satu (dua) tiga. Karena (dan) adalah karakter meta, maka kita perlu mengescapenya jika menginginkan mencocokkan karakter (dan) itu sendiri. Jika kita tidak mengescape regex, (.+), maka regex ini akan mencari satu atau lebih karakter apa saja, tak peduli apakah terdapat kurung buka atau kurung tutup pada teks tersebut.

Pada regex kedua, seperti telah dijelaskan sebelumnya, karakter meta seperti () dianggap literal jadi tidak perlu diescape lagi (meskipun jika di-escape hasilnya akan sama).

```
[a-z]\^\d+
```

Pola regex di atas akan mencari potongan teks seperti a^2, x^3, dan seterusnya. Berbeda dengan yang berikut ini:

```
[0-9]+(\.[0-9]+)?
```

Pola regex di atas berbunyi “satu atau lebih angka 0-9 yang dapat diikuti titik dan deretan angka lagi.” Pola regex ini akan cocok dengan string seperti 123, 1.01, 0.0. Karena kita bermaksud mencari titik itu sendiri, maka kita perlu menuliskan \. (backslash diikuti titik) dan bukan hanya . (titik).

Escape karakter non-printing/kontrol

Selain untuk mengescape karakter meta, backslash juga seperti di C/PHP/Perl digunakan untuk menspesifikasikan karakter-karakter yang tidak dapat dicetak (non-printing) dan karakter kontrol:

`\x00 s.d. \xff`	Karakter ASCII 0 hingga 255 (dinyatakan dalam heksadesimal)
`\000 s.d. \377`	Karakter ASCII 0 hingga 255 (dinyatakan dalam oktal)
`\t`	Tab (\011)
`\n`	Newline (\012 atau \015\012, bergantung OS Unix atau Windows)
`\r`	Carriage return (\015)
`\f`	Form feed (\014)
`\a`	Bell (\007)
`\e`	Escape (\033)

Untuk mencari backslash itu sendiri digunakan \\.

Escape pola

Bila Anda ingin mengescape lebih dari satu karakter sekaligus, maka menggunakan backslash terus-menerus mungkin menjemukan dan membuat pola sulit dibaca. Alternatifnya, di Perl, Ruby, dan PCRE 4.x (contohnya PHP 4.3.0) disediakan karakter meta \Q dan \E, yang berguna untuk mengapit subpola yang ingin dijadikan literal.

Di antara kedua pengutip ini, semua karakter meta menjadi kehilangan maknanya. Contoh:

```
...?
\Q...?\E
```

Pola pertama berarti dua atau tiga karakter apa saja, sementara pola kedua mencari string "...?" apa adanya.

Seperti akan Anda lihat di Bab 3, \Q dan \E berguna jika Anda ingin memasukkan variabel di dalam sebuah pola dan Anda ingin isi variabel tersebut dicari secara literal. Dengan kata lain, abaikan semua karakter meta yang mungkin ada di dalam variabel tersebut.

Latihan 1

Latihan dalam mempelajari sebuah bahasa baru amatlah penting, karena itu subbab ini berisi contoh-contoh dalam bentuk pertanyaan dan jawaban.

Jawaban latihan ada di subbab Jawaban Latihan 1.

Regex dan wildcard

1. Apa wildcard dari "semua nama file berekstensi apa saja"? Bagaimana jika dinyatakan dalam regex?
2. Apa wildcard dari "semua file ZIP yang berawalan A, B, dan C"? Bagaimana jika dinyatakan dalam regex?
3. Apakah wildcard F*.* akan cocok dengan nama file FileSaya di DOS/Linux? Bagaimana dengan f*?

Karakter meta 1

1. Seperti apa regex yang berbunyi "satu hingga tiga deret karakter a baik huruf besar maupun kecil"?
2. Bagaimana mencocokkan karakter yang bukan huruf kecil?
3. Bagaimana mencocokkan string angka 15 hingga 51 dengan regex?
4. Bagaimana mencocokkan dengan tepat string-string anting, ranting, dan ranjang dengan seringkas-ringkasnya?
5. Apakah [AS][US][AS] cocok dengan USA?
6. Apakah (Mega|Rahma)wati cocok dengan Ibu Megawati? Dengan Mega?
7. Apakah Saya(bukan)? pembohong. cocok dengan saya pembohong.?

Karakter meta 2

1. Bagaimana pola sebuah bilangan bulat positif? Bulat negatif? Desimal positif/negatif?
2. Bagaimana regex yang berbunyi "mengandung string aku"?
3. Bagaimana regex yang berbunyi "tepat string aku"?
4. Bagaimana regex yang berbunyi "diakhiri string aku"?
5. Bagaimana regex untuk mencocokkan nama variabel dalam PHP atau variabel skalar Perl?
6. Apakah ^(Mega|Rahma)wati cocok dengan Ibu Megawati?
7. Apakah [^k]apuk cocok dengan bunga kapuk yang mulai lapuk?
8. Apa pola untuk mencari string *** (tiga buah bintang) di dalam sebuah string?

Jawaban Latihan 1

Regex dan wildcard

1. Wildcard dari "semua nama file berekstensi apa saja" adalah *.*. Jika dinyatakan dalam regex, polanya adalah ^.*\..*$ (dibaca, "deretan nol atau lebih karakter apa saja, diikuti sebuah karakter titik, diikuti nol atau lebih karakter apa saja"). Sebetulnya dengan regex \. saja kita sudah dapat melakukan hal yang sama, karena tanpa jangkar ^ dan $, mesin regex tidak peduli menemukan karakter titik di awal, di tengah, atau di akhir string.
2. Wildcard dari "semua file ZIP yang berawalan A, B, C" adalah [ABC]*.zip. Wildcard di Unix/Linux mengenal kelas karakter [], sementara di DOS/Windows tidak. Jadi di DOS/Windows kita tidak dapat menyatakan pola ini dengan satu wildcard tunggal melainkan harus dengan tiga wildcard A*.zip, B*.zip, dan C*.zip. Versi regexnya adalah ^[ABC].*\.zip$.
3. Pertanyaan ketiga sedikit menjebak. Kelakuan wildcard di Linux dan di DOS agak berbeda. Wildcard pertama, F*.* akan cocok dengan file FileSaya di DOS. Namun di Linux, wildcard ini berarti "File yang diawali huruf F dan mengandung titik di tengahnya." Jadi jika nama file FileSaya.doc misalnya, maka wildcard akan cocok di Linux. Tapi jika tanpa ekstensi—yang berarti tanpa karakter titik—maka tidak akan cocok. Wildcard kedua pun hanya cocok di DOS. Ingat, Unix membedakan huruf besar dan kecil. Karena wildcard diawali huruf f kecil, maka tidak akan cocok di Linux.

Karakter meta 1

1. "Satu hingga tiga deret karakter a baik huruf besar maupun kecil" dapat dinyatakan dengan regex [Aa][Aa]?[Aa]? atau [Aa]{1,3}. Jika mode operasi pencocokan regex adalah case-insensitive, maka kita bisa juga membuat pola A{1,3} karena a pun akan cocok dengan A pada mode case-insensitive.
2. Untuk mencocokkan karakter yang bukan huruf kecil, gunakan pola [^a-z]. Catatan, ini berarti angka, simbol, maupun huruf besar cocok dengan pola ini. Karena semuanya bukan huruf kecil.
3. Untuk mencocokkan string angka 15 hingga 51 kita dapat menggunakan pola 1[1-5]|[234][0-9]|5[01]. Pola ini berbunyi "11 hingga 15 *atau* 20 hingga 49 *atau* 50 atau 51." Jika Anda menjawab pola 15-51 maka Anda benar-benar tidak menyimak bab ini!
4. Untuk mencocokkan beberapa buah string kita dapat menggunakan pola seperti ^(anting|ranting|ranjang)$, tapi tentu saja yang diminta adalah versi yang lebih singkat. Kita dapat menggunakan ^(r?anting|ranjang)$. Pola ^(r?an(ting|jang))$, meskipun lebih ringkas, tidak dapat digunakan karena cocok juga dengan anjang, padahal itu tidak kita inginkan.
5. [AS][US][AS] tidak cocok dengan USA.

6. (Mega|Rahma)wati cocok dengan Ibu Megawati, tapi tidak dengan Mega, karena string yang kedua tidak mengandung wati.
7. Jawaban untuk pertanyaan terakhir bergantung pada mode operasi. Saya(bukan)? pembohong. tidak cocok dengan saya pembohong. jika mode operasi adalah case-insensitive (default). Tapi cocok jika mode operasi menggunakan case-insensitive.

Karakter meta 2

1. Untuk mencocokkan bilangan bulat positif, gunakan regex [0-9]+. Untuk bilangan bulat negatif: -[0-9]+. Sebuah bilangan positif juga dapat diberi tanda positif di depan angkanya. Karena + adalah sebuah karakter meta, maka kita dapat memasukkannya ke dalam set atau mengawalinya dengan \ (garis miring terbalik, backslash) untuk membuatnya menjadi bermakna literal. Contoh: [+]?[0-9]+ atau \+?[0-9]+. Kedua pola di atas dapat digabung menjadi pola untuk "bilangan bulat": [+-]?[0-9]+. Bilangan desimal adalah bilangan bulat yang diikuti koma desimal. Polanya adalah sebagai berikut: [+-]?[0-9]+(,[0-9]+)? Atau bisa juga ditulis tanpa pengelompokan dengan pola ini: [+-]?[0-9]+,?[0-9]*. Perhatikan bahwa koma desimal bersifat opsional. (Lihat juga pembahasan yang lebih rinci di Katalog Pola bagian Bilangan.)
2. Untuk mencocokkan string yang mengandung aku, cukup gunakan pola aku.
3. Untuk mencocokkan string yang tepat berisi aku (tanpa tambahan apa-apa lagi), polanya adalah ^aku$.
4. Untuk mencocokkan string yang diakhiri aku, gunakan pola aku$.
5. Sebuah variabel dalam PHP diawali dengan karakter $ dan diikuti dengan huruf besar, huruf kecil, atau garis bawah (_) namun tidak boleh angka. Huruf kedua dan seterusnya dapat berupa huruf, garis bawah, atau angka. Sebuah variabel dapat terdiri dari satu atau lebih karakter. Polanya adalah sebagai berikut: \$[A-Za-z_][A-Za-z_0-9]*.
6. ^(Mega|Rahma)wati tidak akan cocok dengan string Ibu Megawati karena pola berjangkar ^ sehingga potongan Mega atau Rahma harus terdapat di awal string yang ingin dicocokkan.
7. [^k]apuk cocok dengan string bunga kapuk yang mulai lapuk. Tepatnya, cocok dengan lapuk.
8. *{3} atau *** atau \Q***\E. Pola yang terakhir saat ini hanya didukung oleh Perl, Ruby, dan PCRE 4.x.

Mode Operasi

Dalam kompilasi dan/atau eksekusi pola regex, mesin regex dapat menerima beberapa flag yang disebut mode operasi atau modifier. Keberadaan flag-flag ini dapat mengubah kelakuan karakter-karakter meta tertentu.

Berikut ini mode-mode operasi yang dikenal:

Modifier i (case Insensitive)

Secara default, mesin regex melakukan pencocokan secara case sensitive—membedakan huruf besar dan huruf kecil. Tapi dengan modifier i, pencocokan akan dilakukan secara case-insensitive. Dengan kata lain, pada mode i regex [A-Z] atau [a-z] akan berlaku sama seperti [A-Za-z].

Mode case-insensitive sedikit lebih lambat daripada mode case-sensitive, tapi dalam hampir semua kasus Anda tidak perlu mengkuatirkan hal ini.

Modifier s (Single line)

Secara default, ketika menjumpai karakter newline dalam string, karakter ini tidak akan dianggap cocok dengan . (titik). Dengan kata lain, karakter meta . berarti cocok dengan karakter apa saja *kecuali newline*. Namun dengan mode s, . benar-benar berarti cocok dengan karakter apa saja, *termasuk newline*.

Catatan: Di Ruby modifier s digantikan dengan m dan s memiliki arti yang lain. Lihat Bab 3 subbab Ruby untuk jelasnya.

Modifier m (Multiline)

Secara default, ^ dan $ berarti awal dan akhir *string* (kecuali di Ruby, di mana ^ dan $ selalu berarti awal dan akhir *baris*, lihat Bab 3 subbab Ruby untuk jelasnya).

Ini berarti jika ada string multibaris sebagai berikut:

```
Aku seorang kapiten,
memiliki pedang panjang,
kalau berjalan prok, prok, prok,
aku seorang kapiten.
```

maka pola regex berikut tidak akan cocok:

```
^memiliki
```

karena kata memiliki meskipun terletak di awal baris namun ada pada baris kedua dan bukan baris pertama (bukan di awal string).

Demikian pula pola berikut tidak akan cocok:

```
kapiten,$
```


karena kata kapiten yang diikuti koma (,) terdapat di akhir baris pertama dan bukan di akhir baris terakhir (bukan benar-benar di akhir string).
Agar kedua regex di atas menjadi cocok, kita dapat menggunakan modifier m agar ^ dan $ menjadi memiliki arti awal dan akhir *baris*, bukan awal dan akhir *string*.

Modifier c

Hanya ada/relevan di Perl. Lihat Bab 3 subbab Perl.

Modifier g (Global)

Hanya ada/relevan di Perl. Lihat Bab 3 subbab Perl.

Modifier o (Once)

Hanya ada/relevan di Perl. Lihat Bab 3 subbab Perl.

Modifier e (Eval)

Tidak semua mesin regex memiliki/memerlukan modifier ini. Modifier ini hanya relevan saat melakukan substitusi regex. Contoh yang mendukung modifier e adalah Perl dan PHP. Ini akan dibahas di Bab 3 subbab Perl dan PHP.

Modifier x (eXtended legibility)

Kalau Anda seorang pemula yang baru saja membaca bab ini, maka Anda akan melihat bahwa pola regex memang sulit dibaca. Penyebabnya adalah: 1) banyak mengandung karakter-karakter nonhuruf dan nonangka, seperti tanda baca, asterisk, pangkat, berbagai macam tanda kurung (kurung kurawal, kurung siku, kurung biasa), dsb; 2) banyaknya jenis karakter meta yang harus diingat/dihafalkan; 3) amat ringkas, setiap potongan atau huruf—termasuk spasi—mengandung makna, mengakibatkan regex tampak terkompresi dan berjejal-jejal.

Nomor 1 dan nomor 2 memang tidak dapat dihindarkan, karena memang inti kekuatan dari regex adalah karakter-karakter meta itu sendiri, dan untuk menghindari bentrokan dengan huruf-huruf dan angka yang ingin kita cocokkan, karakter meta dipilih memakai berbagai tanda baca dan karakter aneh lainnya.

Tapi untuk nomor 3, sebetulnya ada alternatifnya. Kita dapat memakai modifier x agar mesin regex mengabaikan spasi/tab/newline (whitespace) dan mengizinkan komentar. Modifier x memungkinkan kita menuliskan sebuah pola regex dengan disisipi spasi bahkan newline sehingga pola dapat lebih tampak "jarang-jarang" dan lebih rapi dalam beberapa baris disertai indentasi. Plus adanya komentar memungkinkan kita memberi penjelasan apa arti sebuah subpola dan mengapa kita menuliskannya demikian. Kadang-kadang modifier x bukan hanya membantu tapi bisa dibilang wajib, apalagi untuk regex kompleks yang sulit dibaca.

Contohnya:

```
^[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?$
```

Seseorang yang ahli pun membutuhkan waktu beberapa saat untuk mengetahui apa sebetulnya maksud dari regex ini. Apalagi pemula! Kita tidak dapat memberikan spasi atau newline pada pola regex di atas jika kita tidak menggunakan modifier x, karena akan mengubah arti pola. Tapi jika kita menggunakan modifier x, pola regex dapat kita ubah menjadi sebagai berikut tanpa mengubah arti pola:

```
^
  [+-]?           # pertama, cocokkan tanda positif/negatif jika
ada
  (               # lalu cocokkan mantissa
      \d+\.\d+    #   dalam bentuk a.b
     |\d+\.       #   dalam bentuk a.
     |\.\d+       #   dalam bentuk .b
     |\d+         #   dalam bentuk bil bulat
  )
  ([eE][+-]?\d+)?  # terakhir, cocokkan eksponen jika ada
$
```

Terasa sekali bedanya bukan? Dengan modifier x, kita dapat membuat pola regex seperti sebuah bahasa pemrograman biasa yang disertai komentar, indentasi, dan bahkan baris-baris kosong kalau perlu. Kita jadi lebih jelas mengerti apa maksud dari regex tersebut, yaitu untuk mencocokkan string bilangan real baik dalam bentuk biasa seperti +1.234 maupun dalam bentuk eksponen seperti -1.37e8. Dalam mode x, karakter # dianggap sebagai karakter meta yang berarti bahwa semua sisa karakter hingga akhir baris merupakan komentar yang dapat diabaikan. Anda perlu "berhati-hati" menggunakan modifier x, artinya, kehadiran modifier x dapat membuat sebuah pola regex diartikan berbeda sama sekali. Terutama, Anda harus selalu menuliskan spasi sebagai \s karena spasi literal akan diabaikan di mode x. Karakter # juga menjadi karakter meta di mode x.

Modifier A (Anchored)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl.

Kehadiran modifier ini membuat mesin regex menganggap semua pola diapit dengan karakter jangkar. Artinya, saat Anda memberi pola seperti a.+b, mesin regex akan menganggapnya sebagai ^a.+b$. Saya jarang melihat modifier ini dipakai.

Modifier D (Dollar End Only)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl.

Kehadiran modifier ini membuat mesin regex menganggap $ sebagai karakter meta (yakni jangkar) hanya pada akhir string. Karakter $ di tengah-tengah string akan dianggap sebagai $ biasa. Sebetulnya idenya bagus, karena praktis $ sebagai jangkar tidak pernah disebutkan di tengah-tengah string. Sayangnya ini bukan kelakuan rata-rata mesin regex. Jadi saya pun jarang melihat modifier ini dipakai dalam kebanyakan program.

Modifier S (Study)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl. Di Perl terdapat analogi yang mirip yaitu fungsi study. Dengan modifier ini, mesin regex akan melakukan tahap-tahap tambahan untuk membuat agar mesin state hasil kompilasi pola regex dapat bekerja lebih cepat. Kompilasi akan berjalan lebih lama, tapi pencocokan berjalan lebih cepat.

Kecuali pola regex Anda kompleks dan pencocokan berjalan lambat dan Anda sedang melakukan optimasi, biasanya Anda tidak perlu mengkuatirkan mengenai modifier S atau study.

Modifier U (Ungreedy)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl.

Dengan modifier ini, mesin regex akan membalikkan arti greediness dari pola-pola seperti .+, .+?, .*, .*?, dsb. Semua yang tadinya greedy secara default akan menjadi non-greedy, dan sebaliknya.

Greediness dibahas di bawah.
Saya juga jarang melihat modifier U dipakai orang.

Modifier X (eXtra)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl.

Dengan modifier ini, jika mesin regex melihat adanya backslash pada huruf atau simbol yang tidak dikenal, maka mesin regex akan melaporkan pesan kesalahan. Misalnya, \I atau \i tidak dikenal sementara \A atau \z dikenal. Defaultnya, tanpa modifier ini \F akan dianggap sebagai "F" literal tanpa kesalahan.

Modifier u (UTF8)

Catatan: modifier ini hanya dikenal di library PCRE (mis: di fungsi preg_*() PHP) dan saat ini tidak ada di Perl.

Dengan modifier ini, string yang ingin dicocokkan dianggap merupakan string UTF8 (UTF8 adalah set karakter untuk menulis Unicode dan merupakan superset ASCII 7-bit). Jika Anda hanya memproses teks dalam bahasa Inggris/Indonesia tanpa melibatkan karakter-karakter ASCII di atas 127, maka Anda tidak perlu mengkuatirkan masalah ini.

Match group dan Backtracking

Kita akan mulai membahas fitur-fitur advanced regex yang benar-benar membuatnya jauh lebih ampuh dari sekadar wildcard. Regex lebih dari sekedar menguji sebuah string terhadap pola dan mengeluarkan jawaban akhir ya atau tidak. Kita juga dapat mengambil sebagian atau keseluruhan string yang cocok dengan subpola untuk kita pakai semau kita (ekstraksi).

Untuk mengambil atau mengingat sebuah subpola, digunakan karakter meta tanda kurung (). Anda tentu ingat bahwa tanda kurung juga digunakan untuk pengelompokan. Subpola yang sudah terambil akan diingat oleh mesin regex dan dapat dipanggil kembali dengan menggunakan karakter meta \1, \2, dan seterusnya. (\1 untuk subpola pertama yang diambil, \2 untuk subpola kedua, dan seterusnya.) Hasil tangkapan \1, \2, dan seterusnya. disebut dengan match group.

Pola regex aku|kamu dan (aku|kamu) dalam hal pencocokan biasa tidak ada bedanya; keduanya akan sama-sama menghasilkan jawaban ya atau tidak untuk string yang sama (mis: sama-sama cocok untuk kaku dan tidak cocok untuk kau). Tapi karena pola kedua menangkap match group, maka jika terjadi kecocokan \1 akan diisi dengan aku atau kamu—bergantung pada mana yang tertangkap—sementara di pola pertama tidak ada apa-apa yang tertangkap.
Yang membuat menarik/ampuh, \1 juga dapat disebutkan di subpola berikutnya. Contoh:

```
(\w+)-\1
```

Pola di atas mencari kata ulang, yaitu dua buah kata sama yang dipisahkan dengan tanda strip, contohnya: mata-mata atau kuchi-kuchi hota hae. Bedakan pola tersebut dengan pola ini:

```
\w+-\w+
```


Pola di atas hanya mencari dua buah kata yang dipisahkan dengan tanda setrip, tidak peduli pada apakah kedua kata tersebut sama atau tidak. Jadi, warna-warni atau warna-kuning akan cocok dengan pola ini, tapi tidak akan cocok dengan pola sebelumnya. Pola sebelumnya hanya akan cocok dengan string seperti warna-warna atau warni-warni. Contoh lain:

```
(\w+)-(\w+) \1-\2 \1-\2
```

Pola di atas akan cocok dengan do-re do-re do-re atau do-do do-do do-do atau warna-warni warna-warni warna-warni tapi tidak akan cocok dengan do-re mi-do re-mi atau warna-warna kelap-kelip getar-getir. Dengan kata lain, pola di atas berkata “carilah tiga pasang kata yang dipisahkan setrip”.
Satu contoh lain:

```
(.)\1\1
```

Pola di atas ingin mencari tiga buah karakter sama yang berada berdampingan, contohnya pada string: aaa, heeelp!, atau =======.

Non-Greedy Matching

Non-greedy matching adalah sebuah fasilitas yang pertama kali diperkenalkan dan dipopularkan di regex flavor Perl. Defaultnya, pencocokan oleh mesin regex akan dilakukan *sebanyak* mungkin. Misalnya, jika ada string:

```
Aku seorang kapiten, memiliki pedang panjang.
```

maka pola berikut:

```
Aku(.+)a
```

akan menghasilkan \1 sebagai berikut:

```
 seorang kapiten, memiliki pedang panj
```

ini karena mesin regex akan “melahap” (karena itu muncul julukan greedy) semua teks yang ada sehingga huruf a terakhir, di mana setelah itu tidak ada lagi huruf a yang tersisa. Kadang-kadang, kelakuan ini tidak kita kehendaki. Kadang-kadang kita ingin melakukan pencocokan *sesedikit* mungkin.

Misalnya, pada kasus di atas kita ingin mengambil potongan string setelah Aku hingga sebelum huruf a berikutnya, bukan huruf a terakhir.

Perl memperkenalkan varian-varian quantifier sebagai berikut: *?, +?, ??, {n}?, {n,}?, {n,m}?.
Quantifier ini sama seperti sepupunya *, +, ?, {n}, {n,}, dan {n,m} tapi bersifat non-greedy. Jadi dengan pola:

```
Aku(.+?)a
```

hasilnya \1 akan bernilai:

```
 seor
```

Tanpa quantifier non-greedy, kita masih bisa melakukan yang kita inginkan dengan menggunakan pola:

```
Aku([^a]+)a
```

tapi dengan menggunakan quantifier non-greedy kadang pola lebih mudah ditulis. Apalagi jika melibatkan lebih dari satu karakter. Misalnya kita ingin mencari komentar HTML:

```
<!–(.*)–>
```

Jika kita memiliki HTML seperti ini:

```
Contoh HTML yang <!– diselipi –> komentar <!– dan lagi –>.
```

maka yang akan tertangkap di \1 adalah:

```
diselipi –> komentar <!– dan lagi
```

karena .* bersifat greedy dan akan mencari hingga ditemukannya —> terakhir. Lain halnya jika kita menggunakan .*? yang non-greedy:

```
<!–(.*?)–>
```

hasilnya pada \1 adalah yang sebagaimana mestinya:

```
 diselipi
```

yaitu .*? akan mencari hingga —> yang pertama ditemukan karena .*? mencoba mencocokkan sesedikit mungkin.

Versi non-greedy biasanya cocok di tengah-tengah pola dan bukan di akhir. Penggunaan quantifier versi non-greedy biasanya tidak begitu berguna, contohnya:

```
Cont(.*?)
```

Anda mungkin mengharapkan sisa karakter masuk semua ke dalam \1 bukan? Salah, yang terambil hanyalah 0 karakter. Ini karena .*? berarti mencocokkan 0 atau lebih karakter apa saja, dan yang dipilih adalah yang paling sedikit yaitu 0 karakter. Sementara pola berikut:

```
Cont(.+?)
```

akan menghasilkan 1 karakter saja pada \1 yaitu huruf o, karena .+? berarti cocokkan 1 atau lebih karakter apa saja, dan yang dipilih adalah yang paling sedikit yaitu 1 karakter saja.

Catatan: library regex PCRE mengenal modifier U untuk membalikkan arti greediness. Artinya, .+ menjadi non-greedy sementara .+? menjadi greedy, dsb. Namun saya jarang melihat ini dipakai.

Karakter meta advanced

Non-capturing grouping

Karakter meta (?:...) fungsinya sama dengan (...), tapi perbedaannya adalah dengan (?:...) mesin regex tidak akan melakukan penangkapan/pengingatan terhadap match group. Jadi (?:...) hanya berfungsi untuk pengelompokan. (?:...) berguna agar \1, \2, dsb menjadi subpola yang Anda inginkan, contohnya:

```
((\d\d\d)-(\d\d\d\d))
(?:(\d\d\d)-(\d\d\d\d))
```

Jika string yang ingin dicocokkan adalah 985-7721, maka jika kita menggunakan pola pertama, \1 akan menjadi 985-7721, \2 menjadi 985, \3 menjadi 7721. Sementara jika kita menggunakan pola kedua, \1 menjadi 985 dan \2 menjadi 7721.

Positive look-ahead

(?=...) adalah salah satu zero-width assertion yang memberitahu mesin regex untuk hanya mencari pola yang *diikuti* oleh pola tertentu. Contohnya:

```
\w+(?=\s\d)
```

untuk mencari kata (\w+) yang diikuti oleh spasi dan angka.

Positive look-behind

(?<=...) adalah salah satu zero-width assertion yang memberitahu mesin regex untuk hanya mencari pola yang *didahului* oleh pola tertentu.
Contohnya:

```
(?<=\d\s)\w+
```

untuk mencari kata (\w+) yang didahului oleh angka dan spasi.

Negative look-ahead

(?!...) adalah salah satu zero-width assertion yang memberitahu mesin regex untuk hanya mencari pola yang *tidak diikuti* oleh pola tertentu. Contohnya:

```
\w+(?!=\s\d)
```

untuk mencari kata (\w+) yang tidak diikuti oleh spasi dan angka.

Negative look-behind

(?<!...) adalah salah satu zero-width assertion yang memberitahu mesin regex untuk hanya mencari pola yang *tidak didahului* oleh pola tertentu. Contohnya:

```
(?<!\d\s)\w+
```

untuk mencari kata (\w+) yang tidak didahului oleh angka dan spasi.

Manfaat look-ahead dan look-behind akan dijelaskan dalam contoh resep di Bab 4, misalnya di resep 1-2 dan 2-6.

Kondisi

Karakter meta (?(condition)POLA) dan (?(condition)POLA1|POLA2) berarti pola bersifat kondisional. Sintaks yang pertama berarti, *POLA* hanya dipakai jika condition bernilai true. Sementara sintaks yang kedua berarti, gunakan *POLA1* jika condition bernilai true dan *POLA2* jika false. Sintaks yang kedua mirip dengan operator terner ? ... : ... seperti di C/Perl/PHP.

condition dapat berupa angka (yang berarti match group, mis 1 berarti \1) atau asersi positive/negative look ahead/look behind. Contoh:

```
(\()? [^()]+ (?(1)\))
```


Subpola (\()? berarti mencocokkan tanda kurung buka jika ada. Subpola berikutnya [^()]+ mencocokkan deretan huruf yang bukan tanda kurung. Subpola ketiga (?(1)\)) bersifat kondisional. Jika \1 ditangkap (ada tanda kurung buka), maka wajibkan ada kurung tutup. Jika \1 kosong, maka tanda kurung tutup tidak perlu ada. Contoh berikut diambil dari manual PHP dan ditulis dalam mode x yang berarti whitespace tidak berarti apa-apa:

```
(? (?=[^a-z]*[a-z])
    \d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

Contoh tersebut berarti "jika dijumpai karakter bukan huruf yang diikuti oleh huruf, maka pakai alternatif pola pertama: \d{2}-[a-z]{3}-\d{2}. Jika look-ahead gagal maka gunakan alternatif pola kedua: \d{2}-\d{2}-\d{2}." Artinya, pola dapat mencocokkan tanggal dengan format seperti 12-des-03 atau 12-12-03. (Tentu saja, khusus pada contoh ini, Anda dapat juga menggunakan pola seperti \d{2}-([a-z]{3}|\d{2})-\d{2} yang fungsinya sama.

Komentar dalam regex

Karakter meta (?#...) dapat dipakai untuk menyimpan komentar, yaitu jika Anda tidak menggunakan modifier x. Saran saya adalah menggunakan modifier x saja ketimbang sintaks ini.

Kode dalam regex

Karakter meta (?{ ... }), (??{ ... }), (?p{...}) hanya dikenal di Perl dan hanya relevan di Perl. Sintaks ini memungkinkan kita memasukkan kode Perl di dalam regex untuk penggunaan yang advanced. Mulai versi 4.0, PCRE juga mendukung sintaks ini.

Named capture

Menggunakan \1, \2, dsb untuk menangkap subpola regex kadang-kadang sulit diingat dan menyulitkan. Misalnya, ketika kita memodifikasi dan menambahkan sebuah match group, maka urutan nomor-nomor tersebut dapat berubah.

Kelas regex di .NET adalah yang termasuk pertama-tama memperkenalkan sintaks *named capture*. Bentuk sintaks ini:

```
(?<NAMA>POLA)
```

Sintaks (?<...>...) mirip dengan (...) tapi perbedaannya memberi nama pada pola berupa *NAMA*. Kita nanti dapat merujuk match group hasil tangkapan tidak dengan angka 1, 2, 3 lagi tapi dengan *NAMA1*, *NAMA2*, dan nama-nama lain yang disebutkan sesuka kita.

Perl hingga versi 5.8 belum mendukung named capture, tapi ada modul Perl bernama Regexp::Fields yang dapat digunakan untuk mendukung sintaks ini.

Atomic group dan possessive quantifier

Seperti mungkin telah dijelaskan sebelumnya, mesin regex akan mencoba "sekuat tenaga" dalam mencari kecocokan sebuah string dengan sebuah pola. Jika perlu mesin regex akan melakukan langkah mundur atau backtracking. Contohnya:

```
\w*bd
```

Jika pola di atas diterapkan pada sebuah string:

```
aaaabbaaaabcaaaabdaaaabeaaaaabf
```

maka mula-mula mesin regex akan menelan habis semua karakter (dikarenakan pola \w*). Namun ketika ingin mencocokkan b, karakter di string sudah habis sehingga mesin regex melakukan backtrack. Kini pola \w* hanya menelan semua karakter kecuali satu karakter terakhir. Tapi lagi-lagi, ketika ingin dicocokkan dengan b, yang ada hanyalah f. Maka backtrack lagi. Kali ini b cocok dengan b terakhir, tapi ternyata d tidak cocok dengan f. Backtrack lagi. Begitu seterusnya hingga belasan kali hingga ditemukan bd. Sama halnya jika kita memiliki pola:

```
\w*?bd
```

Bedanya, karena pola non-greedy maka pertama-tama \w* akan menelan 0 karakter. Ternyata huruf a pertama pada string tidak cocok dengan b. Maka dilakukan backtrack, \w* mengambil 1 karakter. Ternyata karakter kedua tidak cocok juga. Backtrack lagi. Begitu seterusnya hingga ditemukan bd.

Meskipun proses backtracking ini berjalan cepat, namun jika sebuah pola cukup kompleks (mis: mengandung beberapa quantifier * atau +) dan string yang ingin dicocokkan panjang, akan terjadi begitu banyak proses backtracking yang membuat pencocokan berjalan amat lambat—dalam hitungan menit bahkan jam! Program Anda akan terkesan seperti hang atau macet dan memakan CPU.

Untuk membantu mempercepat pencocokan kita dapat menyuruh mesin regex untuk menghindari backtracking. Sintaks yang dipakai untuk ini adalah (?>...) dan dinamai atomic group. Dinamai atomik karena begitu string diambil pertama kali oleh mesin regex, tidak lagi bisa backtracking. Contoh:

```
(?>\w*)bd
```


Pola ini tidak akan cocok dengan string di atas. Kenapa? Karena begitu \w* menelan semua string yang ada, ketika b tidak cocok, mesin regex tidak bisa backtrack lagi untuk mencoba kemungkinan lain. Agar pola dapat cocok, kita harus mengubahnya menjadi:

```
(?>a*)bd
```

Kedua pola di atas dapat juga ditulis dengan shortcut:

```
\w*+bd
a*+bd
```

POLA++, POLA*+, dan POLA?+ adalah bentuk pendek dari atomic group dan disebut dengan possessive quantifier. Possessive quantifier pertama-tama didukung oleh Java. Perl sendiri saat ini tidak mendukung sintaks shortcut ini; Anda hanya dapat menggunakan sintaks (?>...) saja.

Apakah atomic group berguna? Ya, terutama dalam mempercepat pencocokan (tepatnya, mempercepat *kegagalan* dengan menghindari backtracking yang sia-sia). Contohnya (contoh ini saya ambil dari tutorial di http://www.regular-expressions.info/):

```
^(.*?,){11}P
```

Pola ini hendak mencari apakah field ke-12 dimulai dengan huruf P. Setiap field pada string dipisahkan dengan tanda koma. Contoh:

```
1,2,3,4,5,6,7,8,9,10,11,12,13
```

Pada kasus di atas, kita dengan mudah dapat melihat bahwa field ke-12 bukanlah "P" melainkan "12". Tapi mesin regex akan melakukan banyak sekali backtracking sebelum mencapai pada kesimpulan ini. Ini dikarenakan .*? juga dapat mencocokkan karakter koma. Jadi setelah gagal pada tahap 1,2,3,4,5,6,7,8,9,10,11, mesin regex akan mencoba 1, untuk .*?.
Tapi karakter berikutnya, 2, tidak cocok dengan , pada pola sehingga mesin regex akan mencoba lagi 1,2. Begitu seterusnya hingga akhirnya mesin regex menyerah setelah semua kemungkinan dicoba.

Dengan pola:

```
^(?>(.*?,){11})P
```

maka mula-mula mesin regex akan menelan sebelas field 1,2,3,4,5,6,7,8,9,10,11,. Setelah melihat field ke-12 tidak cocok, mesin regex langsung selesai dan menghasilkan jawaban tidak cocok. Ini karena ia tidak bisa backtracking lagi sebab (?>(.*?,){11}) berarti atomic group.

Kesimpulan, atomic group terutama berperan untuk mempercepat proses.

Penutup Bab

Bab ini telah membahas belasan karakter meta yang ada. Bagi seorang pemula tentu ini sulit dikuasai semua pada awalnya. Cara terbaik untuk menguasai sintaks regex sedikit demi sedikit adalah melalui contoh dan latihan. Bab 4 adalah bab di mana Anda bisa melihat bagaimana berbagai karakter meta regex berperan dalam kasus nyata. Sementara itu, sebagai penutup bab, silakan coba Latihan 2 berikut ini.

Latihan 2

1. Apa pola untuk menyatakan "nama file yang mengandung string nama akhirannya"? Contohnya: txt-1.txt (akhiran txt dikandung juga oleh nama file), atau executeme.exe (akhiran exe dikandung dalam nama file).
2. Apa pola untuk mendeteksi keberadaan tiga buah kata yang sama dalam sebuah teks, di mana kata tersebut terdiri dari minimal 4 huruf?
3. Bagaimana pola untuk mencari palindrome yang terdiri dari 5 huruf? Palindrome adalah kata yang jika dieja terbalik hasilnya sama. Contohnya: bab, malam, kodok, tamat.

Jawaban Latihan 2

1. Salah satu jawaban: ^[^.]*([^.]+)[^.]*\.\1$. Bagian [^.]*([^.]+)[^.]* berarti kita ingin mencari sepotong string ([^.]+) sebelum tanda titik dan kita menginginkan potongan string ini ada sesudah titik (\.\1). Mesin regex akan berusaha melakukan berbagai cara agar kondisi ini terpenuhi. Contohnya, jika kita memiliki string "filetxt1.txt" mula-mula filetxt1 akan ditelan dulu oleh subpola [^.]* yang pertama. Tapi setelah mesin regex melihat bahwa filetxt1 tidak cocok dengan \.\1, mesin regex akan mundur dan mencoba filetxt, filetx, filet, hingga file. Selanjutnya, txt1 akan ditelan oleh match group pertama. Tapi karena terakhirnya tidak bisa cocok dengan backtrack \1, maka mesin regex akan memundurkan match group pertama menjadi txt. Sampai di sini 1 akan cocok dimakan subpola ketiga, dan backtrack akan cocok sehingga seluruh pola cocok. Tapi Anda tak perlu pusing-pusing memikirkannya, semua sudah dilakukan mesin regex. Yang perlu Anda lakukan hanyalah membuat pola yang sesuai lalu serahkan pada mesin regex.
2. \b(\w{4,})\b.+\1.+\1. Mengapa \b perlu di sini? Karena jika tidak, \w{4,} dapat saja dicocokkan dengan potongan string di dalam kata. Misalnya: teks "semata-mata karena matahari" akan cocok dengan pola regex (\w{4,}).+\1.+\1 di mana \1 adalah mata. Tapi sebetulnya tidak ada tiga kata yang sama di situ, karena kata mata hanya ada satu kali.
3. \b(\w)(\w)\w\2\1\b. Untuk mencari palindrome 7 huruf, \b(\w)(\w)(\w)\w\3\2\1\b. Lagi-lagi, tanpa \b pola pertama bisa saja cocok dengan kemalaman, padahal kata tersebut bukan palindrome.