Informatikai Navigátor

Gondolatok a szoftverek használatáról és fejlesztéséről

2009. *év* 1. *szám*

Tartalomjegyzék

Szerkesztette: Nyiri Imre

1. Mi a szabad szoftverek lényege?

Mielőtt elkezdenénk olvasni ezt a cikket, érdemes elgondolkodni a szabad szoftver definicióján.

1. Meghatározás. *A szabad vagy nyílt forráskódú szoftverek (FLOSS) szabadon használható, másolható, terjeszthető, tanulmányozható és módosítható számítógépes programok. Ilyen például a Linux operációs rendszer, a Mozilla Firefox böngésző vagy az OpenOffice.org irodai csomag.*

Az információs társadalom korában élünk, egyre több ember használ rendszeresen különféle programokat. Használjuk az Internetet az információk beszerzéséhez és megosztásához, ügyeink intézéséhez, illetve az egymással való kapcsolattartáshoz. A szoftverek használata át és átszövi a mindennapjainkat, a tudományt, a szórakozást. Van egy másik aspektus is. A programok előállítása nagyon komplex és időigényes feladat. A számítástudomány hasonlóan fejlődik, mint a többi tudomány, például a matematika. Fontos a tudományos igényességű információátadás, az elért eredményekre és tapasztalatokra épülő elméletek megalkotása és azok szabad használhatósága. Képzeljük el például azt a furcsa esetet, hogy a másodfokú egyenlet $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$ képletét fizetős licence védi és minden alkalmazásakor valamekkora összeget kéne fizetnie a használójának. Valószínűleg könnyű egyetérteni azzal a nézettel, hogy egy ilyen szituáció jelentősen lefékezné az emberiség fejlődését. Fontos észrevenni, hogy a mai komplex világban a programok hasonló eszközöket adnak a kezünkbe, nélkülük a hétköznapi feladatok vagy éppen a tudomány problémái nem, vagy csak nehezen lennének kielégítően és megfelelő gyorsasággal megoldhatóak. Alapvető társadalmi kérdés, hogy mi a jobb? A XXI. század eszközeit is „ingyen”[1] használhatóvá szervezni vagy ezért anyagi ellenszolgáltatást kérni?

1.1. A közösségi elven való használat

Az emberiség a javakhoz való hozzáférést eddig is kétféle módon szervezte meg. Van, amikor a közösségi elven való kifizetés a célszerű, a konkrét használatért pedig már nem (vagy csak részben) kell ellenszolgáltatást adni. Az első esetre néhány példa: útak használata (kivéve az autópálya), egészségügy, rendőrség, állami általános iskolai közoktatás, világító torony, játszóterek,:. Ide tartozik a már említett másodfokú egyenlet (és általában a tudományok eredményeinek) szabad használata is. A jószágok nagy része persze a 2. esethez tartozik, azaz annak elfogyasztásáért, azzal összefüggésben, a vásárlás felmerülésekor valamilyen formában fizetnünk kell. Senki nem gondolja azt, hogy az influenza elleni védőoltások tudományos kutatásáért pont akkor kell fizetnie, amikor számára felmerül egy oltás igénye, azaz a kutatás-fejlesztés közösségi alapon szerveződik, általában egy alapítványi vagy közintézményi formában, ami természetesen nem zárja ki a profitcentrikus magán kutatóintézetek működési lehetőségét sem. Ezen a ponton - a témánk szempontjából - azt fontos megérteni és kiemelni, hogy a modern, jól szervezett emberi társadalmakra mindig is jellemző volt, hogy a javak bizonyos fajtáját a közösség együttesen finanszírozta, illetve igény szerint „ingyen” fogyasztotta el (például a világító torony mindenkinek egyformán világít, a rendőr mindenkinek szolgál és az intézkedésért nem kér pénzt).

Az információs társadalomban a szoftver (számítógépes program, adat, dokumentum, használati eljárás gyűjtemény) központi szerepet kap, az állampolgárok nehezen tudnak meglenni a használatuk nélkül. Az állami tisztviselők egyre gyakrabban hivatkoznak arra, hogy a törvényeket, in-

[1] Ennek pontos jelentését később tisztázzuk

formációkat elérhetjük az Internet-en. Manapság majdnem mindenhol az e-mail címünk felől érdeklődnek. Hátrányba kerülünk, ha nem tudjuk használni a számítógépeket, nem vagyunk fenn valamilyen szociális, közösségi portálon vagy az iskolai tanulmányaink során a beadandó feladatok kidolgozásánál vagy a munkahelyünkön nem használjuk a számítógépes lehetőségeket (táblázatkezelés, szövegszerkesztés, ...). Az előzőekből következik, hogy általános igény a szoftverekhez való hozzájutás lehetősége, hasonlóan ahogy a közbiztonságból - a rendőrségen keresztül - is mindenki szeretne részesülni. Ez pedig azt jelenti, hogy egy modern állam feladata, hogy az állampolgárok alapvető szoftvereszközökhöz közösségi elven jussanak hozzá. Erre a legelterjedtebb megoldások az alapítványok (foundation) és egyéb közösségek (community) fenntartása, amiben az állam is szerepet vállal.

Összefoglalva, fel kell ismerünk, hogy a szabad szoftverek az információs társadalmakban szükségszerűen jelennek meg, azok fejlesztését az Internet adta közösségi szervezetek tudják leghatékonyabban megvalósítani. Másfelől a tudomány fejlődése ellentétes az elért eredmények használatának titkolásával, zárt fejlesztő műhelyekben való használatával, az újrahasznosításának folyamatos fizetési kötelezettségével. Szeretnénk kiemelni, hogy a szabad software nem freeware vagy shareware, hiszen azok kódja sokszor zárt, csak a használatuk engedélyezett, így a tudományos fejlődés lehetőségét nem segítik elő. A shareware programok ráadásul rendszerint funkciónalitásukban is korlátozottak.

1.2. A szabad szoftver mozgalom rövid története

A közkincsként (public domain) terjesztett forráskód egyidős a modern programozással. A *szabad szoftver* mozgalmat Richard M. Stallman indította 1983-ban a GNU projekttel. Az elvi célkitűzést a kiáltvány (The GNU Manifesto, 1984, itt elolvasható magyarul is: `http://www.gnu.hu/gnu-kialtvany.html`), egy alapítvány (*Free Software Foundation*, röviden *FSF*, 1985) és az első általános szabad szoftver licenc (*GNU GPL*, 1989) követte. Az *FSF* szerint a szabad szoftverek a következő szabadságjogokkal kell, hogy rendelkezzenek:

1. A tetszőleges célra történő szabad felhasználás;

2. a szabad tanulmányozhatóság és igény szerinti módosíthatóság, aminek előfeltétele a forráskódhoz való hozzáférhetőség;

3. a másolatok szabad terjeszthetősége, segítve ezzel ismerőseinket;

4. a szabad továbbfejleszthetőség, és az eredmény szabad közzététele a közösség javára. Ennek is előfeltétele a forráskód elérhetősége.

A „szabad” nem feltétlenül jelent „ingyenest”: bárki bármennyiért árusíthatja a kérdéses programokat; az egyetlen feltétel, hogy a fenti négy alapjogot garantálja vevői számra. Miért venné meg bárki is? Azért, mert például nem képes azt magának lefordítani, szüksége van kézikönyvre, CD-n vagy DVD-n szeretné a programokat megkapni, vagy mert támogatásra van szüksége. Az is elképzelhető, hogy valaki egyedi fejlesztést, testreszabást, adott hiba kijavítását, adott funkció beépítésével bíz meg egy programozót vagy céget. Az FSF vezeti a fenti elveknek megfelelő licencek listáját. A GNU GPL a legelterjedtebb szabad szoftver licenc. A nyílt forráskódú fejlesztések nagy része pedig az FSF vezetésével készült nyílt forráskódú fejlesztőeszközöket használja mind a mai napig.

Eric S. Raymond és Bruce Perens kezdeményezésére, a szabad szoftverek vállalati körben való népszerűsítésére 1998-ban megalapult az *Open Source* Initiative (*OSI*) közhasznú társaság, amely

a nyílt forráskódot hangsúlyozza az angolban félreérthető „szabad” helyett. (A szabad szoftver angol nevében szereplő free ingyenes jelentéssel is bír, ami tévesen azt sugallhatja, hogy szabad szoftvereket nem lehet eladásra fejleszteni, vagy egyéb profitorientált módon felhasználni.)

A nyílt forráskódú licenceket az OSI véleményezi és tartja nyilván. A nyílt forráskódú licencek között megtalálni a FSF GNU licenceit, de a Microsoft Ms-PL és Ms-RL licenceket is.

1.3. A szabad licence-ek fajtái

A szabad licence kifejezés olyan licencszerződési formákat jelent, melyek biztosítják, hogy a licencelt szellemi termék szabadon felhasználható. A leggyakoribb formája a számítástechnikában használatos szoftverek licencelése. A fogalmat a szerzői jogilag védett, zárt forráskódú, kereskedelmi, védjegyoltalom alá eső licencekkel szemben használjuk, azoktól való megkülönböztetésre.

A „szabad” értelmezése ebben a környezetben fontos: szabad az, amit bárki jelentős korlátozások nélkül:

- felhasználhat, (program esetén például futtathat)
- tanulmányozhat
- működését (forráskódját) megvizsgálhatja
- lemásolhat, módosíthat és azt publikálhatja

A szabad licenceknek három fő irányát különböztetjük meg:

- *Közkincs* – amiknek a szerzői oltalmi ideje lejárt, vagy a készítőjük azt közkincsnek nyilvánította.
- *BSD-jellegű licencek* – melyek az eredeti BSD rendszerről kapták nevüket; lényegük, hogy a szerző megtartja szerzői jogát azon célból, hogy kijelenthesse azt, hogy nem felelős a szoftver életéért, valamint hogy joga legyen nevének feltüntetéséhez; lehetővé teszi a licencelt szoftver akár zárt forráskódú felhasználását is.
- *Copyleft* – ezen licencek legismertebbike a GNU nyilvános licenc. Ezen licencek szintén megőrzik a készítő jogait azon célból, hogy biztosítsák, hogy a software minden változata szabad maradjon. A copyleft kifejezés (jele: ©) egy angol szójáték, a copyright megfordításának eredménye. A copyleft lényege, hogy a jog adta eszközöket nem az adott szellemi termék terjesztésének gátlására, hanem a megkötések kiküszöbölésére használják fel, így garantálva a felhasználás szabadságát a módosított változatokra nézve is.

A licence-ek tárgya általában egy teljes software, egy programozói könyvtár, egy dokumentáció (írásmű szabad használata). Az OSI által jegyzett nyílt elismert szabad licence-ekről itt olvashatunk: `http://hu.wikipedia.org/wiki/Nyílt_forráskódú_licenc`

1.4. A fejlesztés szervezeti és anyagi keretei

Van egy fontos kérdés! Mi az az út, ahogy a szabad szoftverek készülnek? Kezdetben - amikor az Internet még nem létezett a mai formájában - sok egymástól elkülönült, 1 vagy néhány lelkes ember foglalkozott egy-egy téma fejlesztésével. A helyzet azonban már jó évtizede teljesen megváltozott és kialakultak azok a szervezeti keretek, ahol ezeket a szoftvereket fejlesztik. Nem új formációk ezek, alapítványok és közösségek. A működés hasonló, mint például a rák-kutatás. Az alapítványokban komoly munka folyik, természetesen jó fizetésért. Az alapítványoknak bevételeik vannak az államtól (amely - ahogy erre

korábban kitértünk - ezzel kötelességét teljesíti), és azoktól a szoftver cégektől, amelyek túl drágának találják a házon belüli fejlesztést (vagy annak bizonyos részeit), ezért az alapítványok eredményeit vissza akarják építeni saját termékeikbe. Ebbe a folymatba ma már az összes ismert, nagy software cég bekapcsolódott, még a Microsoft és IBM is.

A szoftverek készítésének költségszerkezete is érdekes, ugyanis tudjuk, hogy 1 db termék előállítási költsége a következő képlet alapján alakul: $K = \sum c + \sum v$, ahol a c az állandó, v a változó költségek. Egy autó új darabjának előállítása sok anyagot és munkaerőt igényel, összemérhető annak c rezsiköltségével. A software-ek nem ilyenek, azaz 1 új példány előállítása nem drága (DVD másolás, net-en való terjesztés), lévén a v nagyon alacson szinten van. Vannak olyan elemzések, amik azt mutatják meg, hogy N db software példány esetén az egység költség lényegében az állandó költség szétterhelése az N db példányra: $C_{egység} \sim \frac{c}{N}$, azaz N növelésével a $C_{egység}$egy kis számhoz konvergál, azaz nem teljesíthetetlen az az állami vállalás (ma már komoly elméleti kérdés annak az elemzése, hogy mit várhatunk egy modern államtól, érdemes megnézni ezt a videót: `http://www.mindentudas.hu/bekesilaszlo/index.html`), hogy a széles körben használt szoftverek biztosítva legyenek az állampolgárok részére.

Összefoglalásképpen kiemeljük, hogy itt olyan szerencsés egybeesések alakultak ki, amik hatásmechanizmusában például az ipari vagy a technikai forradalomhoz hasonlóak:

- Az információs társadalom megjelenése - új igényel, állami kötelezettségek jelentek meg
- Internet, ami képes szervezeteket összefogni
- A software-ek előállításának „sajátos” költségszerkezete

Aki ezeket felismeri és alkalmazza a mindennapi döntéseiben, valószínűleg ezen a téren előnyre tehet szert.

1.5. Mikor használjunk szabad szoftvert?

Befejezésként röviden gondolkodjunk el együtt, hogy mikor is használjunk open source-os, szabad software-eket. A jobb megértés érdekében segítségül hívunk 2 fontos állami kötelezettséget: közbiztonság (rendőrség) és egészségügy (orvosi ellátás), hogy a software-ek használatával meglévő analógiák jobban megmutatkozzanak, hiszen ennek társadalmi folyamata még nincs annyira látható állapotban, mint az említett 2 másik funkció. Amikor valaki rendőri intézkedést kér, nem kell fizetnie. Van egy társadalmilag elfogadott szint, amíg az „ellátás” ingyenes, pontosabban fogalmazva, közösségi elven megszervezett. Abban a pillanatban, amikor valaki, valamilyen egyedi, normán felüli szolgáltatást akar igénybe venni (például egésznapos testőri védelem), akkor természetesen fizetni kell érte. Ez az a pont, ami segít megérteni, hogy hol és mikor használjunk szabad szoftvert az információs társadalomban. Itt is kialakult egy „láthatatlan” normatíva, hiszen mindenki tudja, hogy mik azok a software-ek, amiket mindenki közösségi alapon szeretne használni (és nem ellopni!). Ez nagyon hosszú lista, csak néhány eleme:

- operációs rendszer
- böngésző, levelező kliens
- Internetes szolgáltatások elérése
- szövegszerkesztés, táblázatkezelés, prezentáció készítés
- csoportmunka támogatés

- webszerverek
- fejlesztőeszközök
- közhasznú dokumentumok és programozói könyvtárak (például statisztikai programcsomag)
- ...

Talán már az olvasó is kitalálta, hogy mikor kell fizetős szoftvereket (manapság inkább szolgáltatásokat) vásárolni:

- amikor egyedi igényeket akarunk kielégíteni (analógia: egésznapos testőr)
- emelt szintű support vagy dokumentációs szolgáltatás
- amikor valami olyan új értéket kell létrehozni (részben vagy egészben), ami addig nem volt

Manapság még nagyon sok szoftverért úgy fizetünk, mintha az a fenti 3 kategória valamelyikébe tartozna. Az okok természetesen ismertek, csak néhány ezek közül:

- a „dobozos" termék valamely hozzáadott értéke (tapasztalatom szerint hosszabb távon ezek elvett értékként szoktak inkább jelentkezni)
- árukapcsolás. A fizetős termék csak saját környezetben működik
- a vevő megtévesztése

Érdekes tapasztalat, hogy az ismert, nyílt softwareek jó minőségűek is. Több eset van, amikor egy zárt programot átadnak valamely alapítványnak és nyílt program lesz belőle. Ez a folyamat sosem olyan triviális, mint azt sokan gondolnánk. Az inkubációs szakasz (zárt → nyílt átalakítás) során rengeteg hibajavítás szokott történni.

1.6. Ajánlott szakirodalom

Az Internet-en - lévén az a szabad szoftver és az open source természetes közege, ami maga is ezt a szellemiséget követi - sok érdekes írás található ezen cikkel kapcsolatosan. Az alábbiak elsősorban magyar nyelvre fordított írásokat tartalmaznak, amiket érdemes elolvasnia annak, aki jobban el szeretne mélyedni a témában.

- A szabad szoftver
  `http://hu.wikipedia.org/wiki/Szabad_szoftver`
- A GNU–kiáltvány
  `http://www.gnu.hu/gnu-kialtvany.html`
- A GNU General Public License (GPL)
  `http://www.gnu.hu/gpl.html`
  `http://www.gnu.hu/gplv3.html`
- GNU Lesser General Public License (LGPL)
  `http://www.gnu.hu/lgpl.html`
  `http://www.gnu.hu/lgplv3.html`
  Ez a licence a programozói könyvtárakra vonatkozik.
- GNU Free Documentation License (FDL)
  `http://www.gnu.hu/fdl.html`
- A katedrális és a bazár
  Aki az Open Source lényegét jobban ismerni akarja feltétlenül olvassa el. Az élet többi területén is nagyon tanulságos írás.
  `http://magyar-irodalom.elte.hu/robert/szovegek/bazar/`
- Szabad kultúra
  `http://www.szabadkultura.hu/`
- Wikinómia
  `http://www.euroastra.hu/node/7891`

2. Szabályos kifejezések

Ennek a cikknek az elsődleges célkitűzése a reguláris kifejezések (szabályos kifejezések) bemutatása. Ehhez a C++ nyelvet és a Qt könyvtár reguláris kifejezéseket kezelő képességeit fogjuk használni. A Qt könyvtár egy hatékony szabályos kifejezéseket kezelő motorral rendelkezik, melynek működési modelljét a perl nyelv tudása alapján tervezték meg. Ebben a cikkben bemutatjuk, hogy mi is az a szabályos kifejezés, illetve néhány C++ példaprogrammal megpróbáljuk egyszerűen érzékeltetni, hogy miért is érdemes ezt a kiváló eszközt használni. Már itt az elején szeretnénk felhívni a figyelmet arra, hogy a szabályos kifejezés (reguláris kifejezés) és a minta (pattern) nem ugyanazt a fogalmat takarja, de ennek ellenére a mintát is sokszor szabályos kifejezésnek szokták nevezni. Ez a fogalomkeveredés azért van, mert a minta egy olyan szabályos kifejezés, amit különféle módosító állításokkal (assertions) látunk el.

2.1. Mi is az a szabályos kifejezés?

A formális nyelvek elmélete szerint egy nyelv a megengedett karaktersorozatok halmaza. Nézzük meg ezt pontosabban is! Legyen $\sum$egy jelekből (karakterekből) álló halmaz, amit ABC-nek nevezünk. Ilyen jelkészlet lehet például az UTF8, UNICODE, LATIN2, ... halmazok, de akár a szolmizációs hangok is. Legyen egy tetszőleges $\sum = \{b_1, b_2, \ldots, b_n\}$ÁBC-énk! A $\sum$ ABC jeleiből konstruálható véges vagy megszámlálhatóan végtelen sorozatok halmazát jelöljük $\sum^*$-gal, mely halmazba beleértjük az egyetlen, nulla hosszúságú sorozatot is. Nyelvnek nevezzük a $\sum^*$ halmaz egy tetszőleges részhalmazát.

Legyen a $\sum$=LATIN2 karakterhalmaz, ekkor az $\mathscr{L} = \{ab, abb, acc\} \subseteq LATIN2^*$, azaz $\mathscr{L}$ a LATIN2 ABC felett értelmezett nyelv, aminek elemeit mondatoknak nevezzük. A kérdés ezekután persze az, hogyan is tudunk egy nyelvet jelentő halmazt meghatározni. Szerencsés esetben fel is sorolhatjuk a karaktersorozatokat, ahogy azt a fenti példában tettük. Általában azonban különféle szabályrendszereket adunk meg, amik pontosan meghatározzák a nyelvet jelentő halmazt. Ilyen szabályrendszert képes megadni például a szabályos kifejezés, amit ennek tiszteletére reguláris grammatikával megadott nyelvnek is nevezünk. Nézzük meg, hogy ebben a nyelvtanban milyen karaktersorozat generáló szabályok vannak!

1. Szabály. *Egy egyedülálló karakter, amely nem újsor vagy* `.*[\]^$` *karakterek egyike, az önmagát, mint 1 elemű nyelvhalmazt hozza létre. Példa: a p karakter, mint reguláris kifejezés a* $\{p\}$ *nyelvet jelenti. A kivételezett karakterek megadása escape szekvenciákkal lehetséges:* `\., \[, \\, \], \^, \$`

2. Szabály. *A* [...] *jelek között felsorolt karakterek egy annyi elemű (szavú) nyelvet hoznak létre, amennyi különböző betű a felsorolásban van. Példa: Az* $[sfgz8]$*, mint reguláris kifejezés a következő halmazt (azaz nyelvet) jelenti:* $\{s, f, g, z, 8\}$*. Vannak gyakran használt, emiatt előre megadott szimbólumok, melyek a következőket jelentik:*

- *A . bármely (az újsor kivételével) ABC-beli karaktert jelenti.*
- *A \w az egy szóban előforduló karakterek.*
- *A \W minden, ami nem \w.*
- *A \d a számjegyek, azaz egyenértékű a [0123456789] megadással.*
- *A \D minden, ami nem \d.*
- *A \s a szóközjellegű karakterek (szóköz, tabulátor, újsor).*
- *A \S minden, ami nem \s.*
- *Használhatóak a C/C++ nyelv escape szekvenciái is (Példa: \t, \n)*

Példa: A $\backslash d$ reguláris kifejezés a $\{0,1,2,3,4,5,6,7,8,9\}$ nyelvet definiálja. Ezen nyelv az 1 hosszú, számjegyekből álló mondatok nyelve.

A [...] között megadott karakterosztály tagadását [^...] jelöli. Példa: [^0123456789] a nem számjegyeket jelenti. A fenti 1. és 2. szabályok az elemi szabályos kifejezések (ezek 1 hosszú sorozatokat tartalmazó nyelvet tudnak megadni) megadását teszik lehetővé. A további 3 szabály azt mondja meg, hogy az ilyen elemi szabályos kifejezésekből mi módon lehet bonyolultabb kifejezéseket építeni.

3. Szabály. *Legyen E_1 és E_2 egy-egy szabályos kifejezés, amelyek az $\mathscr{L}_1$ és $\mathscr{L}_2$ nyelveket hozzák létre. Ekkor az E_1E_2 egymás mellé írás egy új reguláris kifejezést jelent, ami egy olyan $\mathscr{L}$ nyelvet ad meg, amit $\mathscr{L}_1 \times \mathscr{L}_2$ descartes szorzat eredményeképpen adott sorozatpárok összeragasztásával kapott sorozatok alkotnak.*

Példa: Legyen $E_1 = [abc]$ és $E_2 = [xyz]$. Ekkor $\mathscr{L}_1 = \{a,b,c\}$ és $\mathscr{L}_2 = \{x,y,z\}$. Az egymás után írás során kapott reguláris kifejezés: $[abc][xyz]$. A kapott $\mathscr{L}$ nyelv pedig $\{ax, ay, az, bx, by, bz, cx, cy, cz\}$ halmaz lesz, azaz a párok össze lettek ragasztva.

4. Szabály. *A választás szabálya. Legyen E_1 és E_2 egy-egy szabályos kifejezés, amelyek az $\mathscr{L}_1$ és $\mathscr{L}_2$ nyelveket hozzák létre. Ekkor a választás művelete (jele: pipeline karakter) az $E_1|E_2$ reguláris kifejezést hozza létre, amely az $(\mathscr{L}_1 \cup \mathscr{L}_2)$ nyelvet generálja.*

Példa: Ha E_1 az $\mathscr{L}_1 = \{af, zg\}$ és E_2 az $\mathscr{L}_2 = \{aaa, drt, dfg\}$ nyelvet alkotja meg, akkor $E_1|E_2$ az $\mathscr{L} = \{af, zg, aaa, drt, dfg\}$ nyelvet generálja.

5. Szabály. *Sokszorozók (Quantifiers). Egy E szabályos kifejezésből az E valamennyi egymás után írásával (3. szabály) újabb reguláris kifejezéseket kapunk, amely konstrukciót sokszorozásnak nevezzük.*

Nézzük meg őket egyenként!

- Az $E\{m\}$ jelentése $\overbrace{EEE\ldots E}^{m}$, azaz m darab egymás után írást jelent, amit a 3. szabály alapján kell képezni.
- $E\{m,n\}$ jelentése: $(E\{m\}|E\{m+1\}|...|E\{n-1\}|E\{n\})$, azaz a 3. és 4. szabályokat használtuk. Példa: Legyen $E = ak$ szabályos kifejezés, ami természetesen az 1 szavas $\{ak\}$ nyelvhalmazt generálja. Ekkor az $(ak)\{2,4\}$ jelentése hosszabban írva: $akak|akakak|akakakak$ szabályos kifejezés, ami triviálisan egy 3 elemű (szavú) nyelvhalmazt generál: $\{akak, akakak, akakak\}$.
- $E?$ jelentése: $E\{0,1\}$. Az $E\{0\}$ az üres sorozatot jelenti. Példa: Legyen a (az 'a' betű) most egy reguláris kifejezés. Ekkor az $a?$ jelentése a$\{0,1\}$, ami a következő halmazt generálja: {üres_sorozat, a}, azaz az a betű 0 vagy 1 előfordulását jelenti.
- E^*jelentése: $E\{0,\infty\}$, azaz az E reguláris kifejezés nulla vagy többszöri egymás után írásából kapott reguláris kifejezésekből való választás. Példa: Egy ab reguláris kefejezés esetén az $(ab)^*$ szabályos kifejezés a $\{üres_sorozat, ab, abab, ababab, abababab, \ldots\}$ halmazt generálja. Figyeljük meg, hogy a zárójelekkel a sokszorozók hatályát lehet befolyásolni. Az ab^* kifejezés például az $\{a, ab, abb, abbb, \ldots\}$ nyelvhalmazt generálná. Vegyük észre, hogy az ab^* egyébként az a és b reguláris kifejezésekből konstruált szabályos kifejezés, ahol a konstruálás lépései az eddigi szabályok alapján történtek.
- $E+$jelentése: $E\{1,\infty\}$. Példa: $xy+z$ halmaza: $\{xyz, xyyz, xyyyz, ...\}$

$E\{m,\}$jelentése: $E\{m,\infty\}$. $E\{,n\}$ jelentése: $E\{0,n\}$.

Összefoglalásaként megállapítható, hogy a fenti 5 szabály tetszőleges számú és sorrendű használatával építhetjük fel az összetett reguláris kifejezéseket, amik mindig egy-egy nyelvet (az ABC-énk jelsorozataiból álló halmazt) határoznak meg.

2.2. Minta és mintaillesztés

A minta egy olyan reguláris kifejezés, amit opcionálisan néhány további jellel láthatunk el. Ezeket a jelzéseket állításoknak (*assertions*) nevezzük. A formális nyelvek elméletében az egyik alapfeladat annak az eldöntése, hogy egy mondat (karaktersorozat) eleme-e egy nyelvnek, amely nyelvet például egy reguláris nyelvtannal is megadhatunk. A kérdés tehát az, hogy a nyelvtan által generált $\mathscr{L}$ nyelvhalmaz tartalmazza-e a mi mondatunkat.

Példa: Legyen a nyelvtan a következő szabályos kifejezés: *a(b{1,2}|c{2})* A valakitől kapott mondat pedig a következő: *abb*. Kérdés: A mi reguláris kifejezésünk le tudja-e generálni ezt a mondatot, azaz a neki megfelelő $\mathscr{L}$ nyelvnek eleme-e az *abb* mondat? Az $\mathscr{L}$ nyelv esetünkben nyilvánvalóan a következő lesz: $\{ab, abb, acc\}$, azaz tartalmazza a mondatot. Ez azt jelenti más szavakkal, hogy a fenti reguláris kifejezés illeszkedik a mondatra.

A UNIX, Java, Perl, Qt és egyéb eszközök az illeszkedést tágabb értelemben határozzák meg, azaz elégséges az, hogy a mondat egy részsztringjét elő lehet állítani a mintával. Azt az esetet, amikor továbbra is azt szeretnénk, hogy a teljes mondatra illeszkedjen a reguláris kifejezés, a következő állítással adjuk meg a mintában: *^E$*.Esetünkben a minta így alakul: *^a(b{1,2}|c{2})$*. A ^ jel a mondat elejére, míg a $ a végére való illeszkedést követeli meg. Ezzel máris megismertünk 2 olyan jelet, amivel ki lehet egészíteni egy szabályos kifejezést. Ezen a ponton azt is könnyedén megérthetjük, hogy ezek szerint minden szabályos kifejezés egyben minta is, de fordítva ez már nem igaz.

Példa: Az *abbxc* mondat nem illeszkedik fenti mintánkra, mert nincs ilyen mondata a létrehozott nyelvnek, de a *^a(b{1,2}|c{2})* minta már igen, mert az nem követeli meg a teljes mondatra illeszkedést, csak azt, hogy az elejére illeszkedjen.

Ha megnézzük az *^a(b{1,2}|c{2})* mintával létrehozott *{ab, abb, acc}* nyelvünket, akkor észrevehetjük, hogy a mi *abbxc* fűzérünk *ab* és *abb* szakasza is eleme ennek a halmaznak. Emiatt felvetődik a kérdés, hogy a string mely szakaszára mondjuk azt, hogy illeszkedik a mintára. A mintaillesztés megköveteli, hogy az illeszkedés egyértelmű legyen, de a fentihez hasonló többértelműség igen gyakori szokott lenni. Mi a megoldás? Be szokták vezetni a *mintaillesztés 2 üzemmódját*. Az egyik a falánk, a másik a lusta üzemmód. A falánk üzemmódban mindig a leghosszabban illeszkedő, míg lusta üzemmódban a legrövidebben illeszkedő nyelvhalmazbeli mondatot kell tekinteni. Általában a falánk üzemmód szokott az alapértelmezés lenni, mely esetünkben az *abbxc* fűzérben az *^a(b{1,2}|c{2})* minta a fűzér *abb* részére illeszkedik, míg lusta üzemmódban pedig az *ab* szakaszra.

Fontos információ, hogy egy minta mely poziciótól kezdve és milyen hosszan illeszkedik egy sztringre. Példa: Az *a(b{1,2}|c{2})* minta a *zzzabbzzz* fűzérre a 3. pozición illeszkedik 3 hosszan *(abb)*, ha a falánk üzemmód aktív.

A ^ és $ jelzésen kívül léteznek további megengedett, illesztési pozicióra vonatkozó állítások is. Ilyen a $\backslash b$, ami annyit követel meg, hogy a vizsgálandó fűzérben szóhatárra történjen az illeszkedés. Legyen most ilyen a mintánk: *\ba(b{1,2}|c{2}*,ami már nem illeszkedik a *zzzabbzzz* mondatra, mert az *abb* részfűzér nem szóhatáron kezdődik. Illeszkedik viszont a *zzz abbzzz* mondatra, amire természetesen a kiinduló mintánk is illeszkedik. A $\backslash B$ állítás azt az igényt fogalmazza meg, hogy ne szóhatáron történjen az illeszkedés.

Létezik két érdekes, az illeszkedés feltételét

megadó, illetve korlátozó állítás. Az egyiket pozitív előretekintésnek (*positive lookahead, jele: (?=...)*), a másikat negatív előretekintésnek (*negative lookahead, jele: (?!...)*) nevezzük. Itt nem csupán pozicionálisan korlátozzuk, hogy a reguláris kifejezésünk a string mely részeire illeszkedhet, hanem azt, hogy mely karakterek következhetnek, vagy nem következhetnek utána. A pozitív előretekintés megköveteli az illeszkedésnél azt is, hogy utána a *(?= ...)* között megadott karakterek jöjjenek. Példa: Legyen a minta ez: *aaa(?=bb)*. Kérdés: Illeszkedik a minta a *aaaxxaaabbcaaa* mondatra? Igen, mert az 5. pozición az *aaa* minta illeszkedik úgy, hogy utána *bb* következik. Ebben az esetben az illeszkedés hossza természetesen 3, hiszen a mi reguláris kifejezésünk csak *aaa*, ami a {*aaa*} nyelvet generálja. A minta *(?=bb)* része (ami egy másik reguláris kifejezés) csak az illeszkedést előíró további feltételt adja meg.

A negatív előretekintés ennek az ellenkezője. Például az *aaa(?!bb)* minta csak akkor illeszkedik egy stringre, ha az illeszkedés olyan helyen van, amit nem *bb* követ.

A minták megadásánál az un. referenciákkal lehet egy-egy hivatkozást eltárolni egy-egy már illesztett szakaszra. Egy *(...)* zárójelezett rész nemcsak az 1-5 szabályokból építhető eredő reguláris kifejezést csoportosítja, de egy referenciaszámot is létrehoz, amit persze nem kötelező használni, de sokszor jól jöhet Egy minta zárójelezései során mindig a nyitó zárójelek jelentik egy referencia kezdetét és sorszámát. Példa: A *aaa(a.)c((.)d)* mintában 3 refrencia van a nyitózárójelek sorrendjében, melyek nevei: *\1, \2, \3*.

Példa: A *a(.)a\1* mintára *abab* illeszkedik, de az *abax* nem, ugyanis a referencia hivatkozás megjegyzi az illesztett mintarészt (jelen esetben a *b*) és ez lesz a *\1* hivatkozásnál használva. Látható tehát, hogy a referencia nem a . karaktert, mint joker karaktert jegyzi meg, hanem a ténylegesen illesztett mintát. Ez azért jó, mert például egy előre nem ismert karaktersorozatra lehet hivatkozni. Legyen egy mintánk ez: *a(.*)b\1*. Ekkor a *avalamibvalamixxx* mondathoz illeszkedik a minta és a *\1* hivatkozás most éppen a valami karaktersorozatot jelöli. Az *axxxbxxxvalami* mondatban pedig a *\1* az *xxx*-et jelenti. Ha csak a szabályos kifejezés részkifejezésekből való helyes felépítés a célunk és nem akarjuk hogy referencia jöjjön létre, akkor a csoportosításra *(?:...)* zárójelezést kell használni.

A jobb megértés és a gyakorlás érdekében nézzünk meg néhány hasznos mintát!

Példa: Egy e-mail helyes szintaxisának vizsgálatát megvalósító minta: *^.+@\w+\..+$* A *^...$* között van a minta, ami azt jelenti, hogy a teljes vizsgálandó fűzérre illeszkedni kell a mintának, azaz azt le kell generálnia. Valahány karakter után egy @-nak kell jönnie, majd a domain név szabályai miatt 1 vagy több szókarakternek, amit egy pontnak (ezt jelöli a *\.*) kell befejeznie, majd tetszőleges karaktersorozat zárja az illesztendő fűzért.

Példa: Készítsünk olyan mintát, ami illeszkedik arra a sztringre, aminek az első 4 betűje tükörszimmetrikus az utolsó 4 betűre. Megoldás: *^(.)(.)(.)(.).*\4\3\2\1$*. Ez illeszkedik például az *almaxxxxxamla* mondatra.

2.3. Mire használhatóak a szabályos kifejezések?

A reguláris kifejezésekben az a szép, hogy olyan dolgokra is használhatjuk, amikre nem is gondolnánk. A sztringek feldolgozása egy nyílvánvaló példa, hiszen egy részsztring keresése, cseréje alapfeladat. A sztringek sorokra vágása, illetve a sorok összeillesztése is visszatérő probléma egy szövegkezelő program számára. Érdekes kérdés egy sztring érvényesítése (validálása). El kell döntenünk ilyenkor olyan kérdéseket, hogy egy fűzér elfogadható e-mail, rendszám, telefonszám, ... formátumú-e. A Qt GUI egy érdekes célra, nevezetesen a beviteli mező maszkolásá-

ra és érvényesítésére is felhasználja a szabályos kifejezéseket.

2.4. Egy karakter alapú tesztprogram

Kezdjük el használatba venni a Qt könyvtárat és írjunk egy olyan parancssori programot, ami átveszi paraméterül a mintát és a vizsgálandó szöveget, majd kiirja a képernyőre az illeszkedési poziciót. Amennyiben nincs illeszkedés, úgy a -1-et írja ki. A forráskódot az 1.algoritmus tartalmazza. Kezdjük el használatba venni a Qt könyvtárat és írjunk egy olyan parancssori programot, ami átveszi paraméterül a mintát és a vizsgálandó szöveget, majd kiirja a képernyőre az illeszkedési poziciót.

A 12. sorban láthatjuk, hogy a szabályos kifejezés fogalmát a *QRegExp* osztály valósítja meg. A 23. sorban meghívott *setPattern()* metódus a parancsorból átvett mintát állítja be, azaz a *rex* objektum számára ez lesz most a beállított minta. A 24. sor *rex.search(fuzer)* hívása a *fuzer* sztringre illeszti a *rex*-ben beállított mintát. A program egy lehetséges futása:

```
# ./teszt0.exe a+ xxxaaaxxx
Eredmény: 3
```

Itt egy fontos dolgot kell észrevennünk. Adjuk ki ezt a parancsot:

```
# ./teszt0.exe (ax)+ xxxaaaxxx
```

bash: syntax error near unexpected token ‘ax’. Miért van hiba? Azért, mert a shell (bash) által vannak speciálisan értelmezett karakterek, ilyen például a ’*(*’ és ’*)*’ jelek is. A mi szándékunk az, hogy ezeket a jeleket a shell most ne értelmezze, azaz a parancs speciális karaktereit egy backslash karakterrel védenünk kell. Nézzük meg így is a parancsot!

```
# ./teszt0.exe \(ax\)+ xxxaaaxxx
Eredmény: 5
```

Helyreálltak a dolgok, de mi ismét tanultunk valamit. Amennyiben a UNIX-ban speciális jelentésű karaktert akarunk használni a grep, awk, perl, ... eszközökben, akkor szükség esetén az escape karaktereket is használni kell, valamint világosan értenünk kell, hogy ekkor a ’\’ karakterek nem a minta részei, kivéve a “\\”-t, ami a ’\’ karaktert jelöl.

Az utolsó tisztázandó kérdés a teszt0.cpp program fordítása, amihez az 1.ábra által mutatott *Makefile* használható.

2.5. Egy egyszerű GUI alapú tesztprogram

Készítsünk egy GUI programot, aminek 2 beviteli mezője van. Egy a minta, egy pedig a vizsgált szöveg számára. Egy nyomógomb hatására írja ki a program, hogy melyik pozición van illeszkedés, illetve ezt a szakaszt jelölje is ki az elemzett szövegben. A program előnye, hogy úgy írhatjuk be és tanulmányozhatjuk a mintákat, hogy nem kell tartanunk a shell speciális karaktereitől sem. A program 3 forrásfile-ból áll: *rexgui.h*, *rexgui.cpp* és *main.cpp*. A program egy futási képét mutatja az 2.ábra. A program 3 forrásfile-ból áll: rexgui.h, rexgui.cpp és main.cpp. A rexgui.h tartalmazza a program egyetlen formját leíró TMyForm osztály specifikációját.

A 17. sor Q_OBJECT makrója arra utal, hogy olyan elemeket is használunk a forráskódban, amit a Qt rendszer moc (Meta Object Compiler) nevű makrófeldolgozója fog C++ kóddá alakítani. A Qt – eltérően például a Borland C++ Builder-től - nem terjeszti ki a C++ nyelvet és nem vezet be property és hasonló kulcsszavakat, ellenben rendelkezik egy fejlett preprocessorral. Ezzel két dolog is elegánsan megoldó-

Algorithm 1 Egyszerű mintaillesztés

```
1  //
2  // teszt0.cpp
3  //
4  #include <qapplication.h>
5  #include <qregexp.h>
6  #include <iostream>
7
8  using std::cout;
9
10 int main( int argc, char ** argv )
11 {
12     QRegExp rex;
13
14     if (argc != 3)
15     {
16       cout << "\nHelyes használat: teszt0.exe minta fuzer\n\n";
17       return -1;
18     }
19
20     QString minta="", fuzer="";
21     minta = argv[1];
22     fuzer = argv[2];
23     rex.setPattern( minta );
24     cout <<  "\nEredmény: " << rex.search( fuzer ) <<    "\n";
25     return 0;
26 }
```

1. ábra. Fordítás és linkelés

```
CXFLAGS = -I/usr/lib/qt3/include -L/usr/lib/qt3/lib LIBS = -lqt-mt
teszt0.exe : teszt0.cpp g++ $(CXFLAGS) -o teszt0.exe $(LIBS) teszt0.cpp
```

dik. Egyrészt a GUI program igényelte magasabb szintű forrásprogram egységek könnyebben kezelhetővé válnak, másrészt ez teremti meg a platformfüggetlen programok írássának a lehetőségét. A TMyForm osztályban a Delphi/C++ Builder-hez hasonlóan ágyazódnak be a vezérlő objektumok (soreditor, cimke, nyomógomb). Látható az is, hogy használható lesz az automatikus elrendezése a vezérlőknek (grid). Egy tipikus Qt makró a public slots:, ami lényegében az eseménykezelést valósítja meg. Ebben a programban csak a kiertekel() metódus kezel eseményt, ami egyben a főform egyik metódusaként jelenik meg.

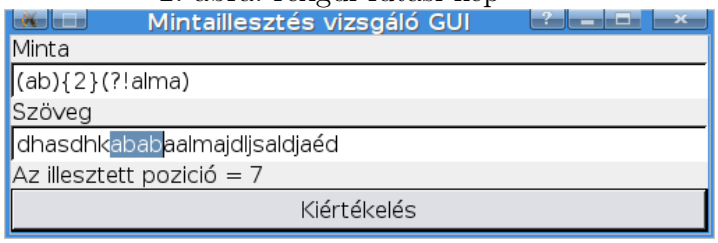
A 40-50 sorig lévő konstruktor részlet létrehozza a vezérlő objektumokat, majd az 52-57 sorok mondják meg, hogy ezeket a GUI elemeket hogyan akarjuk automatikusan elhelyezni a formon. Az 59. sor *connect()* függvénye a nyomógombunk által kibocsátott *clicked()* signal-t kapcsolja össze a mi form objektumunk *kiertekel()* slot-jával. Ettől kezdve minden gombnyomásra lefut a *kiertekel()* metódus. A 63-75 sorokban lévő *kiertekel()* metódus létrehoz egy már ismert *QRegExp* objektumot, a 66. sor lekéri a minta nevű vezérlő szövegét és beállítja azt a reguláris kifejezés mintájaként. A 67. sorban egy mintaillesztés történik a *szoveg* nevű vezérlő (widget) tartalma alapján. A poz és hossz változók tartalmazzák az illesztés kezdetét (ennek értéke a 74. sorban kerül kiirásra) és hosszát. A 70. sor a vizsgált szöveg mintához illeszkedő részét kijelőli, hogy azt a program használója könnyebben tanulmányozhassa.

Végezetül nézzük meg a *main.cpp* főprogramot is! A 87. sorban létrejön az *f* form mutató, ami a 89. sor tanúsága szerint program főformjára mutat.

A fenti program lefordításához nem kell saját *Makefile*-t irni, mindössze egy egyszerű project file-t készítettünk rexgui.pro néven a következő tartalommal:

Ezután a fordítás a következő 2 lépésből áll:

2. ábra. rexgui futási kép



- *$QTDIR/bin/qmake*
- *make*

A QTDIR környezeti változó a Qt rendszer gyökérkönyvtárára mutat. A *qmake* utility automatikusan létrehozza a szükséges Makefile-t a *rexgui.pro* file alapján.

```
1  //
2  // rexgui.h
3  //
4  #ifndef REXGUI_H
5  #define REXGUI_H
6
7  #include <qapplication.h>
8  #include <qdialog.h>
9  #include <qlabel.h>
10 #include <qlineedit.h>
11 #include <qpushbutton.h>
12 #include <qlayout.h>
13 #include <qregexp.h>
14
15 class TMyForm : public QDialog
16 {
17    Q_OBJECT
18    public:
19          TMyForm();
20          QLineEdit    *minta;
21          QLineEdit    *szoveg;
22          QLabel       *valasz;
23          QLabel       *txt1;
24          QLabel       *txt2;
25          QPushButton *btnOK;
26          QGridLayout *grid;
27
28    public slots:
29          void kiertekel();
30 };
31
32 #endif
33
34 //
35 // rexgui.cpp
36 //
37
38 #include "rexgui.h"
39
40 TMyForm::TMyForm()
41 {
42    resize(500, 100);
43    setCaption("Mintaillesztés vizsgáló GUI");
```

```
44    minta     = new QLineEdit(this);
45    szoveg    = new QLineEdit(this);
46    valasz    = new QLabel("Pozició: ", this);
47    txt1      = new QLabel("Minta", this);
48    txt2      = new QLabel("Szöveg", this);
49    btnOK     = new QPushButton("Kiértékelés", this);
50    grid      = new QGridLayout(this);
51
52    grid->addWidget( txt1,      0, 0 );
53    grid->addWidget( minta,     1, 0 );
54    grid->addWidget( txt2,      2, 0 );
55    grid->addWidget( szoveg,    3, 0 );
56    grid->addWidget( valasz,    4, 0 );
57    grid->addWidget( btnOK,     5, 0 );
58
59    connect(btnOK, SIGNAL(clicked()), this, SLOT(kiertekel()));
60
61 }
62
63 void TMyForm::kiertekel()
64 {
65   QRegExp rex;
66    rex.setPattern( minta->text() );
67    int poz   = rex.search( szoveg->text() );
68    int hossz = rex.matchedLength();
69
70    szoveg->setSelection(poz, hossz);
71
72    QString vs;
73    vs = QString("Az illesztett pozició = %1").arg(poz);
74    valasz->setText( vs );
75 }
76
77 //
78 // main.cpp
79 //
80
81 #include "rexgui.h"
82
83 int main(int argc, char **argv)
84 {
85     QApplication app(argc, argv, true);
86
```

```
87      TMyForm *f = new TMyForm();
88      f->show();
89      app.setMainWidget(f);
90
91      return app.exec();
92  }
```

```
1  #rexgui.pro
2  TEMPLATE        = app
3  CONFIG          += qt warn_on release
4  HEADERS         = rexgui.h
5  SOURCES         = rexgui.cpp \
6                    main.cpp
7  TARGET          = rexgui.exe
```

Algorithm 2 Rész string cseréje egy stringben

```
1  //
2  // teszt1.cpp
3  //
4  // Egy sztringben lévő résztring cseréje
5
6  #include <qapplication.h>
7  #include <qregexp.h>
8
9  #include <iostream>
10
11 using std::cout;
12
13 int main( int argc, char ** argv )
14 {
15     QString str = "Egy ember jött álmomban házamba.";
16
17     // Az ember 5 hosszú és a 4.pozición kezdődik
18     // Ezt cseréljük a vendég szóra
19     str.replace(4, 5, "vendég");
20     cout << "\n" << str << "\n";
21
22     QRegExp rex("á(?=l)");
23     // (globális működés!)
24     str.replace(rex, "...a'...");
25     cout << "\n" << str << "\n";
26
27     // Az illesztett részt cseréljük vissza á-ra
28     rex.setPattern("...a'...");
29     int poz = rex.search(str);
30     int illesztett_hossz = rex.matchedLength();
31     str.replace(poz, illesztett_hossz, "á");
32     cout << "\n" << str << "\n";
33
34     return 0;
35 }
```

Algorithm 3 Split és Join teszt

```
1  //
2  // teszt2.cpp
3  //
4  // split és join teszt
5
6  #include <qapplication.h>
7  #include <qregexp.h>
8  #include <iostream>
9
10 using std::cout;
11
12 int main( int argc, char ** argv )
13 {
14     QString txt = "alma italxxxkörtexxxxxxszilva és ringlóxxxxbarack";
15     cout <<  "\n" << txt << "\n";
16
17     QRegExp rex("x{2,}");
18
19     QStringList sl = QStringList::split(rex, txt );
20
21     for (QStringList::Iterator it=sl.begin(); it!=sl.end(); ++it)
22     {
23       cout <<  "\n" << *it;
24     }
25     cout <<  "\n";
26
27     // És egységesen a határoló a ; lesz...
28     txt = sl.join(";");
29     cout <<  "\n" << txt << "\n";
30
31     return 0;
32 }
```

Algorithm 4 A zárójelezés

```
1  //
2  // teszt3.cpp
3  //
4  // A zárójelezés
5
6  #include <qapplication.h>
7  #include <qregexp.h>
8  #include <iostream>
9
10 using std::cout;
11
12 int main( int argc, char ** argv )
13 {
14     QString sz = "Petőfi Sándor;1823;költő";
15
16     QRegExp rex("^([^;]+);([^;]+);([^;]+)$");
17     if ( (rex.search( sz )) != -1 )
18     {
19       cout << "\nNév: " << rex.cap(1);
20       cout << " Született: " << rex.cap(2);
21       cout << " Foglalkozás: " << rex.cap(3) << "\n" ;
22     }
23     else { cout << "\nNincs értelmezve a szöveg!\n"; }
24
25     return 0;
26 }
```

Algorithm 5 E-mail cím szintaxis ellenőrző

```
1  //
2  // joemail.cpp
3  //
4
5  #include <qapplication.h>
6  #include <qregexp.h>
7
8  #include <iostream>
9
10 using std::cout;
11
12 int main( int argc, char ** argv )
13 {
14     if (argc != 2 ) // A program + 1 paraméter
15     {
16        cout << "\n Pontosan 1 paraméter megadása szükséges!\n";
17        return -1;
18     }
19
20     // e-mail szintaxis
21     QRegExp rex( "^.+@\\w+\\..+$" );
22
23     //QString str = argv[1];
24
25     cout <<  rex.search( argv[1] ) <<   "\n";
26
27     return 0;
28 }
```

Algorithm 6 Egy soreditor beviteli maszk

```
1  //
2  // maszkteszt.cpp
3  //
4  #include <qapplication.h>
5  #include <qdialog.h>
6  #include <qlabel.h>
7  #include <qlineedit.h>
8  #include <qlayout.h>
9  #include <qregexp.h>
10 #include <qvalidator.h>
11
12 QRegExp rex( "[abc]{2}\\d+" );
13 QRegExpValidator validator(rex, 0);
14
15 ////////////////////////////////////
16 class TMyForm : public QDialog
17 {
18     public:
19           TMyForm();
20
21           QLineEdit *lNev;
22           QGridLayout *grid;
23 };
24
25 TMyForm::TMyForm()
26 {
27   lNev    = new QLineEdit(this);
28   resize(500, 100);
29   lNev->setValidator(&validator);
30   grid = new QGridLayout(this);
31   grid->addWidget( lNev,    0, 0 );
32 }
33
34 int main(int argc, char **argv)
35 {
36     QApplication app(argc, argv, true);
37
38     TMyForm *f = new TMyForm();
39     f->show();
40     app.setMainWidget(f);
41     return app.exec();
42 }
```

2.6. A Qt könyvtár szabályos kifejezéseket kezelő osztályai

Ennyi előzetes után nézzük meg egy kicsit alaposabban azokat a Qt osztályokat, ami a különféle szbályos kifejezéseket használó feladatok megoldása során gyakran előfordulnak: *QRegExp*, *QRegExpValidator*, *QString*, *QStringList*.

A *QRegExp* a reguláris kifejezés fogalmát valósítja meg, részletes leírását a Qt dokumentációban nézzük meg. Kiemeljük a már ismert *search()* metódust, ami maga a mintaillesztés és az illeszkedés pozicióját adja vissza. A *setMinimal(bool)* metódus a falánk és lusta üzemmód közötti váltást teszi lehetővé. Érdekes lehetőséget nyújt a *setWildcard(bool)* metódus, ami képessé teszi az osztályt a filenév minták (*?* = valamilyen karakter, *=karaktersorozat, *[...]*) kezelésére. Ezeket a mintatípusokat a shell script programokból ismerhetjük.

A *QRegExpValidator* egy viszonylag egyszerű osztály. A *setRegExp()* metódussal beállítható, hogy milyen reguláris kifejezést akarunk használni.

Példa:

QRegExp rex("[1-9][0-9]{,2}");
QRegExpValidator v(rex, 0);
QString szam = "876";
v.validate(szam, 0); ...

A 3 jegyű pozitív számok a jók számunkra, amit a *v* objektum vizsgál a *rex* használatával. A validálás eredménye egy felsorolás típus értékeiből kerülhet ki: *Invalid*, *Acceptable*, *Intermediate*.

A *QString* egy sokat tudó sztringosztály, de a mi szempontunkból a *replace()* metódusa az, amivel a perl-ben létező karakterfűzér helyettesítési feladatokat könnyen el tudjuk végezni. Használatát a 2. algoritmus *(teszt1.cpp)* program illusztrálja.

A *QStringList* osztály a szövegsorok fogalmát valósítja meg. Megtalálható benne a jól ismert grep program funkcionalitása, amire a *grep()* metódus szolgál. A *grep(rex)* hívás egy *QStringList* objektumot ad vissza, ami pontosan azokból a listaelemekből áll, amik illeszkednek a *rex* mintára. A perl vágás (*split*) és összeragasztás (*join*) műveleteit bemutató program forráskódját a 3. algoritmus (*teszt2.cpp* file) tartalmazza.

2.7. Egy karakterfűzér részeinek cseréje

Gyakori feladat az, hogy egy string egy részét kell kicserélni egy másik karaktersorozatra. Amennyiben tudjuk a kezdő poziciót és a cserélendő hosszt, nem is olyan bonyolult a feladat. De mit tennénk, ha a szintaktikailag helyes e-mail címeket kéne mondjuk az *"XXX"* sztringre cserélni egy szövegben? Nézzük át a 2. algoritmusnál mutatott C++ forráskódot (teszt1.cpp forrásprogram)!

A műveletek bemutatását a 15. sorban definiált *str* fűzér segítségével tesszük meg. A 19. sor egy hagyományos sztringhelyettesítés. Kitörli az *str* fűzér 4. poziójától következő 5 karaktert (az *ember* szót), majd erre a helyre beszúrja a *vendég* szót. A 24. sorban lévő *replace()* már izgalmasabb, használja a 22. sorban megadott szabályos kifejezést. Mit mond a 22. sor mintája? Keressük meg azt az '*á*' betűt, ami után '*l*' betű következik. A 24. sor a *rex*-re illeszkedő részsztringet kicseréli a "*...a'...*" karaktersorozattal. A *replace()* metódus globálisan hajtja végre a cserét, azaz végigvizsgálja a teljes sztringet és minden illeszkedésnél cserél. A 27-32 sorok mutatják, hogy mi módon lehet csak az első illeszkedés cseréjét végrehajtani. A program futási képe a következő:

linux:/home/inyiri/cpp/qt_regexp # ./teszt1.exe
Egy vendég jött álmomban házamba.
Egy vendég jött ...a'...lmomban házamba.
Egy vendég jött álmomban házamba.

2.8. Vágás és ragasztás - igazi script módra

Képzeljük el, hogy van egy hosszabb sztringünk, aminek a belsejében egy reguláris kifejezéshez illeszkedő szakaszok vannak. Milyen jó a perl *split* szolgáltatása, ami úgy működik, hogy kiejti azokat a részeket, amikhez illeszkedik a minta, majd az így megmaradt sztring szigeteket egy listában adja vissza. Az összeragasztás egy sztringlistából egy sztringet állít elő, amiben a listaelemeket egy előre megadott szeparátor jellel választja el (*join* művelet). Erre mutat egy rövid példát a 3. algoritmusban mutatott program (teszt2.cpp):

A 14. sorban lévő fűzér egyes szakaszai két vagy több *'x'* karakterrel vannak elválasztva, amit a 17. sor szabályos kifejezése meg is fogalmaz. Mielőtt továbbmennénk nézzük meg a program futási eredményét:

linux:# ./teszt2.exe

alma italxxxkörtexxxxxxszilva és ringlóxxxxbarack

alma ital körte szilva és ringló barack

alma ital;körte;szilva és ringló;barack

Látható, hogy a 19. sor *split()* metódusa milyen sztring listát ad vissza, amit a 28. sorban összeragasztunk 1 db sztringgé a *';'* szeparátort használva.

2.9. A GUI vezérlők és a szabályos kifejezések együttműködése

Mindenki tudja, hogy milyen hasznos, ha egy beviteli szerkesztő sornak meg tudjuk mondani, hogy melyik pozición milyen karaktert fogadhat el. A szabályos kifejezésekkel ezt a feladatot nagyon általánosan és széleskörűen meg tudjuk oldani, amit a Qt rendszer támogat. Erre mutat példát a 6. algoritmusnál mutatott (maszkteszt.cpp) program. A bemutatott maszkolás olyan, hogy az első 2 pozición az *'a'* vagy *'b'* vagy *'c'* betűk lehetnek, amit tetszőleges számú (minimum egy) számjegy követhet. A 12-13 sorban meghatározunk egy érvényesítő objektumot, amit a 29. sorban az *lNev* szerkesztő vezérlőnek beállítunk a *setValidator()* metódusa segítségével.

2.10. Mintha a Java nyelv StringTokenizer osztályát használnánk...

Az utolsó példánkban (a 4. algoritmus mutatja) azt szeretnénk bemutatni, hogyan lehet egy sztringet részekre (azaz token-ekre) vágni úgy, hogy ehhez a reguláris kifejezést hívjuk segítségül. A minta *(...)* zárójel közötti részeit a mintaegyeztető algoritmus megjegyzi és ezekre a *\1*, *\2*, ... referenciákkal hivatkozni is lehet. Egy *QRegExp* objektum ezekre a megszerzett (mert már egyeztetett ott a minta) részfűzérekre a *cap(referencia_szám)* metódussal tud hivatkozni. Nézzük a már említett példát (teszt3.cpp) és talán minden sokkal világosabb lesz!

A 16. sor reguláris kifejezése szavakkal elmondva: Az egész vizsgálandó szövegre kell illeszkedni. Egy vagy több *nem ';'* után jön egy *';'*, majd ez a szabály még kétszer megismétlődik. A megjegyzendő helyeket a mintában *(...)* közé tettük. A 17. sor vizsgálja az illeszkedést, ami, ha sikeres, feltölti a *(...)* között meghatározott tokeneket, amiket a *cap()* metódussal lehet lekérni.

A program futási képernyője a következő:

linux:/home/inyiri/cpp/qt_regexp # ./teszt3.exe

Név: Petőfi Sándor Született: 1823 Foglalkozás: költő

Gondoljunk bele abba, hogy itt egy karakterfűzér szerkezeti felépítését adtuk meg egy minta segítségével, azután pedig ezeket a szerkezeti elemeket (tokeneket) könnyedén le tudtuk kérdezni.

3. Videózás Linuxon

Linuxon számos jó minőségű multimédia program megtalálható, a szerző kedvencei a *vlc*, *mplayer* (és benne a *mencoder*) és az *ffmpeg*, ami a nevével ellentétben csaknem minden formátumot magas szinten támogat. Ezen cikk elsősorban az *ffmpeg*(`http://ffmpeg.org`) lehetőségeire építve bemutatja a legfontosabb mindennapi videó file-okkal kapcsolatos feladataink lehetséges megoldásait.

3.1. Az ubuntu kiterjesztett multimédia támogatásának használatba vétele

A szerző elsődleges operációs rendszere az Ubuntu, ahol lehetőség van a már egyébként is magas szinvonalú környezetet tovább javítani, amire a *Mediabuntu* (`http://www.medibuntu.org`) oldal ad komoly támogatást. A software források repository adatbázisát egészítsük ki ezzel a hivatkozással:

deb http://packages.medibuntu.org/ jaunty free non-free

Ezzel sok multimédia csomag legszélesebb körét tudjuk - igény szerint - birtokba venni. Használatának előnye, hogy itt sokkal frissebb az *mplayer*, *vlc*, ... programok, illetve olyanok is vannak itt, amik az alap ubuntu-ban nincsenek.

Ezt a repo-t egy wget utasítással töltsük le:

sudo wget http://www.medibuntu.org/sources.list.d/hardy.list -O /etc/apt/sources.list.d/ medibuntu.list

Telepítsük ennek a kulcskarikáját is, különben állandóan rákérdez a csomagkezelő, hogy OK-e:

sudo apt-get install medibuntu-keyring

Aztán update parancs: *sudo apt-get update.* A következő a csomagok nincsenek benne az alap ubuntu-ban, most feltehetjük: *sudo apt-get install w32codecs libdvdcss2*

A Firefox plugin-t is telepítjük: *sudo apt-get install mozilla-mplayer*

3.2. Néhány szó az *ffmpeg* softwareről

Az *ffmpeg* sok részből áll (csak az *ffmeg* paranccsal fogunk foglalkozni), legalább a komponenseit érdemes megemlíteni:

- *ffmpeg.* Egy parancssori program, amivel rengetek hang és videó file formátumot tudunk egymásba konvertálni, hatékonyan és jó minőségben
- *ffserver.* Multimédia szerver (streaming server for live broadcasts). A cikkben nem ezt fogjuk használni, hanem a *flowplayer*-t.
- *ffplay.* Egy multimédia lejátszó
- programozói könyvtárak (*libavutil, libavcodec, libavformat, libavdevice, libswscale*). Külön érdemes kiemelni a *libavcodec* könyvtárat, ami sok audió és videó encoder/decoder függvényt tartalmaz nagyon magas szinten megvalósítva.

3.3. A videótartalom előállítása

Vágjunk a közepébe, nézzük meg, hogy egy tetszőleges kódolású videó file-ból milyen shell paranccsal lehet flash file-t készíteni:

```
ffmpeg -i egy-video.avi -ar 22050
-ab 32 -f flv egy-video.flv
```

Az *ffmpeg* gyakorlatilag minden input videó formátumot ismer. A példában egy *avi* videóból készítettük el az *flv* változatot. A minőség beállítására rengeteg paraméter szolgál, ennek összefoglalása itt olvasható:

`http://ffmpeg.org/documentation.html`

Fontos, hogy a videók között böngészve jó előnézeti képet kapjunk, ami általában 2 állapotú szokott lenni:

- amikor az egér nincs az előnézeti képen: egy állókép
- amikor az egér az előnézeti képen van: egy animált *gif* a videóról

A képsorozat előállítására a következő *ffmpeg* parancs alkalmas (a 0.1 érték azt jelenti, hogy az input videóról másodpercenként 0.1 képkockát vesz le, azaz 10 másodpercenként 1 darabot, most $100x100$-as méretben):

```
ffmpeg -i egy-video.avi -r 0.1
-s 100x100 -f image2 kep-%03d.jpeg
```

Ebből a képsorozatból kivehetjük a nekünk tetsző képet, ez lehet az inaktív (amikor az egér nincs az előnézeti képen) kép. A kapott képsorozatból egy videó így készíthető:

```
ffmpeg -f image2 -i kep-%03d.jpeg
-r 12 -s 100x100 kepek.avi
```

A kapott avi videó könnyen *animált gif* képpé alakítható, erre a Gimp program alkalmas (`http://www.gimp.org`)
A fenti módszer előnye, hogy jól automatizálható, nagyobb tömegű videó átkódolását, ütemezetten is el lehet vele készíteni, ugyanakkor mindegyik szoftver-elem széleskörben használt, jó minőségű és ingyenes (GPL licence).

Mini ffmpeg „szakácskönyv”-ünket az *mpeg4* formátumra való kódolás parancsával zárjuk:

```
ffmpeg -i <input_file> -an -b 300
-vcodec mpeg4 <output_file>
```

3.4. DVD készítés

A kiinduló állapot az, hogy rendelkezünk néhány, a kamerából letöltött AV (avi) file-lal: *hastanc_egom.avi*, *jazzbalett.avi*, *matrahaza.avi*, *hastanc_tisza.avi.* A feladat az, hogy ebből csináljunk egy olyan DVD-ét, amit az asztali lejátszón is meg tudunk nézni. A következő pontok azokat az eszközöket és lépéseket ismertetik, amiket a cikk szerzője szokott használni.

3.4.1. Az AVI→MPEG2 kódolások elvégzése

A DVD-ék anyagai *mpeg2* formátumúak. A feladathoz az *ffmpeg* programot használtam.

```
ffmpeg -i hastanc_egom.avi -target pal-dvd
-b 9000k hastanc_egom.mpeg
ffmpeg -i jazzbalett.avi
-target pal-dvd -b 9000k jazzbalett.mpeg
ffmpeg -i matrahaza.avi -target pal-dvd
-b 9000k matrahaza.mpeg
ffmpeg -i hastanc_tisza.avi -target pal-dvd
-b 9000k hastanc_tisza.mpeg
```

Megjegyzés: A *-b* a videó bitráta, ez határozza meg alapvetően az eredményfile méretét. Általában a *-b* kapcsoló nem kell, ez esetben egy 2 órás (4.7 GB) DVD-t tudunk készíteni. Amennyiben maximális minőséget akarunk (Ez azt jelenti, hogy 1 órás anyag a 4.7 GB-os DVD-re), akkor a *-b 9000k* a maximális bitrátához közeli érték (9800 k a DVD-nél használható maximum).

Leteszteltem ezt a parancsot is (azaz a bitráta a PAL DVD-nél szokásos 6000k körüli default értékkel):

```
ffmpeg -i hastanc_egom.avi
-target pal-dvd -b hastanc_egom.mpeg
```

A kapott hastanc_egom.meg mérete ebben az esetben: 2.1 GB, míg 9000k mellett: 3.0 GB. A nyers *hastanc_egom.avi* mérete érdekességként: 9.3 GB. A DVD média mérete 4.7 GB (vagy DL esetén ennek duplája), így az INPUT videót (videók együttes méretét) olyan kicsire kell

MPEG2-re encode-olni (tömöríteni), hogy azok elférjenek rá. Itt a *-b* kapcsoló 5000k-tól, 9000k-ig használható, a célméret igényei szerint.

3.4.2. A DVD összeállítása (authoring)

Ehhez a feladathoz is sok eszköz létezik, mi a *dvdauthor* programot használtuk. Ez is egy parancssoros program, így kell használni: *dvdauthor -o dvd -x movie.xml*

A lényeg a paraméterben átadott XML-ben van, ami azonban meglehetősen összetett is lehet,

ezért ennek az összeállításához a *tovid* utility csomag (az alap ubuntu csomag része) használata javasolt.

Egy DVD menü készítése Csináljunk 1 nekünk tetsző jpeg képből egy rövid mpeg videót, ami a DVD menűje lesz:

```
makemenu -pal -dvd -background
IMGA0424.JPG "Zsazsatánc - 2008
tavasz, nyár" -out mymenu
```

A makexml a *tovid* (telepítése: *sudo apt-get install tovid*) csomag része. A képen a "Zsazsatánc - 2008 tavasz, nyár" felirat is meg fog jelenni az előállított *mymenu.mpeg* mpeg2 mozi file-ban.

Amennyiben ez túl egyszerű nekünk, úgy használható a *mencoder* vagy *ffmeg* is erre a célra.

A dvdauthor xml file-jának elkészítése Adjuk ki ezt a *makexml* parancsot (ami szintén része a tovid csomagnak):

```
makexml -dvd -menu "mymenu.mpg"
-chapters 5 hastanc_egom_hd.mpeg
hastanc_tisza_hd.mpeg
matrahaza_20080725_hd.mpeg
jazzbalett_hd.mpeg
zsazsa_hastanc_20061209.mpg
-out myout
```

A "chapters 5" 5 percenkénti ugró indexet tesz az elkészített DVD-be. Látható, hogy az egy mpeg2 mozi file-okat egyszerűen csak fel kell sorolni. Az elkészített *myout.xml* (a format="pal" aspect="4:3" részt érdemes kézzel javítani a generált XML-ben):

A dvd szerkezet és az abban lévő file-ok legyártása Adjuk ki ezt a parancsot:

```
dvdauthor -o dvd -x myout.xml
```

Futás után a *myout* alkönyvtárba létrejön a szokásos *AUDIO_TS* és *VIDEO_TS* könyvtár.

A VIDEO_TS tartalma:

```
-rw-r--r-- 1 inyiri inyiri 6,0K 2008-08-01 16:37 VIDEO_TS.BUP
-rw-r--r-- 1 inyiri inyiri 6,0K 2008-08-01 16:37 VIDEO_TS.IFO
-rw-r--r-- 1 inyiri inyiri 84K 2008-08-01 16:36 VTS_01_0.BUP
-rw-r--r-- 1 inyiri inyiri 84K 2008-08-01 16:36 VTS_01_0.IFO
-rw-r--r-- 1 inyiri inyiri 658K 2008-08-01 16:36 VTS_01_0.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:36 VTS_01_1.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:37 VTS_01_2.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:37 VTS_01_3.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:37 VTS_01_4.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:37 VTS_01_5.VOB
-rw-r--r-- 1 inyiri inyiri 1,0G 2008-08-01 16:37 VTS_01_6.VOB
-rw-r--r-- 1 inyiri inyiri 605M 2008-08-01 16:37 VTS_01_7.VOB
```

3.4.3. Az elkészített DVD szerkezet tesztelése és kiírása

Tesztelés Ez a lépés azért célszerű, mert esetleg olyan hibákat vehetünk észre, amiket a már kiirt DVD-én nem javíthatunk. A *myout* a fenti DVD szerkezetet befogadó könyvtár, közvetlenül is lehet a DVD képet tesztelni (lejátszani).

VLC-vel: *vlc dvd:// -dvd-device myout*
vagy MPLAYER-rel:
mplayer dvd:// -dvd-device myout

Kiírás DVD-re A mi DVD anyagunk most nem fog felférni egy 4.7 GB-os lemezre (kb. 6 GB-os), ezért DL-re (Double Layer) írjuk majd. A DVD ISO képfile elkészítése ezzel a paranccsal lehetséges (a *myout* könyvtár, ahova az előző lépésben a DVD szerkezetet létrehoztuk):

```
mkisofs -dvd-video -udf
-o mydvd.iso myout/
```

Íme az elkészült DVD képfile:

6,6G 2008-08-01 17:44 mydvd.iso

DVD-re a k3b GUI alkalmazással írjuk ki az elkészült *mydvd.iso* képfile-t.

3.5. A hang leszedése egy videó forrásról

A feladatra az mplayer program megfelelő. A következő példában az *egyvideo.flv* flash videóról leszedjük a hangot, ami egy *audiodump.wav* fileba fog kerülni:

```
mplayer -vo null -vc dummy
-ao pcm egyvideo.flv
```

Természetesen a videó forrása bármilyen *mplayer* által ismert forrás vagy media lehet (azaz az egyetlen kritérium, hogy az *mplayer* le tudja játszani a videót).

Például egy DVD 4. sávján lévő videó hanganyagának lementése:

```
mplayer -vo null -vc dummy
-ao pcm dvd://4
```

Ha filmből a magyar hangot akarjuk kiszedni:

```
mplayer -alang hu -vo null
-vc dummy -ao pcm dvd://1
```

3.6. Egy Tube videószerver kialakítása

Manapság az Internet-en gyorsan növekszik a videó TUBE helyek száma. Miért ennek az előnye? A válasz egyszerű: a flash player plugin elég, hogy a böngésző része legyen és le tudjuk játszani a videót. Ez előrelépés ahhoz képest, amikor az Internet-en rengeteg videó formátum volt és azok lejátszása próbára tette a kliens oldalt (decoder telepítés). A megoldáshoz javasolt software a *flowplayer* (egyedüli oka, hogy ezt teszteltük le), de használhatnánk az *ffserver*-t is. A megoldáshoz használt komponensek:

- *Flowplayer* a videóstream szolgáltatáshoz (`http://flowplayer.org`)
- *ffmpeg* a videótartalom üzemszerű előállításához (`http://ffmpeg.org`)
- *Linux* + *Apache* webszerver operációs környezetként

A flowplayer egy tar.gz file-ban található, egyszerűen ki kell csomagolni az apache web document könyvtárába és már üzemkészen rendelkezésünkre áll. Az eszköz *flv* videókat játszik le, a kliens böngészőjével a *flash* és egy *Javascript* bridge segítségével tartja a kapcsolatot, amit a *html* dokumentum headerjében kell include-olni:

```
<script type="text/javascript"
src="flowplayer-3.1.1.min.js"></script>
```

Egy *video.flv* lejátszásának beszúrása:

```
<a href="http://localhost/video.flv"
style="display:block;width:520px;
height:330px" id="player"> </a>

<!-- this will install flowplayer
inside previous A- tag. -->
<script> flowplayer("player",
"../flowplayer-3.1.1.swf"); </script>
```