Reguláris kifejezések

Szövegfeldolgozás elmélet és gyakorlat regexekkel

A szövegekben keresés, egyes részek cseréje, kiemelése már az informatika kezdete óta foglalkoztatja a szakembereket. A regexek segítségével rendkívül kifejező módon tudunk részleteket megfogni egy hosszabb és nem szigorúan szabályos szövegből is. Ennek elméleti és gyakorlati kérdéseit boncolgatjuk a következő kilenc oldalban.

Regex történelem

Reguláris kifejezés, angolul Regular Expression. Eredetileg a neuronfiziológiában vezették be ezt a fogalmat az 1940-es években. Éredekes nem, hisz akkor hol voltak még a maihoz hasonló számítógépek? Aztán jópár évvel később Stephen Kleene kitalálta a Reguláris Halmazokat mint matematikai elméletet, és ehhez vezetett be egy jelölésmódot, amit Regular Expression-nek hívott. Akit érdekelnek a matematikai részletek (engem nem), annak itt a referencia: Robert L. Constable, "The Role of Finite Automata in the Development of Modern Computing Theory," in The Kleene Symposium, eds. Barwise, Keisler, and Kunen
(North-Holland Publishing Company, 1980).

A regexek első gyakorlati felhasználása Ken Thompson nevéhez fűződik (remélem nem kell bemutatnom, ha igen, akkor sürgősen do google!), aki „Regular Expression Search Algorithm" című cikkében vezeti be a reguláris kifejezések használatát szövegfeldolgozásra.

Ő írta meg a qed nevű szövegszerkesztőt, ami a Unixon jól ismert ed kifejlődéséhez vezetett. Ezekben a szövegszerkesztőkben már volt regex kiértékelő, így lehetővé vált bonyolultabb szövegrészek megtalálása és cseréje is.

Az ednek volt egy parancssori eszköze, ami szövegfájlok soraira egyező reguláris kifejezéseket nyomtatott ki. Ez volt a "Global Regular Expression Print", rövidítve grep. Bízom benne, hogy nem Unixon felnőtt programozók is hallottak erről a programról.

Ezek a kezdeti programok persze még sokkal egyszerűbb reguláris kifejezéseket ismertek, mint a mai értelmezők, de (mint minden fejlődés ebben a szakmában) a regexek is iteratív módon fejlődtek.

Ezután jött a jóval több metakaraktert használó egrep, ami már teljesen más elven működött, mint elődje. A grep Deterministic Finite Automat-ot (DFA) használt, míg az egrep Nondeterministic Finite Automat-ot (NFA). Egyszerűen megfogalmazva a DFA gyors, de butább, az NFA egyes esetekben rettentő lassú, de sokkalta okosabb, és szabadságot ad a regexeken keresztül a motor közvetlenebb vezérlésére. A mai regex motorok zöme NFA.

Majd jött a sed, az awk, a lex, amelyek mind valamilyen szempontból továbbfejlesztették a regexek nyelvtanát. A sokféle nyelvjárás között a POSIX szabvány próbált rendet rakni. Legfőbb előnye, hogy bevezette a locale fogalmát, így a betű végre nem csak az angol ABC karaktereit jelentette. Mi magyarok ismerjük a szakmában az „angol diszkriminácót", így örülünk a Posix törekvéseinek.

Napjainkban a Perl az a környezet, amely a reguláris kifejezések használatában és újításokban a fő húzóerő. A példákban a .NET Framework regex osztályait használjuk, amelyeket a Perl 5 regexei alapján modelleztek, így nagyon magasszintű regex támogatást kapunk.

Bevezetés

A regex a reguláris kifejezés rövidítése. A cikkünkben tárgyalt szövegeket bogarászó regexek már szokszor nem regulárisak matematikai értelemben, ezért általában egyszerűen regexként hivatkozunk rájuk, megkülönöztetve őket a matematikai reguláris kifejezésektől.

Nos, mi is az a regex? Egy olyan leíró nyelv, amely segítségével szövegek különböző részeit ragadhatjuk meg, írhatjuk le. Gondoljunk a fájlrendszerre:

```
dir a.txt
```

Ez kilistázza az a.txt fájlt. Mi van, ha az összes szövegfájl kell?

```
dir *.txt
```

Bevezettünk egy *metakaraktert*, a *-ot (csillagot), amit úgy definiáltunk, hogy egyezik bármilyen fájlnévre. Nos, a regexek hasonlóak, csak sokkal több metakarakter található bennük, így sokkal gazdagabban fogalmazhatjuk meg az illesztendő szöveget.

Kiinduló példánk a következő lesz. Szeretnénk egy szövegben megkeresni a *dadogásokat* dadogásokat. Gyakori szövegszerkesztési hiba az ismétlés, ezt kellene megkeresni egy tetszőleges szövegben. Azt gondolnánk, minek ide regex, sima sztringkezelő eszközökkel is megoldható a probléma.

Például feldarabolhatnánk a szöveget szavakra (whitespace-ek mentén), majd végigmenve a listán összehasonlítjuk az egymás után következő szavakat. Ha egyeznek, ismétlést találtunk.

Igen ám de lehet, hogy a szavakat markup tagok határolják, mint pl. egy html szövegben:

```
Ez is <i>dadogásnak</i> <u>Dadogásnak</u> számít.
```

Ebben az esetben is meg kell találni az ismétlődő szavakat. Természetesen ez és minden más probléma is megoldható alapvető sztringműveletekkel (Find, Split, Replace), de sokszor egy regexes megoldás sokkal egyszerűbb lesz.

A problémát megoldó regex így néz ki:

```
→\b(\w+)(\s|<[^<>]+>)+(\1)\b←
```

A regex csak a nyilak közötti rész, de a nyilakat mindig kiírom mind a szövegekben mind a kódblokkban, hogy jelezzem ha regexről beszélünk. A cikk célja, hogy a végére mindenki számára magától értetődő legyen ez a regex.

Karakteregyezések
Egy karakter saját magával mutat egyezést, ha nem vezérlőkarakter. A példákban az egyezéseket mindig aláhúzással jelölöm. Ahol fontos, ott kiírom, hány egyezést találna a regex motor.

```
→e←
Mesterkurzus – 2 egyezés
```

A legtöbb regex motor átkapcsolható kis-negybetűre nem érzékeny módra, ilyenkor értelemszerűen alakulnak az egyezések. .NET-ben ez a RegexOption.IgnoreCase opcióval érhető el.

```
→e←
Embergyerek – 4 egyezés
```

Karakterhalmaz egyezések
Egy karakterhalmaz egyezést mutat ugyanazzal a karakterhalmazzal, szóhatártól függetlenül.

```
→ek←
Mekk Elek legyek – 3 egyezés
```

A regexekben minden karakter számít, még a whitespacek is.

```
→1492. ←
Született: 1492. 08. 12.  – 1 egyezés
```

de

```
→1492. ←
Született: 1492.08.12.  – 0 egyezés
```

Ez fontos, mert sokszor hajlamosak vagyunk egy bonyolultabb regexet picit szellősebbé tenni szóközökkel, ám ettől megváltozik a regex viselkedése.
Egyes regex implementációkban, mint a Perl vagy a .NET, lehetőség van whitespace-ek és kommentek használatára a regexekben. .NET-ben a RegexOptions.IgnorePatternWhitespace opció után a whitespace-ek nem számítanak a patternben, és # után még megjegyzéseket is lehet fűzni a sorokhoz. De akkor ebben az esetben hogyan írjuk le a whitespace-eket? Hasonlóan, mint a legtöbb stringet feldolgozó programnyelven: escape szekvenciákkal. A következő táblázatban megtekinthetjük a legfontosabb (de nem az összes) helyettesítő karaktert.

Karakter	Leírás
Közönséges karakterek	Mind, kivéve . $ ^ { [(|) * + ? \
\b	Visszatörlés (backspace) \u0008 ha [] karakterosztályban van, egyébként szóhatár (ezekről bővebben kicsit később)
\t	Tab \u0009.
\r	Kocsivissza \u000D.
\n	Újsor karakter \u000A.
\x20	Egy ASCII karakter hexa kóddal, pontosan két digiten leírva. Ez pl. egy szóköz.
\cC	ASCII vezérlőkarakter (32-nél kisebb kódú karakter), ez pl. a CTRL-C
\u0170	Egy Unicode karakter, pontosan négy hexa számjeggyel leírva, ez egy nagy Ő betű
\	Ha nem escape-elt karakter előtt van, akkor egyszerűen elhagyásra kerül, így marad a mögötte levő karakter. Pl. \g egyszerűen egy g betű.

A \uxxxx hexa szám a karakter ún. code pointja az unicode táblázatban, nevezzük egyszerűen karakterkódnak. Látható, hogy ezt a jelölést nyugodtan lehet használni regexekben, így biztos nem fürdünk be a sunyiban o betűvé átkonvertálódott ő betűkkel (a szövegszerkesztők miatt). A kódokat legegyszerűbben a Windows Character Map-ban találhatjuk meg.
A visszafele perjel (\) beírásához duplázni kell (\\).

Karakterosztályok
Eddig csak konkrét karakteregyezéseket vizsgáltunk meg. Az f betű az f-fel egyezik, kész. Természetesen lehet használni olyan konstrukciókat, amelyek több mint egy*féle* karakterrel egyeznek, ezek a karakterosztályok.
Karakterosztályokat →[...]← (szögletes zárójelek) között lehet definiálni. Az osztályban felsorolt bármelyik karakter szerepelhet az egyezésben, de nagyon fontos, hogy pontosan *egyetlen* karaktert helyettesít egy karakterosztály kifejezés, nem többet.

```
→[rsk]←
Mesterkurzus - 5 egyezés
```

Látható, hogy a →[rsk]← jelentése: egy karakter, ami r vagy s vagy k lehet.

```
→[0123456789]←
Született: 1492. 08. 12. - 8 egyezés
```

A →[0123456789]← bármilyen decimális számjegyre illeszkedik, ezért 8 ponton egyezik a második sor tesztszövegével.
Karaktertartományokat is megadhatunk karakterosztályokban - (mínusz)-szal elválasztva, így nem kell felsorolni minden egyes karaktert.
Ez a példa egzaktul azonos az előzővel, csak rövidebb:

```
→[0-9]←
Született: 1492. 08. 12. - 8 egyezés
```

A következő regexet így kell értelmezni: bárhol a szövegben egy decimális számjegy, amit egy decimális számjegy követ. Lehet, hogy így túl analitikusan hangzik, de bonyolultabb regexeknél a túl intuitív, „belelátom én mit csinál egyszuszra" gondolkodásmód gyakran tévútra csal.

```
→[0-9][0-9]←
Született: 1492. 08. 12. - 4 egyezés!
```

Karakterosztályon kívül a - jel közönséges karakter:

```
→[0-9][0-9]-←
Született: 1492-08-12. - 2 egyezés!
```

Érdemes odafigyelni, hogy sok karakter másképp viselkedik karakterosztályon kívül és belül. De ez még nem minden!
A - jel karakterosztály elején és végén közönséges karakter. Nézzük kontrasztba állítva. Ebben a példában a regex a a-tól z-ig terjedő karaktert tartomány jelöli ki:

```
→[a-z]←
c - d - 2 egyezés
```

Itt viszont az első mínusz közönséges karakter:

```
→[-a-z]←
c - d - 3 egyezés
```

Karakterosztályon belül az utolsó pozíción is közönséges karakter lenne, így az →[a-z-]← azonos a fenti regexel.
Egy karakterosztályban több tartomány és karakter is felsorolható vegyesen is. Például:

```
→[eza0-2]←
Született: 1975-01-10 - 8 egyezés
→0x[0-9abcdefABCDEF][0-9abcdefABCDEF]←
0x15, 0xaF, 0x5H, 0x1B, hexa számok - 3 egyezés
```

Gyakran egyszerűbb megfogalmazni úgy egy karakterhalmazt, hogy bármilyen karakter, kivéve ezt és ezt. Erre való a negált karakterosztály. A jelölésmódja egy ^ (kalap) a karakterosztályt jelölő [(nyitó szögletes zárójel) után közvetlenül. A következő példa jelentése: bármilyen karakter, kivéve a 0-9-ig terjedő tartományt, magyarul bármi, ami nem szám:

```
→[^0-9]←
Született: 1492. 08. 12.  - 16 egyezés
```

Nem első pozíción már közönséges karakter a ^:

```
→[0-9^e]←
Szü^letett: 1492. 08.^ 12.  - 12 egyezés
```

Előre definiált karakterosztályok

Vannak gyakori esetek, amelyeket nem fontos hosszan karakterosztályként definiálni, hanem vannak előre elkészített rövidítések rájuk.

Gyakran kell a →[0-9]← karakterosztályt használni, amit a →\d← helyettesíthet. A következő példa négy egymás utáni számjegyre ad egyezést. Figyelem! Nem négydigites számokat keres! Mint már tudjuk, az egyezések függetlenek a hétköznapi értelemben vett szóhatártól, így az első hosszú számban három egyezés is lesz, a három egymást követő négy számjegyből álló blokk:

```
→\d\d\d\d←
1258632455458: 1492. 08. 12.  - 4 egyezés
```

A \D a →[^0-9]← karakterosztállyal egyezik meg, azaz nem számjegy.

```
→\D\D\D\D←
Született volna: 1492. 08. 12.  - 4 egyezés
```

Az egyezések kifejtve:

```
0 => Szül
1 => etet
2 => t vo
3 => lna:
```

Sajnos az aláhúzásos jelölésmód időnként nem egyértelmű, ezért néha kifejtem a kimenetet. Amikor progamozzuk a regexeket, ebből nincs gond, mert a találatokat egy kollekcióban kapjuk vissza.

A →\w← bármilyen betűt, számot vagy aláhúzásjelet (_) jelent. Nem bármilyen karaktert, hanem bármilyen betűt, beleértve a gonosz ő̋ű betűket is. Pontosabban: ez attól függ. A .NET-es implementáció alapban minden betűt beleért, de átkapcsolható ECMA módba is (RegexOptions.ECMAScript), ahol a →\w← == →[a-zA-Z0-9_]←. Azaz ettől kezdve csak az angol ABC betűivel foglalkozik, így az ékezetes betűinket mind kihagyná.

Figyelem! Más implementációban lehet, hogy eleve így működik a regex motor, azért éles bevetés előtt mindenképpen tesztelni kell ékezetes betűkkel is az egyezéseket!

Normál módban:

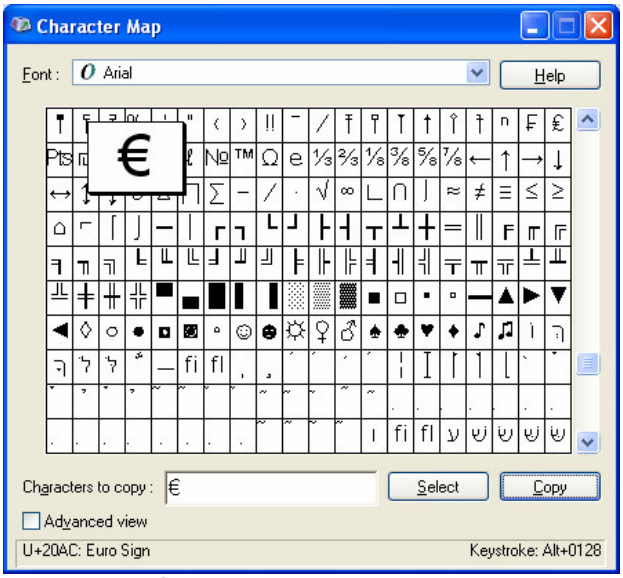
```
→\w\w\w←
Született: 1492. 08. 12.
0 => Szü
1 => let
2 => ett
3 => 149
```

RegexOptions.ECMAScript esetén látható, hogy az ü betű nem tartozik bele a →\w←-be, így az első hármas betűcsoportot csak a „let”-nél találja meg a motor:

```
→\w\w\w←
Született: 1492. 08. 12.
```

```
0 => let
1 => ett
2 => 149
```

Mi a helyzet az euro (€) karakterrel? Az betű? Egyáltalán, hol található ez a billentyűzeten? Nos, első körben én sem találtam. Aztán rákerestem az euro-ra az unicode.org-on **[1]**, ahol találtam egy riportot **[2]**, mely szerint az euro karakter a 2.1-es unicode szabványban jelent meg. Most egyébként (2003. október) a 4.0 unicode szabvány az aktuális.
Nos, az euro a 20AC hexa kódot kapta. Szép, mi? Hol vannak már az ASCII 7 és 8 bites számokkal ábrázolt betűk!
Kitartóbbak a Character Map-ban is megtalálhatják, legalábbis XP+SP1-en biztos:

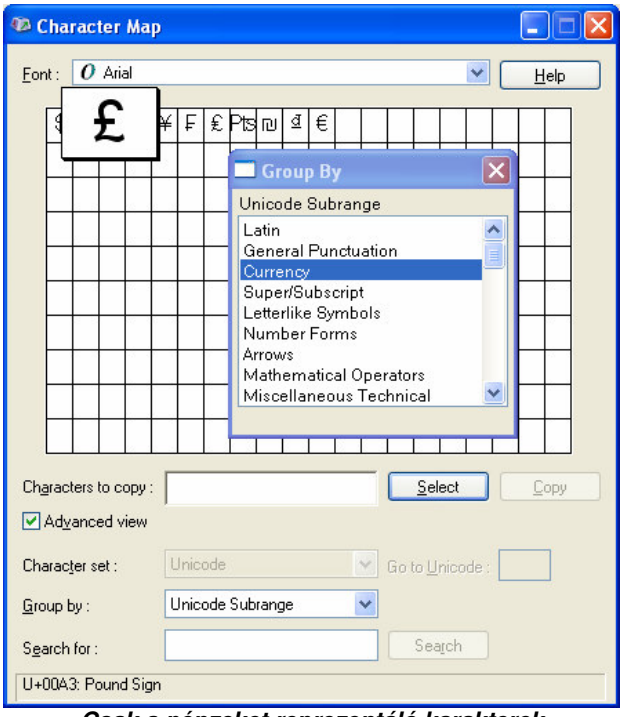


Euro szimbólum a Windows XP Character Map-ben

Nos, a kérdés ugye az, hogy a →\w←-be beletartozik-e az euro? Rövid teszttel kiderül, hogy *nem*. Az euro *szimbólum* kategóriájú karakter, nem betű. A .NET regex motor nagyfokú unicode támogatást ad, így a karakterek osztályozásánál is az unicode szabvány által meghatározott kategóriákat használja **[3]**. Olyannyira, hogy ezt még ki is vezették nekünk. A →\p{Kategórianév}← kifejezéssel hivatkozhatunk a megfelelő kategóriába tartozó karakterekre. A kategóriák nevét a **[3]** táblázat tartalmazza. Nézzünk néhány érdekes példát!

```
→\p{Ll}← #Letter, lowercase, azaz kisbetű
Született A Nagybetű € $ % ^ ½ ¼ .+
→\p{Lu}← #Letter, uppercase, azaz nagybetű
Született A Nagybetű € $ % ^ ½ ¼ .+
→\p{No}← #Number, other, azaz egyéb szám
Született A Nagybetű 45 € $ % ^ ½ ¼ .+
→\p{Ps}← #Punctuation, open, azaz nyitó írásjel
(alma) [körte] {szilva}
→\p{Po}← #Punctuation, other, egyéb írásjel
€ $ % ^ ½ ¼ .+ - _ "Idézet.", 'ez is!'
→\p{Sc}← #Symbol, currency, pénz szimbólum
€ $ % ^ ½ ¼ . + - _ / * ^ ~ Ft £ ¥ £
```

Csak meglett az euró, a pénz szimbólumok között találjuk!
A második alkategória karaktert el lehet hagyni, így a →\p{L}← az összes betűt jelenti, amiben nincsenek benne a számok és az aláhúzásjel, azaz nem ugyanaz, mint a →\w←.
Egyébként a furcsa karakterek megkereséséhez hasznos lehet a Character Map Advanced nézete, amikor Unicode kategóriák szerint szűrve láthatjuk a karaktereket.



Csak a pénzeket reprezentáló karakterek

A →\W← (nagy W) minden nem betűt jelöl, azaz →^\w←.
A →\s← bármilyen whitespace karaktert jelent. A whitespace-eket is az Unicode szabvány rögzíti, de leegyszerűsítve a szóköz, tabulátor, kocsivissza és a soremelés tartozik bele. Vannak még extra karakterek is (pl. függőleges tabulátor), de ezek általában nem érdekesek számunkra. A pontos lista így néz ki:

```
→[\f\n\r\t\v\x85\p{Z}]←
```

A Z unicode kategória a szeparátor karaktereket jelöli, a 85 hexa kódú karakter pedig a szóköz nem PC-s rendszerekben (bizarr).

```
→\s←
alma,_majd_egy_tab:____és_más
```

A →\S← (nagy S) minden nem whitespace-t jelöl, azaz →^\s←.

Az univerzális dzsóker: a pont
A →.← (pont) bármilyen karakterrel egyezik kivéve az újsor (\n) karaktert. Ha RegexOptions.Singleline módban vagyunk, akkor az újsorral is.

```
→.t.←
Született ma - 2 egyezés
0 => ete
1 => tt(space)
```

```
→.....←
Született<i>ma</i> - 3 egyezés
0 => Szüle
1 => tett<
2 => i>ma<
```

Normál módú viselkedés (a \r\n a kocsivissza-soremelés páros, amik nem látszanának):

```
→.......← #7 db karakter
Első sor(\r\n)
Második sor(\r\n)
```

Látható, hogy az első sorban már nem volt másik 7 egybefüggő karakterhalmaz, így csak egy egyezést tapasztalunk. A második sorban úgyszintén.
Ezzel szemben RegexOptions.Singleline módban a sorvégi újsor (\n) karakter nem állítja meg a motort, így a sorokon átívelve is egyezést talál a →.....←:

```
Első sor(\r\n)
Második sor(\r\n)
0 => Első so
1 => r(\r\n)Máso
2 => dik sor
```

A második egyezésben benne van az első sor „r" betűje, a „kocsivissza-soremelés" páros és a „Máso" karakterek.
A pontot nagyon gyakran fogjuk arra használni, hogy ismeretlen szövegre állítsunk fel egyezéseket. A következőkben sok példát fogunk látni a használatára.

Pozícionális karkterek (Anchors vagy Atomic Zero-Width Assertions)
Az eddig látott karakterosztályok, jokerek mindig helyet fogaltak el, azaz miután a regex motor egyezést talált, továbblép egy karakterpozícióval mind a forrásszövegben mind a regexben, és onnan keres a regex maradékára egyezést. Például a →.t.← esetén (tesztsztring: Született ma) az első →.← talál egy karaktert, az „S"-t, majd a motor továbblép a regexben a →t←-re, és megnézi, hogy az illeszkedik-e a sorok következő karakterre, a „z"-re. Mivel nem, eldobja ezt a próbálkozást, és továbblép a forrásszövegben a „z"-re, visszatekeri a regexet az elejére, és nekiáll a →.←-ot újra ráilleszteni az „ü" karakterre, stb.
Azaz a lényeg, hogy a normál karakter vagy karakterosztály egyezések helyet foglalnak el, továbbléptetik a forrásszöveget.
Ezzel szemben a pozícionális karakterek nem foglalnak el helyet, csak kijelölnek egy pozíciót a forrásszövegben, aminek teljesülni kell ahhoz, hogy egyezést kapjunk.
A →^← (kalap) a szöveg vagy sor elejét jelenti. Alapban szöveg elejét, RegexOptions.Multiline esetén a sor elejét. Azaz a multiline üzemmódban a regex motor soronként dolgozza fel a szöveget, hasonlóan a grep-hez. (A normál eset a sed működéséhez hasonló.)
Normál mód:

```
→^..←
Első sor(\r\n)
Második sor(\r\n)
```

Multiline mód:

```
→^..←
Első sor(\r\n)
Második sor(\r\n)
```

A →$← (dollár) a szöveg vagy sor végét jelenti. A multiline ugyanúgy hat rá, mint a →^←-ra.

```
→ma$←
alma alma
```

A második példa csak azokra a sorokra mutat egyezést, amelyekben pontosan és csakis az „alma" szó szerepel:

```
→^alma$←
alma
almama
```

A

```
→^←
```

minden sorra egyezést mutat (multiline módban), azaz nem túl praktikus regex. Habár, ha ezzel a kifejezéssel szétszedünk darabjaira egy szöveget, sorokra bontva kapjuk vissza. Mondjuk ennél egy String.Split egy cseppet gyorsabb lenne, de ezt azért még lehet tovább alakítani, például szűrni a sorokat.
A

```
→^$←
```

az üres sorokat válogatja ki, amikben még whitespace sincs (természetesen ez is multiline módban működik jól).
A →\b← szóhatáron egyezésre való. Szóhatár a →\w← és →\W← átmenet, tetszőleges irányban, így a →\b← szó elejének és végének keresésére is jó.
Baloldalt behatárolt „al” sztring:

```
→\bal←
szilva alma hatalmas almamáter
```

Az „alma” szó keresése kétoldali határral:

```
→\balma\b←
szilva alma hatalmas almamáter
```

Egybetűs, kéttagú urn-ek keresése:

```
→\burn:\w:\w\b←
Például: urn:a:b (vagy urn:x:y)
```

A →\B← (nagy B) *nem* szóhatáron egyezést jelöl ki.

```
→\Balma←
A hatalmas alma hatalma.
```

A →\A← olyan mint a →^←, csak mindig a szöveg és nem a sor elejét jelenti függetlenül a multiline opciótól.
A →\z← (figyelem, kis z) pedig olyan mint a →$←, csak mindig a szöveg és nem a sor végére mutat egyezést.
A →\Z← (nagy Z) annyival engedékenyebb, mint a kisbetűs párja, hogy a szöveg végén még lehet egy plusz soremelés is. Ez amúgy igaz a →$←-ra is.
Vannak még további pozícionális kifejezések is (pl. (?!...)), amelyekre most terjedelmi okokból nem térek ki. **[6]** mindegyiket tárgyalja.
Pozícionális karktereknél nagyon oda kell figyelni, hogy Windowsban a sorok vége nem \n, hanem \r\n, ami miatt sokszor nem jól működnek a Unixon helyesen zenélő regexek. Agyrém, de erre fel kell készülni.

Számosság (Quantifiers vagy Modifiers)
Amit eddig láttunk, az csak a jéghegy csúcsa. A regexek első igazi erőssége a számosság adta flexibilitás.
Miről is van szó? Az →a← egy darab a betűt jelöl, fogyaszt el. Az →a?← („a” betű, utána egy kérdőjel) viszont azt jelenti, hogy az „a” karakter 0 vagy egyszeri előfordulása. Ez azt jelenti, hogy akkor is egyezést mutat, ha az adott pozíción van „a” betű, de akkor is, ha nincs. Azaz a kérdőjel jelentése: az előtte levő karakter vagy karakterosztály opcionális.
Az alábbi példa törtszámokat próbál meg elcsípni, a tört rész opcionális:

```
→\d\d\.?\d?\d?←
13.85, 12.5, 15., 45
```

A →\.← a pont karaktert jelöli, csak mivel az metakarakter, meg kellett védeni egy visszafele perjellel. Látható, hogy csak az első két számjegy kötelező, az utána következő karakterek nem. Sajnos ez a regex megengedi a „15.”-t is, ami nem szabályos. Amíg nem ismerjük a csoportosítást, addig ezen nem tudunk segíteni.
A →+← 1 vagy több (legalább 1) előfordulást jelöl. A példa az összefüggő számsorokat keresi meg:

```
→\d+←
12, 5445, 12.345, 0.33
```

A →*← 0 vagy több (bármennyi) számosságot definiál. A ponttal együtt használva könnyedén leírható a bármiből bármennyit minta:

```
→.*←
alma  - 1 találat
```

Ha belegondolunk, ez a minta mindenre egyezik még az üres sorra is, és a bármilyen karaktereket tartalmazó sorokra is.
További számossági jelzők is léteznek. A →{n}← jelentése: pontosan n előfordulás. A példa három összefüggő betűt keres:

```
→\w{3}←
Cica, kutya, sas, őz, ló, kecske
```

Azaz nem hárombetűs szavakat keresünk, ahhoz be kell vetnünk a szóhatárt is, mindkét oldalról:

```
\b\w{3}\b
Cica, kutya, sas, őz, ló, kecske
```

→{n,}←: legalább n találat. Minimum 3 digites egész számok keresése:

```
→\d{3,}←
123, 5445, 12.345, 0.33, alma58942-szilva
```

Láthatóan a tizedes tört utáni részt is megtalálja. Hogy azt kiszűrjük még okosítani kell a regexünket (→(?<!\.)\d{3,} ←, az érdeklődők kedvéért :) .
És végül a →{n,m}← legalább n, de legfeljebb m darabszámot ír elő.

```
→g{1,2}y←
gyurgyalag, aggyad má
```

Az összes eddig látott számosságjelző mohó, azaz megpróbál olyan hosszú egyezést összehozni amennyit csak lehetséges.
Például a →\w{3,}← megpróbálja a lehető leghosszabb, de minimum három karakter hosszú betűcsoportokat megtalálni:

```
→\w{3,}←
Cica, kutya, sas, őz, ló, kecske - 4 találat
```

Azért mohó, mert nem elégszik meg a minimálisan megkövetelt darabszámmal.
Bármelyik számosságjelző mohóságát elvehetjük, ha mögé rakunk egy kérdőjelet:

```
→\w{3,}?←
Cica, kutya, sas, őz, ló, kecske - 5 találat
0 => Cic
1 => kut
2 => sas
3 => kec
4 => ske
```

Látható, hogy most megáll a feltételt minimálisan kielégítő számú karakternél, aztán folytatja a keresét. Ezért vágta ketté a kecskét.
Gyakoroljuk kicsit az eddigieket! Hogyan keresnénk meg a kikommentezett sorokat (//) C# kódban?

```
→???←
int i;
//long g;
  // dim i as Integer
```

Hogyan gondolkodunk? Soreleje, majd jön utána akárhány darab whitespace, majd két egymás utáni perjel, aztán a sorvégéig bármi. Elég könnyű lefordítani regexre:

```
→^\s*//.*$←
```

Csoportosítások (grouping)
Következő hatalmas fegyverünk a csoportosítás, melyet zárójelezéssel érünk el. Többféle okból csoportosítunk:

- a csoportokra használhatunk számossági jelzőket
- a csoportok által megfogott tartalomra hivatkozhatunk a regex többi részében (backreferences)
- a csoportok által elkapott tartalmat kinyerhetjük programozott eszközökkel

Mivel a számosság használható csoportokra, a korábbi törtszámokat kereső regexünket mostmár tökéletesre írhatjuk:

```
→\d+(\.\d+)?←
13.85, 12.5, 15., 45
```

Magyarra fordítva: minimum egy decimális számjegy, aztán egy opcionális csoport, ami belül úgy néz ki, hogy egy pont, aztán minimum egy számjegy. Azaz csak akkor fogjuk meg az egész rész után álló pontot, ha utána van számjegy, egyébként nem.
Egyszerű, de nem teljes email ellenőrző:

```
→\w+@\w+(\.\w+)+←
soci@netacademia.net, alma%@sexybabes.com
soci12@alma.korte.neta.com
```

Visszahivatkozások (backreferences)
Tegyük fel, hogy html tagok közötti kifejezéseket akarunk leírni regexszel. Első nekibuzdulásunkban megszüljük ezt:

```
→<\w+>[\w\s]*</\w+>←
<h1>alma</h1> és <h2> körte </h2> <p></p>
```

Az eddigiek alapján ennek teljesen érhetőnek kell lennie. Igen ám, de ez könnyen átverhető:

```
<h1>alma</xxx> - hibás!
```

Megeszi ezt is. Valahogyan meg kellene mondani, hogy a második kacsacsőrös részben azt akarjuk látni, amit az elsőben elkapott a regex motor.
Ehhez először be kell zárójeleznünk az elkapandó kifejezést:

```
→<(\w+)>[\w\s]*</\w+>←
```

Ez nem változtat semmit a kifejezés működésén, de a regex motor már tudja, hogy valami célunk van a zárójeles kifejezéssel, ezért megjegyzi azt.
Már csak az a dolgunk, hogy a zárótagnál hivatkozzunk a zárójeles tartalomra. Erre való a visszahivatkozó kifejezés:

```
<(\w+)>[\w\s]*</\1>
<h1>alma</xxx> és <h2> körte</h2>  <p></p>
```

A →\1← azt jelzi, hogy itt olyan tartalmat várunk el, amit a *balról legelső* zárójeles kifejezés fogott meg. Fontos megjegyezni a szabályt, balról az n., mert egymásba ágyazott zárójelek esetén így könnyű megtalálni, mire akarunk hivatkozni.
Az →\2← a második, ... kifejezésre hivatkozik. A visszafelé hivatkozás hatalmas lehetőség a regexekben, és ilyet csak az NFA motorok tudnak, ezért aztán a legtöbb engine NFA.

Elágazások (Alteration)
Ha a karakterosztályoknál megadhattunk választást egy karakterpozíción, akkor ezt miért ne tehetnénk meg nagyobb regex kifejezésekre is? Erre való az elágazás, melyet a →|← (pipe, cső, függőleges vonal) szimbólum reprezentál. A következő regex jelentése: „alma” *vagy* „körte” karakterek egymásutánisága:

```
→alma|körte←
alma, körte, körtealma, cser, hatalmas - 5 egyezés
```

Sokféle kommentet kereső kifejezés:

```
→^\s*(//|#|rem|').*$←
int i;
//long g;
        ' dim i as Integer
  rem dos komment
# unix comment
```

Az elágazások bármelyike lehet összetett regex is. A következő példa által ellenőrzött értékek behatárolását a kedves olvasóra bízom.

```
→\b0\d\b|\b1\d\b|\b2[0-3]\b←
15, 28, 21, 14, 5, 142
```

Gondolkodtató példák
Exponenciális számokat felfedező regex:

```
→((\d+)?\.)?\d+e[+-]?\d+←
3e8, 4e+4, 5e-8, 45.6e-5, .34e-6
```

Fájl elérési útból a fájlnevet kiszedő kifejezés:

```
→[^/]*$←
/winnt/system32/drivers/etc/lmhosts.sam
```

Mohó számosság esetén az első .* felemészt mindent, a második kifejezésnek csak a legutolsó szakaszt hagyja meg:

```
→^(.*)/(.*)$←
winnt/system32/drivers/etc/hosts.txt
0 => winnt/system32/drivers/etc
1 => hosts.txt
```

A mohóságát csillapítva megelégszik az első /-ig tartó legrövidebb kifejezéssel:

```
→^(.*?)/(.*)$←
winnt/system32/drivers/etc/hosts.txt
0 => winnt
1 => system32/drivers/etc/hosts.txt
```

Mi történik, ha a második csillag mohóságát is elvesszük? Semmi változás nem történik, mert miután az első kifejezés önmegtartóztató módon csak a „winnt" karaktersorozattal egyezik, a második minden visszafogottsága ellenére kénytelen elvinni a többit.
És a teljesség kedvéért: ha az első mohó a második nem, az első felszed mindent az utolsó perjelig, így a második kapja a maradék részt, ha mohó, ha nem.
Html tagek kitakarítása, például fórum szoftverekhez:

```
→</?\w+[^>]*>←
<html><head>
<link rel="stylesheet" type="text/css" href="/css/main.css">
<title>NetAcademia - A legjobbakat tanítjuk</title>
</head>
<b>Tanfolyami térképek:</b><br>
```

```
</html>
```

Az eddigiek után a kiinduló példánkban szereplő regex már gyerekjáték kell legyen:

```
→\b(\w+)(\s|<[^>]+>)+(\1)\b←
```

Ha nem, akkor érdemes újra elolvasni a cikket, és tesztprogramokkal (**[4]** és **[5]**) próbálgatni a kódokat (legalább kiderül, mennyi bug maradt benne :). Ez a leggyorsabb módja a regex tanulásának.
Utolsó példaként tetszőleges szepatárorokkal elválaszott, de év-hó-nap formátumú dátumok megtalálását nézzük meg:

```
→\D*(\d\d\d\d)\D(\d\d)\D(\d\d)\D*←
```

Hogy ezen példát hasznunkra tudjuk fordítani itt az ideje, hogy megnézzük programozott módon hogyan lehet elérni a regex szolgáltatásokat.

Regex programozás a .NET Frameworkben
Kiinduló osztályunk a System.Text.RegularExpressions.Regex lesz. Ez képes eltárolni egy regexet, amit aztán rászabadíthatunk egy sztringre.
Létrehozásakor megadhatjuk a kívánt regexet stringként, illetve a működési opciókat a RegexOptions enumerációs típus segítségével:

```
RegexOptions options = RegexOptions.IgnoreCase;
Regex regex =
  new Regex(@"\b(\w+)(\s|<[^>]+>)+(\1)\b", options);
```

Esetünkben lényeges, hogy a kis-nagybetű különbség ellenére két szót azonosnak tekintsünk, ezért a RegexOptions.IgnoreCase opció.
A regex által elkapott darabokat a következőképpen kaphatjuk meg:

```
MatchCollection matches = regex.Matches(input);
```

Az input a bemeneti stringünk. Az eredményeken könnyű végigiterálni:

```
foreach(Match match in matches)
{
  Console.WriteLine("Pos: {0} ", match.Index);
  Console.WriteLine("1. szó  : {0} ",
                    match.Groups[1]);
  Console.WriteLine("Ismétlés: {0} ",
                    match.Groups[3]);
}
```

A MatchCollection az összes megtalált kifejezést tartalmazza. Ezeket egyedi Match objektumokként érhetjük el a ciklusban. Minden egyes Match tartalmazza azt a szöveget, amit a regex elkapott. A Match.Index az egyezés pozícióját adja vissza a bemeneti szövegben.
Nekünk csak az első és a harmadik zárójeles kifejezés az érdekes, a középső nem, annak csak az volt a dolga, hogy lenyelje a két szó közötti html tagokat és whitespace-eket.
Szerencsére a zárójelezett tartalmakat közvetlenül elérhetjük a Match objektum Groups kollekcióján keresztül.
A 0. csoport mindig a teljes Match-et tartalmazza, ezért az első zárójeles regex (→(\w+)←) által elkapott tartalmat az első Group elemben érhetjük el:

```
Console.WriteLine("1. szó  : {0} ", match.Groups[1]);
```

Értelemszerűen a 3. csoport a match.Groups[3] mögött lesz.
Mit találunk a második csoportban? Nos, ez csak az igazán érdekes.

```
→(\s|<[^>]+>)+←
```

Ez a csoport akár többször is szerepelhet az egyezésben, ezért ezt nem lehet egyszerűen match.Groups[2]-ként elérni.
Ha a bementi sztring

```
Ez is <i>dadogásnak</i>  <u>Dadogásnak</u> számít.
```

akkor a match.Groups[2].Value-ban „<u>”-t találunk. Pedig ha belegondolunk, a második csoport 4-szer is működött: egyszer elkapta a “</i>”-t (→<[^>]+>←), aztán egy szóközt (→\s←), aztán mégegyet, majd a „<u>”-t. A Value jellemző csak az utoljára elkapott darabkát adja vissza!
Az összeset a csoport Captures jellemzőjén keresztül szedhetjük elő:

```
int i = 0;
foreach(Capture c in match.Groups[2].Captures)
{
  Console.WriteLine("{0}.: {1}", i++, c.Value);
}

0.: </i>
1.:
2.:
3.: <u>
```

A dátumnormalizálós példánkhoz az elkapott csoportok tartalmából kell összeállítani egy formázott dátumot, és azzal kell *kicserélni* a talált, rosszul formázott dátumnak látszó sztringet:

```
private void Run()
{
  string[] dates =
  {
    "2003/08/12",
    "2003.08.12",
    "2003.08.12....",
    "aa2003/08.12z"
  };
  foreach(string d in dates)
  {
    Console.WriteLine("{0} -> {1}",
    d, Normalize(d));
  }
}

public string Normalize(string date)
{
  Regex r =
  new Regex(@"\D*(\d\d\d\d)\D(\d\d)\D(\d\d)\D*");
  return r.Replace(date, "$1-$2-$3");
}

2003/08/12 -> 2003-08-12
2003.08.12 -> 2003-08-12
2003.08.12.... -> 2003-08-12
aa2003/08.12z -> 2003-08-12
```

A cserét a Regex.Replace hajtja végre. A $n kifejezésekkel az n. elkapott csoportra hivatkozhatunk, hasonlóan a visszahivatkozások →\n←-jéhez.
A következő példa hyperlinkeket gyűjt ki egy html lapból. A találatokat most alternatív módon érjük el:

```
Regex r;  Match m;

r = new Regex(@"href\s*=\s*""([^""]*)""",
  RegexOptions.IgnoreCase);
for (m = r.Match(ReadTestFile());
  m.Success;
  m = m.NextMatch())
{
  Console.WriteLine("href: {0}, pozíció: {1} ",
    m.Groups[1], m.Groups[1].Index);
}

href: /training/course.aspx?id=2124, pozíció: 2843
href: /training/course.aspx?id=2273, pozíció: 3985
```

Ez utóbbi módszer előnye, hogy a regex egyeztetés lépésenként megy végbe, így a ciklusból idő előtt kilépve a maradék részen nem kell dolgozni a motornak.

Zárszó
Az eddigiek megértése után gyakorlati munkák előtt még érdemes áttekinteni a .NET Framework regexekkel foglalkozó fejezetét **[6]**, ugyanis jóval gazdagabb csoportosítási lehetőségek is vannak még, mint amelyekről a cikkben olvashattak.
Jó regexelést!

Soczó Zsolt
zsolt.soczo@netacademia.net
A szerző a NetAcademia vezető fejlesztési oktatója,
MCSE, MCDBA, MCSD.NET, MCT

A cikkben szereplő URL-ek:
[1] http://www.unicode.org/
[2] http://www.unicode.org/reports/tr8/index.html
[3] http://tinyurl.com/qw7c [4] http://www.sellsbrothers.com/tools/#regexd
[5] http://tinyurl.com/ny0f
[6] http://tinyurl.com/atlv

Kapcsolódó tanfolyamaink:
2524 - XML Webszolgáltatások fejlesztése ASP.NET segítségével 2349/2415 - A .NET keretrendszer programozása C#/VB.NET nyelven