A RegEx vagy RegExp a **Regular Expression** rövid formája. Magyarul talán a "szabályos kifejezés" lenne a megfelelő fordítás. Ez a leírás azért született, hogy elindítson a technika megismerésében, korán sem tekinthető teljesnek.

A RegEx lehetőséget ad szabályok, azaz minták egyszerű leírására. Ezekkel a mintákkal aztán sok hasznos dolgot tehetünk. Kereshetünk rájuk egy stringben, vagy kicserélhetjük őket valamilyen szabály szerint. Használhatjuk adatellenőrzésre vagy szerkezetek (pl. dátum) szétdarabolására, értelmezésére.

Essünk túl a kötelező analógián: a DOS-ból jólismert *joker karakterek* is kifejezéseket írnak le, amiknek fájlokat feleltetünk meg, vagy van egyezés, vagy nem. A `ka*.doc` és `ka???.doc`kifejezések közül a **kalap.doc** mindkettőnek, még a **kapa.doc** csak az elsőnek felel meg.

Hogy még egy kicsit rosszabb legyen, mielőtt jobb lesz: DOS-os `ka*.doc` RegEx megfelelője:**ka.*\.doc** a `ka???.doc` peig nem más, mint **ka.{3}\.doc**

RegEx operátorok

A DOS-os példához hasonlóan a mi mintáink is konkrét karakterekből (szavak, szótöredékek), és speciális jelentésű operátorokból épülnek fel.

Karakter megfeleltetés

. (pont)
: Bármilyen karakter: A `b.ka` kifejezésnek megfelel a **béka** és **bika** szó is.

[*karakterek*]
: A kapcsoszárójelek között felsorolt karakterek valamelyikével megegyező karakter: A`b[éa]ka` kifejezésnek megfelel a **béka** és **baka** szó, a **bika** viszont nem. A - (minusz) jellel tartományt is megadhatunk. Például `[0-9]` megfelel bármely számjegynek vagy `[a-zA-z]`bármely kis vagy nagybetűnek.

[^*karakterek*]
: A kapcsoszárójelek között felsorolt karakterek egyikével sem egyező karakter (az előző operátor tagadása): A `b[^é]ka` kifejezésnek nem megoldása a **béka**. A **baka** és **bika**viszont igen.

Többszörözés

?
: A megelőző minta 0 vagy 1 alkalommal fordul elő: A `borda.?` kifejezés igaz a **borda** és a**bordal** szavakra is.

+
: A megelőző minta 1 vagy több alkalommal fordul elő: A `bo.+ka` kifejezésnek megfelel a**boróka**, a **boka** viszont nem.

*
: A megelőző minta 0 vagy több alkalommal fordul elő: A `bo.*ka` kifejezésnek már megfelel a**boka** és **boróka** is.

{*m*,*n*}
: Segítségével megadható minimum és maximum vagy pontosan megadott számú előfordulás - {3} pontosan 3 előfordulás; {3,} legalább 3 előfordulás; {2,5} legalább 2 legfeljebb 5 előfordulás; {,10} legfeljebb 10 előfordulás. A `d.{,5}ány` igaz minden esetben, ha**legfeljebb** 5 karaktert kell helyettesíteni, például a **dolmány** esetén. A **díszkópatkány**viszont már nem akad fenn rajta.

Horgonyok

Az előzőekben nem szemléltettem, de a felsorolt kifejezések akkor is igazak ha a vizsgált string belsejében találhatók meg. A `b.ka` igaz a **bikaviadal** mintára is.

^
: A minta eleje: Ezzel jelezhetjük, hogy a kifejezést a minta elején keressük. A `^béka`kifejezésnek megfelel a **békanyál** minta, a **kecskebéka** viszont nem.

$
: A minta vége: Az előző horgonyhoz hasonlóan a minta végét testesíti meg. A `ék$` mintának megfelel minden erre végződő szó (kerék, pék).

Természetesen kombinálhatók is. A `^p.k$` kifejezés csak akkor igaz, ha az input pontosan egy hárombetűs szó. A **legpikánsabb** nem megoldása, ahogy a **pikáns** sem. A **pék** vagy **pók** viszont jó.

Logika

|
: Vagylagos egyezés: Két lehetőség közé téve bármelyikkel való egyezés találatot produkál. Gyakorlati példához picit előre kell ugorjunk, a normál (kerek) zárójelekre, jelen felhasználás viszont nem kíván különösebb magyarázatot: `ka(lap|bát)`

( )
: Kifejezések csoportosítása: Nem csak a vagylagos egyezés az egyetlen lehetséges felhazsnálás. Egy csoportot létrehozva elláthatjuk paraméterrel például a `(hókusz)?pók`segítségével a **hókuszpók** és **pók** szavak is megtalálhatók. A csoportokra később hivatkozhatunk is, ez cserénél vagy stringek értelmezésénél lesz hasznos.

Escape-elés

Ezek az operátorok lefednek néhány gyakran használt karaktert is. Ha például egy pont karaktert nem speciális értelmében szeretnénk hazsnálni, hanem egy pontként a C-ból, PHP-ből, Javascriptből megszokott módon \ (backslash) karakterrel tudjuk megfosztani speciális jelentésétől.

Összetett példák

Dátum feldolgozás

`[0-9]{4}[^0-9]*[0-9]{2}[^0-9]*[0-9]{2}`
Hogy is olvasandó a példa? `[0-9]{4}` négy darab számjegy (év); amelyet követ `[^0-9]*` nulla vagy akár több NEM számjegy karakter (esetleges elválasztó); majd `[0-9]{2}` két számjegy (hónap); utána ismét `[^0-9]*` jöhet elválasztó; és végül `[0-9]{2}` két újabb számjegy.

Ezzel még csak félmunkát végeztünk. Tudunk azonosítani egy dátumot, de nem tudunk hivatkozni az egyes tagokra. A kerekzárójellel emeljük ki azokat a csoportokat amik számunkra érdekes adatokat hordoznak.

`([0-9]{4})``[^0-9]*`*`([0-9]{2})`*`[^0-9]*`*`([0-9]{2})`*
A használt nyelvnek megfelelő stílusban (általában egy tömbben) kapjuk vissza a megjelölt csoportok tartalmát a RegEx futtatása után. Például PHP-ben:

```
$datum = "2006. 07. 22.";

ereg ('([0-9]{4})[^0-9]*([0-9]{2})[^0-9]*([0-9]{2})', $datum, $talalat);

var_dump ($talalat);
```
Eredménye a következő:

```
array(4) {

  [0]=>

  string(12) "2006. 07. 22"

  [1]=>

  string(4) "2006"

  [2]=>

  string(2) "07"

  [3]=>

  string(2) "22"

}
```
A tömb nullás sorszámú rekeszébe kerül az egész kifejezésnek megfelelő minta (látható, hogy a végső pont nincs benne, hiszen a keresett kifejezésünk véget ér a napot jelölő két számjeggyel). Az egyes sorszámtól kezdve a tömbbe kerültek az általunk (kerekzárójellel) megjelölt csoportok. A kiindulási dátum lehetett volna "2006-07-22" vagy "20060722" is, a kifejezés mindet megfejti.

Bizonyos dátumformátumok nem pótolják ki kétszámjegyűre az adatokat. Pédlául "2006/7/22". Ilyenkor az elválasztók jelenléte a döntő. Az elválasztók között mindig van egy vagy több számjegy:

`([0-9]{4})``[^0-9]*`*`([0-9]+)`*`[^0-9]*`*`([0-9]+)`*
Nézzünk egy PHP példát a csrére. A következő függvény a tetszőlegesen elválasztott dátumformátumot "-" jellel elválasztottra konvertálja (ahogy például a MySQL szereti).

```
$datum = "2006. 7. 22.";

$ujdaum = ereg_replace ('([0-9]{4})[^0-9]*([0-9]+)[^0-9]*([0-9]+)[^0-9]*', '\1-\2-\3', $datum);

echo ($ujdaum);  // eredménye: 2006-7-22
```
Itt megtoldottuk a mintát egy `[^0-9]*` kóddal, hogy lenyeljük az esetleges elválasztó jeleket a dátum végéről. A \1 \2 \3 pedig referenciák a zárójellel kiemelt csoportokra.

E-mail ellenőrzés

`^[0-9a-z\.-]+@`*`([0-9a-z-]+\.)+`*`[a-z]{2,4}$`
Olvassuk el: `^` a minta elején; egy vagy több alfanumerikus karakter, a pont és kötőjel is megengedett (A pont speciális karakter, egy backslash-sel jelöljük, hogy nem szeretnénk, hogy most értelmezze a RegEx motor. A minusz szintén speciális jelentéssel bír a kapcsoszárójelek között. Az ő esetében a backslash-módszer néhány értelmezőnek gondot okoz, ezért hogy megfosszuk speciális jelentésétől az utolsó kell legyen a záró kapcsoszárójel előtt.) Ezek után egy `@`(at) karakter következik; majd `[0-9a-z-]+` egy vagy több alfanumerikus karakter; amit `\.` egy pont követ. A mintát egy `[a-z]{2,4}` legalább 2 legfejlebb 4 betűből álló rész (TLD) zárja `$`.

A kifejezés közepén (kiemelt rész) képezünk egy csoportot, aminek engedélyeztük az ismétlődést. Ennek hatására nem csak a "user@*domain*.tld" felel meg, hanem ugyanúgy a "user@*subhost.host.domain*.tld" is.

```
$email = "gipsz.jakab@szerver.szervezet.hu";

$megfelel = eregi ('^[0-9a-z\.-]+@([0-9a-z-]+\.)+[a-z]{2,4}$', $email);
```