Praktikum 3
Regular Expression (regex)

Tujuan Pembelajaran

Mahasiswa dapat memahami dan menggunakan regular ekspression dalam bahasa pemrograman awk.

Dasar Teori

Regular expression atau regex adalah cara untuk mendeskripsikan sebuah set dari string. Regex diapit oleh tanda slash (‘/’) dalam pola awk yang sesuai dengan setiap input record yang teksnya dimiliki oleh set tersebut. Regular expression merupakan fasilitas yang dapat digunakan untuk melakukan evaluasi suatu data string terhadap pola tertentu. Terdapat banyak pola yang didukung oleh regex, termasuk character class yang meliputi alphanumerik, digit, dan lain-lain.

Regular expression ditulis dalam *formal language* (bahasa formal). Regular expression dapat sangat bermanfaat dalam kompilasi terutama untuk proses pengolahan source code, untuk memilah-milah (scanning) sintak, token-token, analisis kesalahan dan sebagainya. Selain itu Regular expression juga sangat berguna untuk validasi string yang biasanya dipakai untuk input.

Regex yang paling sederhana adalah rangkaian beberapa huruf, angka, atau keduanya. Regex semacam itu akan cocok dengan sembarang string yang mengandung rangkaian (pola sederhana) tersebut. Misalnya regex ‘foo’ cocok dengan sembarang string yang mengandung ‘foo’ di record manapun. Jenis regex lain memungkinkan Anda untuk menentukan kelas string lain yang lebih kompleks.

Regex sudah banyak digunakan didalam pencocokkan pola, berikut ini beberapa aplikasi regex pada pencocokkan pola yang banyak digunakan untuk validasi beberapa string berikut ini.

- username dan password
- e-mail, URL dan HTML tag
- alamat IP
- nomor telepon

Percobaan 1: Bagaimana menggunakan *regex*

Sebuah regular expression dapat digunakan sebagai *pattern* (pola) dengan cara mengapit dengan slash (/…/). Kemudian regular expression tersebut diuji coba terhadap keseluruhan teks pada setiap record. (Secara normal, ini hanya perlu sebagian teks saja yang cocok agar berhasil.) Contoh berikut ini mencetak field ke-2 dari setiap record yang mengandung string 'foo' dimanapun. Berikut ini program yang menggunakan regex sederhana, yakni mencari satu set karakter 'foo' kemudian mencetak field ke-2 pada record yang cocok pada file BBS-list.

```
$ awk '/foo/ { print $2 }' BBS-list
= 555-1234
= 555-6699
= 555-6480
= 555-2127
```

Dengan perintah di atas, maka saat ditemukan baris yang mengandung 'foo' akan dicetak karena 'print $2' berarti mencetak seluruh baris pada kolom tersebut. Karakter slash ('/') menandakan bahwa 'foo' adalah pola yang dicari. Tipe pola tersebut merupakan *regular expression*. Pola yang digunakan ini membolehkan kesesuaian untuk sebagian kata. Terdapat single quote (tanda petik) yang melingkupi program awk sehingga shell tidak akan mengenalinya sebagai karakter shell khusus.

Reguler ekspresion juga dapat digunakan dalam pencocokan ekspresi. Program berikut akan memakai operator '~' (tilde). Operator tersebut digunakan pada regular expression sebagai ekspresi pencocokan. Ekspresi tersebut memungkinkan untuk mencari string mana yang cocok. Operator '~' dan '~!' melakukan perbandingan terhadap regular expression. Operator '~' digunakan untuk mencari ekspresi yang sama dengan sintaks sebagai berikut:

```
exp ~ /regexp/
```

Sedangkan operator '~!' digunakan untuk mencari ekspresi yang tidak sama dengan sintaks sebagai berikut.

```
exp !~ /regexp/
```

Pada percobaan yang menggunakan ekspresi tersebut dapat ditambahkan blok **if**. Selain **if**, ekspresi tersebut juga dapat diterapkan pada **while**, **for**, dan **do**. Penggunaan **if** tersebut menghasilkan output yang sama dengan tanpa menggunakan **if**.

Program berikut ini program untuk menampilkan semua input record yang diawali dengan karakter 'J'.

```
$ awk '$1 ~ /J/' inventory-shipped
= Jan 13 25 15 115
= Jun 31 42 75 492
= Jul 24 34 67 436
= Jan 21 36 64 620
```

Atau perintah berikut ini:

```
$ awk '{ if ($1 ~ /J/) print }' inventory-shipped
= Jan 13 25 15 115
= Jun 31 42 75 492
= Jul 24 34 67 436
= Jan 21 36 64 620
```

Berikut ini program untuk menampilkan semua input record yang diawali selain karakter 'J'.

```
$ awk '$1 !~ /J/' inventory-shipped
= Feb 15 32 24 226
= Mar 15 24 34 228
= Apr 31 52 63 420
= May 16 34 29 208
...
```

Regex tertututup dengan slash seperti /foo/ pada contoh diatas, kita sebut sebagai konstanta regex, seperti sebuah konstanta numerik and "foo" adalah konstanta string.

Percobaan 2: Escape sequence

Beberapa karakter tidak dapat disertakan secara literal dalam string yang berisi ("foo") atau regex constants (/foo/). Melainkan sebaiknya direpresentasikan dengan *escape sequence*, yaitu rangkaian karakter yang diawali dengan backslash ('\'). Satu fungsi *escape sequence* adalah untuk menyertakan sebuah karakter *double-quote* (") dalam string constant. Karena double quote biasa mengakhiri string, maka harus menggunakan ' \" ' untuk merepresentasikan sebuah karakter *double-quote* yang aktual sebagai bagian dari string. Contoh seperti pada program dibawah ini.

```
$ awk 'BEGIN { print "He said \"hi!\" to her." }'
= He said "hi!" to her.
```

Tanda *escape* tersebut menunjukkan bahwa panda petik hendak dicetak sebagai karakter biasa. Jika tanpa tanda tanda escape maka akan dikenali sebagai karakter khusus, yang mungkin saja pada tempat yang salah akan dapat

mengakibatkan error. Berikut ini adalah daftar tabel semua escape sequences yang digunakan di awk dan merepresentasikan apa (kecuali dinyatakan sebaliknya, semua escape sequence berlaku untuk konstanta string dan konstanta regex).

Tabel. Escape Sequences yang digunakan di awk

Escape sequence	Information
\\	*Literal backslash, '\'.*
\a	*"Alert" character, Ctrl-g, ASCII code 7 (BEL). (This usually makes some sort of audible noise.)*
\b	*Backspace, Ctrl-h, ASCII code 8 (BS).*
\f	*Formfeed, Ctrl-l, ASCII code 12 (FF).*
\n	*Newline, Ctrl-j, ASCII code 10 (LF).*
\r	*Carriage return, Ctrl-m, ASCII code 13 (CR).*
\t	*Horizontal TAB, Ctrl-i, ASCII code 9 (HT).*
\v	*Vertical tab, Ctrl-k, ASCII code 11 (VT).*
\nnn	*The octal value "nnn", where nnn stands for 1 to 3 digits between '0' and '7'. For example, the code for the ASCII ESC (escape) character is '\033'.*
\xhh...	*The hexadecimal value "hh", where "hh" stands for a sequence of hexadecimal digits ('0'–'9', and either 'A'–'F' or 'a'–'f'). Like the same construct in ISO C, the escape sequence continues until the first nonhexadecimal digit is seen. (c.e.) However, using more than two hexadecimal digits produces undefined results. (The '\x' escape sequence is not allowed in POSIX awk.)*
\/	*A literal slash (necessary for regexp constants only). This sequence is used when you want to write a regexp constant that contains a slash. Because the regexp is delimited by slashes, you need to escape the slash that is part of the pattern, in order to tell awk to keep processing the rest of the regexp.*
\"	*A literal double quote (necessary for string constants only). This sequence is used when you want to write a string constant that contains a double quote. Because the string is delimited by double quotes, you need to escape the quote that is part of the string, in order to tell awk to keep processing the rest of the string.*

Cobalah pergunakan masing-masing escape sequence di atas dalam program berikut ini:

```
$ awk 'BEGIN { print "Is this character \\ ?"}'
$ awk 'BEGIN { print "Is this character \a ?"}'
$ awk 'BEGIN { print "Is this character \b ?"}'
$ awk 'BEGIN { print "Is this character \f ?"}'
$ awk 'BEGIN { print "Is this character \n ?"}'
$ awk 'BEGIN { print "Is this character \r ?"}'
$ awk 'BEGIN { print "Is this character \t ?"}'
$ awk 'BEGIN { print "Is this character \v ?"}'
```

Tanda escape juga digunakan untuk menampilkan karakter dengan memberikan kode karakter oktal maupun heksa desimal. Untuk karakter heksa escapenya

adalah “\nnn” (nnn.. adalah kode heksa desimal). Contohnya seperti pada program berikut ini:

```
$ awk 'BEGIN { print "Is this character \052 ?"}'
```

Untuk karakter heksa desimal escapenya adalah “\xhh...” (hh... adalah kode heksa desimal). Contohnya seperti pada program berikut ini:

```
$ awk 'BEGIN { print "Is this character \x65 ?"}'
```

Kode 052 dan 2a pada contoh diatas adalah kode oktal dan heksa desimal untuk karakter asterik (*). Berikut ini adalah tabel dari nilai dari karakter ASCII-128.

Tabel nilai dari karakter ASCII-128

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Escape sequence \/ digunakan untuk konstanta regex (*regex constants*) dan \” dugunakan untuk konstanta string (*string constants*). Cobalah program dibawah ini.

```
$ awk 'BEGIN { print "Is this character \/ ?"}'
$ awk 'BEGIN { print "Is this character \" ?"}'
```

Untuk karakter backslash (\) terdapat warning yang menyatakan urutan escape tersebut dikenali sebagai karakter slash ‘/’ biasa._Lebih lanjut apa itu konstanta regex dan konstanta string akan dibahas pada akhir bab ini.

Percobaan 3: Regular Expression Operator

Regular Expression Operator pada awk memungkinkan untuk mengkombinasikan regular expression dengan karakter khusus, yang disebut *regular expression operator* atau *metacharacters*, untuk meningkatkan kemampuan dan flesibilitas dari regular expression tersebut. Escape sequences yang dijelaskan sebelumnya dimulai oleh karakter '\' dan dikenali dan diubah sesuai dengan karakter yang sebenarnya sebagai langkah pertama dalam pemprosesan regex. Berikut ini adalah tabel dari metakarakter (semua karakter yang tidak bukan escape sequences dan yang tidak tercantum dalam tabel adalah karakter yang berdiri sendiri):

Tabel. Metakarakter yang digunakan di awk

Metacharacter	Information
`\`	*This is used to suppress the special meaning of a character when matching. For example, '\$' matches the character '$'.*
`^`	*This matches the beginning of a string. For example, '^@chapter' matches '@chapter' at the beginning of a string and can be used to identify chapter beginnings in Texinfo source files. The '^' is known as an anchor, because it anchors the pattern to match only at the beginning of the string. It is important to realize that '^' does not match the beginning of a line embedded in a string. The condition is not true in the following example:* *if ("line1\nLINE 2" ~ /^L/) ...*
`$`	*This is similar to '^', but it matches only at the end of a string. For example, 'p$' matches a record that ends with a 'p'. The '$' is an anchor and does not match the end of a line embedded in a string. The condition in the following example is not true:* *if ("line1\nLINE 2" ~ /1$/) ...*
`. (period)`	*This matches any single character, including the newline character. For example, '.P' matches any single character followed by a 'P' in a string. Using concatenation, we can make a regular expression such as 'U.A', which matches any three-character sequence that begins with 'U' and ends with 'A'. In strict POSIX mode (see Section 2.2 [Command-Line Options], page 25), '.' does not match the nul character, which is a character with all bits equal to zero. Otherwise, nul is just another character. Other versions of awk may not be able to match the nul character.*
`[...]`	*This is called a bracket expression. It matches any one of the characters that are enclosed in the square brackets. For example, '[MVX]' matches any one of the characters 'M', 'V', or 'X' in a string. A full discussion of what can be inside the square brackets of a bracket expression is given in Section 3.4 [UsingBracket Expressions], page 42.*
`[^ ...]`	*This is a complemented bracket expression. The first character after the '[' must be a '^'. It matches any characters except those in the square brackets. For example, '[^awk]' matches any character that is not an 'a', 'w', or 'k'.*
`|`	*This is the alternation operator and it is used to specify alternatives. The '|' has the lowest precedence of all the regular expression operators. For example, '^P|[[:digit:]]' matches any string that matches either '^P' or '[[:digit:]]'. This means it matches any string that starts with 'P' or contains a digit. The alternation applies to the largest possible regexps on either side.*
`(...)`	*Parentheses are used for grouping in regular expressions, as in arithmetic.*

	They can be used to concatenate regular expressions containing the alternation operator, '|'. For example, '@(samp|code)\{[^}]+\}' matches both '@code{foo}' and '@samp{bar}'. (These are Texinfo formatting control sequences. The '+' is explained further on in this list.)
*	*This symbol means that the preceding regular expression should be repeated as many times as necessary to find a match. For example, 'ph*' applies the '*' symbol to the preceding 'h' and looks for matches of one 'p' followed by any number of 'h's. This also matches just 'p' if no 'h's are present. The '*' repeats the smallest possible preceding expression. (Use parentheses if you want to repeat a larger expression.) It finds as many repetitions as possible. For example, 'awk '/\(c[ad][ad]*r x\)/ { print }' sample' prints every record in sample containing a string of the form '(car x)', '(cdr x)', '(cadr x)', and so on.*
+	*Notice the escaping of the parentheses by preceding them with backslashes. This symbol is similar to '*', except that the preceding expression must be matched at least once. This means that 'wh+y' would match 'why' and 'whhy', but not 'wy', whereas 'wh*y' would match all three of these strings. The following is a simpler way of writing the last '*' example:* `awk '/\(c[ad]+r x\)/ { print }' sample`
?	*This symbol is similar to '*', except that the preceding expression can be matched either once or not at all. For example, 'fe?d' matches 'fed' and 'fd', but nothing else.*
{n} {n,} {n,m}	*One or two numbers inside braces denote an interval expression. If there is one number in the braces, the preceding regexp is repeated n times. If there are two numbers separated by a comma, the preceding regexp is repeated n to m times. If there is one number followed by a comma, then the preceding regexp is repeated at least n times:* `wh{3}y` *matches 'whhhy', but not 'why' or 'whhhhy'.* `wh{3,5}y` *matches 'whhhy', 'whhhhy', or 'whhhhhy', only.* `wh{2,}y` *matches 'whhy' or 'whhhy', and so on.* *Interval expressions were not traditionally available in awk. They were added as part of the POSIX standard to make awk and egrep consistent with each other. Initially, because old programs may use '{' and '}' in regexp constants, gawk did not match interval expressions in regexps. However, beginning with version 4.0, gawk does match interval expressions by default. This is because compatibility with POSIX has become more important to most gawk users than compatibility with old programs. For programs that use '{' and '}' in regexp constants, it is good practice to always escape them with a backslash. Then the regexp constants are valid and work the way you want them to, using any version of awk. Finally, when '{' and '}' appear in regexp constants in a way that cannot be interpreted as an interval expression (such as /q{a}/), then they stand for themselves.*

Pada percobaan ini pengujian regex akan mengambil input dari keyboard dengan memanfaatkan utilitas **cat**. Kemudian dengan kontrol **if-else** untuk menentukan cocok atau tidak. Tanda escape digunakan bila menginginkan pengujian regex yang mengandung karakter khusus.

Pada percobaan berikut ini karakter dolar ‘$’ merupakan karakter khusus sehingga harus menggunakan metakarakter escape.

```
$ cat | awk '{ if($0 ~ /\$/) print "# match"; else
print “# not match” }’
> Someone
> $1000
```

Jika tidak menggunakan karakter escape, maka karakter dolar ‘$’ dikenali sebagai operator. Seperti contoh di di bawah ini

```
$ cat | awk '{ if($0 ~ /$/) print "# match"; else print
“# not match” }’
> Someone
> $1000
```

Operator ‘^’ akan cocok untuk awal dari string. Untuk regex ‘^Ro’ hanya akan cocok bila string tersebut diawali dengan ‘Ro’. ‘Ro’ di tengah maupun akhir tidak akan cocok.

```
$ cat | awk '{ if($0 ~ /^Ro/) print "# match"; else
print “# not match” }’
> Ronaldo
> romaRio
```

Operator ‘$’ akan cocok untuk akhir dari string. Untuk regex ‘o$’ hanya akan cocok bila string tersebut diakhirir dengan ‘o’. ‘o’ di tengah maupun awal tidak akan cocok.

```
$ cat | awk '{ if($0 ~ /o$/) print "# match"; else
print “# not match” }’
> sukarno
> suharto
> megawati
```

Operator ‘.’ akan cocok untuk setiap karakter tunggal termasuk juga karakter newline. Untuk regex ‘.p’ akan cocok bila sebelum karakter ‘p’ terdapat satu karakter. Jika sebelum karakter ‘p’ tidak ada karakter maka tidak cocok. Sedangkan untuk regex ‘t.p’ maka akan cocok bila antara arakter ‘t’ dan ‘p’ terdapat tepat satu karakter.

```
$ cat | awk '{ if($0 ~ /.p/) print "# match"; else
print “# not match” }’
> cap
> pet
```

```
$ cat | awk '{ if($0 ~ /t.p/) print "# match"; else
print "# not match" }'
> top
> trap
```

Operator […] merupakan operator untuk *character list* (daftar karakter). Sejumlah karakter dapat dimasukkan dalam operator (kurung siku) tersebut. Regex tersebut akan cocok apabila terdapat salah satu dari karakter yang ada dalam oprator tersebut. Untuk regex '[aWk]' akan cocok apa bila ditemukan salah satu dari 'a', 'W', atau 'k'. Regex secara default bersifat case sensitive, sehingga dibedakan antara 'W' dan 'w'.

```
$ cat | awk '{ if($0 ~ /[aWk]/) print "# match"; else
print "# not match" }'
> swap
> swat
> kick
> rocK
```

Operator '^' yang ada di dalam operator […] berarti komplemen, yaitu regex tersebut akan cocok bila tidak ditemukan salah satu karakter dari *character list* yang ada di dalam operator tersebut. Program berikut ini jika memukan salah satu dari a/i/u/e/o maka akan menampilkan tidak cocok.

```
$ cat | awk '{ if($0 ~ /[^aiueo]/) print "# match";
else print "# not match" }'
> a
> i
> r
```

Operator '|' adalah alternation operator yang digunakan untuk beberapa alternatif yang diberikan. Operator tersebut seperti operator logika OR. Sehingga akan memenuhi bila salah satu karakter cocok dengan pola yang diberikan.

```
$ cat | awk '{ if($0 ~ /[A|a]/) print "# match"; else
print "# not match" }'
> Anom
> anom
```

Operator tanda kurung "(…)" pada regex berfungsi sebagaimana tanda kurung pada kalimat matematika, yakni untuk mengelompokkan notasi/regular ekspresi. Program berikut ini akan menunjukkan pilihan dari 'red' atau 'blue' yang dikelompokkan dengan tanda kurung. Sehingga dapat diketahui bahwa regex dalam tanda kurung tersebut merupakan alternatif pola yang dapat dipilih salah satu. Untuk regex '(red|blue)hat' akan cocok dengan 'redhat' atau 'bluehat'.

```
$ cat | awk '{ if($0 ~ /(red|blue)hat/) print "#
match"; else print "# not match" }'
> redhat
> bluehat
> blackhat
```

Operator '*' berarti bahwa regex sebelum tanda ini bisa diulang sebanyak apapun agar cocok. Jumlahnya tidak ada ketentuan, bahkan tetap cocok bila tidak ada.

```
$ cat | awk '{ if($0 ~ /hanso*me/) print "# match";
else print "# not match" }'
> hansome
> hansoome
> hansooooooooooooome
> hansme
```

Operator '+' mirip dengan operator '*' hanya saja pengulangannya minimal satu. Sehingga apabila karakter tersebut tidak ada, maka tidak cocok.

```
$ cat | awk '{ if($0 ~ /hanso+me/) print "# match";
else print "# not match" }'
> hansome
> hansoome
> hansooooooooooooome
> hansme
```

Operator '?' mirip dengan operator '*' hanya saja pengulangannya maksimal satu kali. Sehingga hanya cocok bila ada satu kali atau tidak ada, dan tidak cocok bila terdapat pengulangan lebih dari satu kali.

```
$ cat | awk '{ if($0 ~ /hanso?me/) print "# match";
else print "# not match" }'
> hansome
> hansoome
> hansooooooooooooome
> hansme
```

Percobaan 4: Menggunakan "bracket expression"

Bracket expression (ekspresi dengan kurung siku) mencocokkan setiap karakter sesuai dengan yang ditulis di antara kurung siku buka dan kurung siku tutup. Di antara kurung siku ada *range expression* terdiri dari dua karakter yang dipisahkan dengan *hyphen* (-). Ini akan mencocokkan semua *single character* yang berada diantara dua karakter yang didefinisikan pada *range expression*. Contoh:

`'[0-9]'` sama dengan `'[0123456789]'`

Untuk memasukkan karakter '\', ']', '-', atau '^' pada bracket expression, letakkan '\' di depannya. Contoh:

`'[d\]]'` akan mencocokkan karakter `'d'` atau `']'`

Character classes (kelas karakter) adalah fitur yang ada pada standar POSIX. Sebuah *character class* adalah notasi yang khusus untuk menggambarkan sekumpulan karakter yang memiliki atribut yang spesifik. Kadang-kadang karakter tersebut dapat berbeda di setiap negara. Sebagai contoh notasi (penulisan) *alphabetic character* berbeda antara Amerika Serikat dan Perancis. Kelas karakter hanya valid digunakan pada regex didalam kurung siku pada *bracket expression* yang terdiri dari:

`'[: keyword character class :]'`

Table berikut ini menuliskan sejumlah *character classes* yang standar POSIX.

Tabel: POSIX Character Classes

Class	Meaning
`[:alnum:]`	*Alphanumeric characters.*
`[:alpha:]`	*Alphabetic characters.*
`[:blank:]`	*Space and TAB characters.*
`[:cntrl:]`	*Control characters.*
`[:digit:]`	*Numeric characters.*
`[:graph:]`	*Characters that are both printable and visible. (A space is printable but not visible, whereas an 'a' is both.)*
`[:lower:]`	*Lowercase alphabetic characters.*
`[:print:]`	*Printable characters (characters that are not control characters).*
`[:punct:]`	*Punctuation characters (characters that are not letters, digits, control characters, or space characters).*
`[:space:]`	*Space characters (such as space, TAB, and formfeed, to name a few).*
`[:upper:]`	*Uppercase alphabetic characters.*
`[:xdigit:]`	*Characters that are hexadecimal digits*

Contohnya, sebelum ada standard POSIX, Anda harus menuliskan /[A-Za-z0-9]/ untuk mencocokkan karakter alphanumeric. Jika karakter yang Anda tuliskan memiliki karakter lain, tidak akan dapat dicocokkan. Jika menggunakan kelas karakter POSIX, Anda bisa menuliskan denan /[[:alnum:]]/ untuk mencocokkan karakter *alphabetic* dan *numeric* pada karakter yang Anda masukkan.

Class [:alnum:] cocok untuk karakter alphanumerik, meliputi huruf dan angka, yaitu huruf A-Z atau a-z dan angka 0-9. Tidak termasuk symbol atau karakter khusus.

```
$ cat | awk '{ if($0 ~ /^[[:alnum:]]*$/) print "# match"; else print "# not match" }'
> abcdefghijklmnopqrstuvwxyz
> 0123456789
> ~!@#$%^&*()
```

Class [:alpha:] cocok untuk karakter huruf, yaitu a-z atau A-Z. Tidak termasuk angka, symbol atau karakter khusus, maupun karakter lain seperti tab, spasi, dll.

```
$ cat | awk '{ if($0 ~ /^[[:alpha:]]*$/) print "# match"; else print "# not match" }'
> abcdefghijklmnopqrstuvwxyz
> 0123456789
> ~!@#$%^&*()
```

Class [:blank:] cocok untuk karakter karakter TAB dan karakter spasi.

```
$ cat | awk '{ if($0 ~ /[[:blank:]]/) print "# match"; else print "# not match" }'
> ini '' adalah karakter SPACE
> ini '	' adalah karakter TAB
> kalimatinitidakmenggunakanSPACEatauTAB
```

Class [:cntrl:] cocok untuk karakter control. Karakter control adalah karakter yang tidak merepresentasikan simbol tetapi merepresentasikan *character encoding*.

```
$ cat | awk '{ if($0 ~ /[[:cntrl:]]/) print "# match"; else print "# not match" }'
>  (tekan tombol CTRL+A)
>  (tekan tombol DEL)
>  (tekan tombol ENTER)
```

Karakter control antara lain `NL CR LF TAB VT FF NUL SOH STX EXT EOT ENQ ACK SO SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC IS1 IS2 IS3 IS4` dan `DEL`. Berikut ini tabel yang menunjukkan beberapa karakter kontrol ASCII:

Tabel: Karakter Kontrol ASCII

Desimals	Character	Information
0	(null, NUL, \0, ^@)	*originally intended to be an ignored character, but now used by many programming languages to mark the end of a string.*
7	(bell, BEL, \a, ^G)	*which may cause the device receiving it to emit a warning of some kind (usually audible).*
8	(backspace, BS, \b, ^H)	*used either to erase the last character printed or to*

		overprint it.
9	(horizontal tab, HT, \t, ^I),	*moves the printing position some spaces to the right.*
10	(line feed, LF, \n, ^J)	*used as the end of line marker in most UNIX systems and variants.*
12	(form feed, FF, \f, ^L)	*to cause a printer to eject paper to the top of the next page, or a video terminal to clear the screen.*
13	(carriage return, CR, \r, ^M)	*used as the end of line marker in Mac OS, OS-9, FLEX (and variants). A carriage return/line feed pair is used by CP/M-80 and its derivatives including DOS and Windows, and by Application Layer protocols such as HTTP.*
27	(escape, ESC, \e [GCC only], ^[).	*escape character.*
127	(delete, DEL, ^?)	*originally intended to be an ignored character, but now used in some systems to erase a character. Also used by some Plan9 console programs to send an interrupt note to the current process.*

Class [:digit:] cocok untuk karakter numerik (angka) yaitu 0-9.

```
$ cat | awk '{ if($0 ~ /[[:digit:]]/) print "# match"; 
else print "# not match" }'
> abcde
> 0123456789
> ~!@#$%^&*()
```

Class [:graph:] cocok untuk karakter yang dapat dicetak dan tampak. Contoh karakter yang dapat dicetak namun tidak tampak adalah karakter SPACE dan TAB. Jadi karakter graph adalah karakter yang tidak mengandung SPACE dan TAB.

```
$ cat | awk '{ if($0 ~ /^[[:graph:]]*$/) print "# 
match"; else print "# not match" }'
> ada SPACE
> ada     TAB
> t1d4k4dAk4r4ct3rSPACE/TAB
```

Class [:lower:] cocok untuk karakter lowercase (huruf kecil).

```
$ cat | awk '{ if($0 ~ /^[[:lower:]]*$/) print "# 
match"; else print "# not match" }'
> lowercase
> UPPERCASE
> AnAkAlAy
```

Class [:print:] cocok untuk karakter yang dapat dicetak (bukan karakter control) termasuk karakter SPACE tetapi tidak untuk TAB.

```
$ cat | awk '{ if($0 ~ /^[[:print:]]*$/) print "# match"; else print "# not match" }'
> ada SPACE
> ada     TAB
> t1d4k4dAk4r4ct3rSPACE/TAB
> (tekan CTRL+A)
```

Class [:space:] cocok untuk karakter spasi termasuk TAB dan SPASI.

```
$ cat | awk '{ if($0 ~ /[[:space:]]/) print "# match"; else print "# not match" }'
> ada SPACE
> ada     TAB
> t1d4k4dAk4r4ct3rSPACE/TAB
> (tekan CTRL+A)
```

Class [:upper:] cocok untuk karakter huruf besar (huruf kapital).

```
$ cat | awk '{ if($0 ~ /^[[:upper:]]*$/) print "# match"; else print "# not match" }'
> lowercase
> UPPERCASE
> AnAkAlAy
```

Class [:xdigit:] cocok untuk karakter yang merepresentasikan hexadesimal yaitu 0-9, dan a-f atau A-F.

```
$ cat | awk '{ if($0 ~ /^[[:xdigit:]]*$/) print "# match"; else print "# not match" }'
> 0123456789
> ABCDEF
> abcdef
> GHIJKL
```

Percobaan 5: Operator regular expression pada gawk

Software GNU yang berurusan regular expression mendukung beberapa operator regex tambahan. Operator berikut ini tidak tersedia pada implementasi awk yang lain. Kebanyakan operator tambahan berurusan dengan pencocokan *word* (kata), dimana *word* adalah serangkaian satu atau lebih huruf, angka, atau underscore ('_'). Berikut ini adalah tabel operator regular expression pada gawk.

Operators	Informations
\s	*Matches any whitespace character. Think of it as shorthand for [[:space:]].*
\S	*Matches any character that is not whitespace. Think of it as shorthand for [^[:space:]].*
\w	*Matches any word-constituent character—that is, it matches any letter, digit, or underscore. Think of it as shorthand for [[:alnum:]_].*
\W	*Matches any character that is not word-constituent. Think of it as shorthand for [^[:alnum:]_].*
\<	*Matches the empty string at the beginning of a word. For example, /\<away/ matches 'away' but not 'stowaway'.*
\>	*Matches the empty string at the end of a word. For example, /stow\>/ matches 'stow' but not 'stowaway'.*
\y	*Matches the empty string at either the beginning or the end of a word (i.e., the word boundary). For example, '\yballs?\y' matches either 'ball' or 'balls', as a separate word.*
\B	*Matches the empty string that occurs between two word-constituent characters. For example, /\Brat\B/ matches 'crate' but it does not match 'dirty rat'. '\B' is essentially the opposite of '\y'.*

Operator \w cocok dengan sembarang karakter yang merupakan unsur kata, meliputi karakter alphanumerik dan karakter underscore ('_').

```
$ cat | awk '{ if($0 ~ /^\w*$/) print "# match"; else
print "# not match" }'
> abcdefghijklmnopqrstuvwxyz
> ABCDEFGHIJKLMNOPQRSTUVWXYZ
> _0123456789
> ~!@#$%^&*()
```

Operator \W merupakan komplemen dari \w. \W cocok dengan sembarang karakter yang bukan merupakan unsur kata .

```
$ cat | awk '{ if($0 ~ /^\W*$/) print "# match"; else
print "# not match" }'
> abcdefghijklmnopqrstuvwxyz
> ABCDEFGHIJKLMNOPQRSTUVWXYZ
> _0123456789
> ~!@#$%^&*()
```

Operator \< cocok untuk string kosong pada awal dari suatu kata. Regex '\<hat' cocok bila sebelum 'hat' adalah karakter kosong (SPACE atau TAB).

```
$ cat | awk '{ if($0 ~ /\<hat/) print "# match"; else
print "# not match" }'
> hate
> chatting
> I (SPACE) hate you
> I (TAB) hate you
```

Operator \> cocok untuk string kosong pada akhir dari suatu kata. Regex ‘hat\>’ cocok bila setelah ‘hat’ adalah karakter kosong.

```
$ cat | awk '{ if($0 ~ /hat\>/) print "# match"; else
print "# not match" }'
> haters
> chat
> I hate (SPACE) you
> I hate (TAB) you
```

Operator \y cocok untuk string kosong pada awal dan akhir dari suatu kata. Regex ‘\ylov?\y’ cocok bila sebelum dan setelah‘cap’ adalah karakter kosong.

```
$ cat | awk '{ if($0 ~ /\ylove?\y/) print "# match";
else print "# not match" }'
> love
> my lovely
> I am (SPACE) love (SPACE) you
> I am (TAB) love (TAB) you
```

Operator \B merupakan komplemen dari \y. \B cocok bila terdapat karakter kosong di antara dua karakter yang termasuk dalam karakter unsur kata.

```
$ cat | awk '{ if($0 ~ /\ymaaf?\y/) print "# match";
else print "# not match" }'
> maaf
> pemaaf
> maafkan
> memaafkan
```

Percobaan 6: Case sensitive

Case secara normal berpengaruh signifikan pada regular expression, baik kesesuaian dengan karakter biasa maupun dalam kumpulan karakter. Sehingga, sebuah ekspresi ‘w’ dalam sebuah regular expression hanya cocok dengan karakter ‘w’ (lowercase) dan tidak dengan sebuah ‘W’ (uppercase). Cara sederhana untuk memberikan pencocokan yang tidak bergantung pada case adalah dengan menggunakan *character list*, contohnya ‘*Ww+’. Namun, hal tersebut jadi tidak praktis untuk lebih dari satu karakter, selain itu akan membuat regular expression lebih sulit dibaca. Terdapat dua alternatif yang lebih disarankan.

Cara pertama untuk memungkinkan pencocokan yang case-sensitive pada beberapa poin penting dalam program adalah mengkonversi data kedalam satu single case, menggunakan fungsi `tolower` atau `toupper`. Contoh:

```
tolower($1) ~ /foo/ { ... }
```

mengubah field pertama ke lowercase sebelum dicocokkan dengan pola.

```
$ cat | awk '{ if(tolower($0) ~ /ari/) print "# match";
else print "# not match" }'
> Ari Sihasale
> Leska Tariani
> ANOM BESARI
```

Cara lainnya adalah, khusus pada gawk, adalah untuk mengatur nilai variable IGNORANCE menjadi nilai bukan nol. Seperti pada program berikut ini:

```
$ cat | awk 'BEGIN { IGNORECASE = 1} {if($0 ~ /ari/)
print "# match"; else print "# not match" }'
> Ari Sihasale
> Leska Tariani
> ANOM BESARI
```

Ketika IGNORANCE tidak bernilai nol, semua regex dan operasi string mengabaikan case. Mengubah nilai IGNORANCE akan secara signifikan mengubah kontrol case-sensitive pada program yang dijalankan. Secara default, IGNORANCE bernilai nol. Seperti pada program berikut ini:

```
$ awk 'BEGIN { x = "aB" ; if (x ~ /ab/) print "#
match"; else print "# not match" }'

$ awk 'BEGIN { IGNORECASE = 1; x = "aB" ; if (x ~ /ab/)
print "# match"; else print "# not match" }'
```

IGNORECASE dapat diset pada command line atau pada aturan BEGIN. Pengaturan IGNORECASE dari command line adalah cara yang tepat untuk membuat program case-insensitive tanpa harus melakukan merubahnya lagi.

Secara umum, Anda tidak dapat menggunakan IGNORECASE untuk membuat aturan case-insensitive dan yang lainnya case-sensitive. Untuk melakukan ini, penggunaan tolower() dan toupper() dapat secara dinamis merubah case-sensitif aktif atau tidak. Regex dan operasi string comparison (pembandingan string) dipengaruhi oleh IGNORECASE.

Dalam locales multibyte, para ekuivalensi antara karakter huruf besar dan huruf kecil diuji berdasarkan lebar karakter nilai-nilai set karakter lokal itu. Jika tidak, karakter yang diuji berdasarkan ISO-8859-1 (ISO Latin-1) set karakter. Ini set karakter adalah superset dari karakter ASCII-128, yang juga menyediakan sejumlah karakter yang cocok untuk digunakan dalam berbagai bahasa.

Percobaan 7: Sebanyak apakah teks itu dicocokkan?

Coba masukkan perintah berikut ini pada command line:

```
$ echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
= <A>bcd
```

Contoh tadi menggunakan fungsi "sub" (akan kita bahas lebih lanjut) yang akan digunakan untuk membuat perubahan/penggantian pada record input. Di sini, regex /+a/ menunjukkan satu atau lebih karakter 'a' dan teks penggantinya adalah '<A>'. Input berisi karakter 'aaaa'. Regex pada awk selalu memulai pencocokan sesuai dengan urutan dari mulai yang paling kiri, karakter terpanjang masukan yang bisa menandingi. Dengan demikian, karakter 'aaaa' diganti dengan '<A>'.

Untuk pengujian program pencocokan/pen-tidak-cocokan yang sederhana sebetulnya adalah hal tidak begitu penting. Tetapi jika diiringi dengan melakukan penggantian/penyesuaian/pembetulan terhadap teks, itu menjadi sangat penting. Substitusi dapat dilakukan dengan fungsi seperti `match()`, `sub()`, `gsub()`, dan `gensub()` yang nanti akan kita bahas di bab mendatang. Karena memahami prinsip ini juga penting untuk regex yang digunakan untuk RS *(record separator*) dan FS (*field separator*).

Percobaan 8: Menggunakan regex yang dinamis

Kemampuan regex yang dinamis adalah setelah operator '~' atau '!~' tidak memerlukan konstanta regex (seperti string atau karakter yang diapit slash). Ekspresi tersebut dievaluasi dan dikonversi kedalam string jika diperlukan, yang kemudian dapat digunakan sebagai regex. Regex yang seperti ini disebut dengan regex dinamis.

Program dibawah ini menginisialisasi variabel "digits_regex" menjadi regex yang menjelaskan satu atau lebih digit, dan coba apakah apakah input record cocok dengan regex tersebut.

```
$ awk 'BEGIN { digits_regex = "[[:digit:]]+" } if ($0 ~
digits_regex) print "# match"; else print "# not match"
}'
> 12345
> abcde
> 123ABC
> ABC123
```

Ketika menggunakan operator '~' dan '~!', ada perbedaan antara konstanta regex yang ditutup garis miring (/…/) dan konstanta string yang tertutup dengan *double*

quote (kutipan ganda). Jika Anda akan menggunakan konstanta string, Anda harus memahami bahwa string pada dasarnya dilakukan *scanning* (pembacaan) dua kali. Yang pertama ketika awk membaca program Anda, dan yang kedua kali ketika awk mencocokkan string sebelah kiri operator dengan pola di sebelah kanan operator. Hal ini berlaku dari setiap ekspresi seperti contoh “digits_regex” tidak hanya konstanta string.

Apa yang membedakan jika ada sebuah string dibaca (*scan*) dua kali? Jawabannya harus dilakukan dengan escape sequence, dan terutama dengan backslashes (\). Contohnya untuk mendapatkan bentuk regex dari backslash dalam string, anda harus mengetikkan dua backslashes (\\).

Sebagai contoh, /*/ adalah konstanta regex untuk karakter literal '*'. Hanya satu backslash diperlukan. Untuk melakukan hal yang sama dengan string, Anda harus mengetikkan "*". Backslash pertama meng-escape yang kedua, sehingga string benar-benar berisi dua karakter '\' dan '*'.

Mengingat bahwa Anda dapat menggunakan kedua bentuk “konstanta regex” dan “konstanta string” untuk menggambarkan regular expression. Pertanyaannya yang mana yang harus Anda gunakan? Jika jawabannya adalah "konstanta regex" berikut ini beberapa alasannya:

1) Konstanta string lebih rumit untuk ditulis dan lebih sulit untuk dibaca. Sedangkan menggunakan konstanta regex membuat program Anda lebih aman dari kesalahan. Jika Anda tidak memahami perbedaan antara dua jenis konstanta ini dapat menyebabkan kesalahan.
2) Konstanta regex lebih efisien, karena awk dapat mencatat bahwa Anda telah menggunakan regex dan menyimpannya secara internal ke dalam sebuah form yang akan membuat pencocokan pola lebih efisien. Apabila menggunakan konstanta string, awk harus terlebih dahulu mengubah string ke dalam bentuk internal dan kemudian melakukan pencocokan pola.
3) Konstanta regex merupakan bentuk yang lebih baik dibanding konstanta string, itu menunjukkan dengan jelas bahwa Anda menggunakan regex sebagai alat pencocokkan pola.