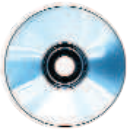
Az XML és a Perl (3. rész)

Húzd szorosabbra!



A legutóbbi részben ***olvas.pl*** néven közösen elkészítettünk egy nem túl összetett programot egy igen egyszerű XML jelölőnyelvvel írt dokumentum nyomtatható szöveggé alakításához. Nagy vonalakban így írtuk le a nyelvet: a dokumentum egy vagy több fejezetből áll, amelyek egymást követő bekezdéseket és beágyazott fejezeteket tartalmaznak. Minden fejezetnek van egy címe, és természetesen minden bekezdés karakterekből épül fel.
Bizonyára sokaknak feltűnt, hogy a program nem tiltakozik, ha a bemenet jól formázott XML ugyan, de nem felel meg a fenti leírásnak. Ha például egy olyan dokumentumot készítesz, amelyik a `<bekezdes>` elemen kívül is tartalmaz szöveget, az ***olvas.pl*** szó nélkül figyelmen kívül hagyná a karaktereket. Eddig azért nem beszéltem sokat az „alkalmazásszintű ellenőrzésről”, mert nem akartam, hogy a túl sok részlet között elvesszen a lényeg. Az alkalmazásszintű ellenőrzés azonban az XML-feldolgozásnak nagyon fontos része, így itt az ideje, hogy bővebben is bemutassam.

Szabályos nyelvtan

Említettem, hogy el lehet készíteni a jelölőnyelv nyelvtanát, az úgynevezett DTD-t (Document Type Definition), amely a megengedett elemeket egy XML leírónyelvben írja le. Egy ellenőrző értelmező ez alapján vizsgálhatja meg a dokumentumot. Bár mi nem használunk ellenőrző értelmezőt, egy DTD megírása így is hasznos gyakorlat lehet a nyelv szerkezetének leírásához.
Egy XML DTD-jelölőmeghatározásokból épül fel, amelyek `<!*` és `**>` között állnak. A jelölőmeghatározásoknak számos típusa létezik az XML-ben (és még több SGML-ben), de mi egyelőre csak az elemtípus-meghatározásokkal és a tulajdonságlista-meghatározásokkal foglalkozunk.
Egy elemtípus-meghatározás adja az elem nevét és a tartalommodellt, ez az elem lehetséges tartalmáról ad leírást. A tartalommodell valamennyire hasonlít a Perl szabályos kifejezéseihez. Készítsünk egy elemtípus-meghatározást `<dokumentum>` elemünkhöz:

```
<!ELEMENT dokumentum (fejezet+)>
```

Ez emberi nyelvre lefordítva mindössze annyit állít a `<dokumentum>` elemről, hogy egy vagy több `<fejezet>` elemet tartalmazhat. A `+` pontosan ugyanazt jelenti, amit a Perl a szabályos kifejezésekben tesz, így a `*` és a `?` is. Ha ezt írnánk:

```
<!ELEMENT dokumentum (fejezet)*>
```

azzal azt mondanánk, hogy a `<dokumentum>` nulla vagy több `<fejezet>`-ből állhat. A DTD elkészítésekor nagyon gondosan járjunk el, hogy a későbbiek folyamán ne kelljen felesleges ellenőrzőkódot írni a programunkba.
Most határozzuk meg a `<fejezet>` és `<bekezdes>` elemeinket:

```
<!ELEMENT fejezet (bekezdes | fejezet)*>
<!ELEMENT bekezdes (#PCDATA)>
```

A függőleges vonal a logikai „vagy” jele. A `#PCDATA` jelentése a „csak karakterek, semmi egyéb”, és valójában a „parsed character data” rövidítése, amelyben a „parsed” abban az értelemben áll, hogy már átfutott egy értelmezőn, és nem tartalmaz elemeket vagy hasonló XML-eszközt. Emiatt a `<`, `&` és a hasonló karaktereket a HTML-ből esetleg már megismert kóddal kell helyettesíteni, úgymint `&lt` és `&amp`.
A `<dokumentum>` és a `<fejezet>` tartalommodellje az, amit elemtartalomnak hívnak. Az ilyen tartalommal bíró elemek szöveget nem tartalmazhatnak, csak más elemeket.
A `<bekezdes>` tartalommodellje viszont úgynevezett vegyes tartalom – ez már szöveget is tartalmaz, és lehetséges volna olyan modellt készíteni, amiben elem és szöveg egyaránt megtalálható.
A szóköz, tabulátor, sortörés stb. kivétel képez az alól a szabály alól, hogy elemtartalomban nem szerepelhet szöveg. Másképp nem lehetne szépen megjeleníteni egy XML-dokumentumot, sok esetben még kézzel elkészíteni sem. Egy XML-értelmező nem hagyhatja figyelmen kívül az ilyen karaktereket, át kell őket adnia az alkalmazásnak, mintha vegyes tartalomban fordulnának elő. Ugyanakkor egy ellenőrző értelmező köteles szólni az alkalmazásnak, ha az említett karakterek az elemtartalomban jelennek meg; egy szövegformázó program valószínűleg figyelmen kívül hagyja őket.
Végül meg kell határoznunk a `<fejezet>` megengedett tulajdonságait (attributums):

```
<ATTLIST fejezet
       cim CDATA #IMPLIED >
```

Egy tulajdonságlista-meghatározás több tulajdonságot is felsorolhat, ebben az esetben azonban csak egyet tüntettünk fel, a ***cim***-et. A `CDATA` jelentése „character data”, ami egyszerű szöveget takar (ez viszont nem tartalmazhat közvetlenül például < nagyobb jelet). A `#IMPLIED` azt jelenti, hogy nem kötelező a tulajdonság megadása, és elhagyása esetén a feldolgozó alkalmazás alapértelmezett értékkel látja el. Ennek ellenére a tulajdonság nem állhat érték nélkül (lásd a Linuxvilág 32., októberi számának 70. oldalán). Ha ezen a helyen `#REQUIRED` állna, a tulajdonság elhagyása hibát okozna. Továbbá a `#IMPLIED` helyén egyszerű szöveg is szerepelhetne, amely hiányzó tulajdonság esetén alapértelmezett értékként viselkedne.
Az XML DTD sokkal több, mint amennyit itt össze tudtam foglalni. Nagyobb részletességgel szolgál az XML-ajánlás, és remek leírásokat találsz a weben, ha beizzítod a Google-t.

Kötelező nyelvtan

Ha az ***olvas.pl*** előző változatát kötelező nyelvtannal akarjuk felvértezni, ellenőriznie kell az elemek és a karakterek megfelelő helyen történő előfordulását. Jogos a kérdés, hogy miként tehetem ezt meg egy folyamalapú értelmezővel?

Igazság szerint az `XML::Parser` sokkal több, mint aminek az előző részekben elmondtam, és tartalmaz némi korlátozott lehetőséget az ellenőrzésre. Pontosabban fogalmazva a segítségével kitalálhatod, hogy a jelenlegi elem mely elemekbe lett beágyazva. Mint azt tudod, az első érték, amit az értelmező egy esemény kiváltásakor a meghívásra kerülő függvénynek átad, egy hivatkozás az értelmező objektumra. Ennek az objektumnak a típusa valójában az `XML::Parser::Expat`, és nem az `XML::Parser` maga. Az `XML::Parser::Expat` POD-jának gondos átolvasásával az ellenőrzés számos módszerét ismerheted meg. Az `in_element()` és a `within_element()` elemfüggvények különösen hasznosak a feladat megoldására. Álljon itt az ***olvas1.pl***, amely az ***olvas.pl*** hibaellenőrzéssel kibővített változata:

```
#!/usr/bin/perl -w
use strict;

use XML::Parser;
use Text::Wrap;
use Getopt::Std;

my ($indlevel,@sectnums,$parabuf,%opts);

getopts('c',\%opts);
die "Usage: ".$0." [-c] file\n"
➥unless @ARGV == 1;
my $p = new XML::Parser
➥(Style => 'Stream',ErrorContext=>2);
```

Az `ErrorContext` azt mondja meg az értelmezőnek, hogy hiba esetén hány sort mutasson meg a hibát tartalmazó sor előtt és után:

```
$p -> parsefile ($ARGV[0]);

sub StartTag {
     my ($expat,$eltype) = @_;
     if ($eltype eq "dokumentum") {
       $expat->xpcarp("<dokumentum> itt nem
       ➥fordulhat elo") if $expat->depth();
```

Az `xpcarp()` elemfüggvény segítségével egy figyelmeztető üzenetet jeleníthetünk meg a hiba környezetével együtt. A `depth()` elemfüggvény adja meg, hogy a jelenlegi elem hány elembe van beágyazva. A beágyazás számlálója a `StartTag()` meghívása után növekszik, így ez az ellenőrzés minden `<dokumentum>`-ot visszadob, a gyökérelemet leszámítva.

```
       $indlevel = -1;
       $sectnums[0] = 0;
       $parabuf = "";
       return;
     }
     $expat->xpcroak("<dokumentum> kell,
     ➥hogy a gyoker elem legyen") unless
     ➥$expat->within_element("dokumentum");
```

Ha az elemünk nincs egy `<dokumentum>`-on belül, feladjuk az értelmezést (ha nem tennénk, minden ezt követő elem esetén egy figyelmeztetést adnánk ki). Az `xpcroak()` hasonlít az `xpcarp()`-ra, de ez a figyelmeztetés mellett megszakítja a program futását. A `within_element()` nem figyeli a beágyazás mértékét. Akkor ad hamis értéket, ha a beágyazás egyik szintjén sem talált `<dokumentum>` elemet.

```
     if ($eltype eq "fejezet") {
       if (not
       ➥($expat->in_element("dokumentum")
       ➥or $expat->in_element("fejezet"))) {
           $expat->xpcarp("<fejezet> itt
           ➥nem fordulhat elo");
       }
```

Az `in_element()` azt nézi meg, hogy a jelenlegi elem melyik elembe van közvetlenül beágyazva. A fenti ellenőrzéssel megbizonyosodhatunk arról, hogy a `<fejezet>` csak a `<dokumentum>`-on vagy egy másik `<fejezet>`-en belül fordulhat elő.

```
       ++$sectnums[++$indlevel];
       $sectnums[$indlevel+1] = 0;
       print ' 'x(4*$indlevel),join
       ➥('.',@sectnums[0..$indlevel]),
       ➥" ",$_{cim},"\n";
       print "\n" unless $opts{c};
     } elsif ($eltype eq "bekezdes") {
       $expat->xpcarp("<bekezdes>
       ➥itt nem fordulhat elo") unless
       ➥$expat->in_element("fejezet");
     } else {
       die "Ismeretlen elem: ",$eltype,"\n";
     }
}

sub Text {
  my $expat=shift;
  tr/\n/ /;
  s/^\s+//;
  s/\s+$//;
  return if $_ eq "";
  $expat->xpcarp("szoveg itt nem fordulhat elo")
       ➥unless $expat->in_element("bekezdes");
```

Figyeld meg, hogy csak azután ellenőrzünk, miután megbizonyosodtunk róla, hogy a szöveg nem csak az angol szakzsargonban whitespace-nek nevezett karaktereket tartalmaz. Ha nem így tennénk, gyakorlatilag minden elem után figyelmeztetést kapnánk.

```
     $parabuf = $_;
}

sub EndTag {
     my ($expat,$eltype) = @_;
     if ($eltype eq "dokumentum") {
     } elsif ($eltype eq "fejezet") {
       --$indlevel;
     } elsif ($eltype eq "bekezdes") {
       my $ind = ' ' x (4*$indlevel);
       print wrap($ind,$ind,$parabuf),
       ➥"\n\n" unless $opts{c};
     }
}
```

A zárócímkéknél semmilyen ellenőrzésre nincs szükség, hiszen magának az értelmezőnek kell szólnia, ha ezek nem a megfelelő helyen állnak. Egy jellegzetes folyamalapú értelmezővel dolgozó alkalmazásnál igen gyakori, hogy az ellenőrzés és a változók használatra való felkészítése a nyitó címkék feldolgozásánál, míg az eredmény visszaadása a zárócímkéknél történik.

Faalapú értelmezés

Az előző hónapban több, kizárólag faalapú feldolgozásra tervezett modult is megemlítettem. Valójában az `XML::Parser` is nyújt két faalapú stílust. A „Tree" stílus egy nagyon egyszerű faszerkezetet hoz létre, melynek csomópontjai Perl-tömbökre mutató hivatkozások. Két típusú levél létezik:

- A szövegcsomópont: a tömb első tagja üres, míg a második maga a szöveg.
- Az elemcsomópont: a tömb első tagja az elem neve, a második pedig hivatkozás egy tömbre, amely az elem tartalmát foglalja magában. Az utóbbi tömb első tagja egy hivatkozás egy asszociatív tömbre (hash), amely az elem tulajdonságait sorolja fel az értékeikkel együtt. A további tagok szöveg- vagy elemcsomópontok.

Amikor a Tree stílust használod, az `XML::Parser parse()` és a `parsefile()` elemfüggvényei egy, a gyökérelem csomópontjára való hivatkozást adnak vissza.
Habár a Tree stílus leírása az `XML::Parser` utóbbi változataiban határozottan tisztább és áttekinthetőbb a réginél, ennek ellenére a használt adatszerkezeteket megérteni nem a legegyszerűbb dolog a világon. Ne legyen rossz érzésed, ha első nekifutásra nem értetted meg teljesen a fenti magyarázatot. Ha megnézted a következő példaprogramot, újra elolvastad a fentieket, ismét átnézted a kódot, előbb-utóbb rájössz, mit is jelent a „heuréka!" kifejezés. Az egyik ok, amiért érdemes lehet faalapú értelmezőt használni a folyamalapúval szemben, az az, hogy az értelmező egyszeri meghívásával ugyanazt a tartalmat többször akarod elemezni. Változtassuk meg az ***olvas.pl*** feladatát: ahelyett, hogy vagy a tartalomjegyzéket írja ki, vagy a tartalmat, inkább írja ki mindkettőt: előbb a tartalomjegyzéket, s ezt kövesse a tartalom. Ezt megtehetnénk egy folyamalapú értelmezővel is, ha két átmeneti tárban (buffer), külön tárolnánk a kimenetet, de az nem lenne olyan szórakoztató. Inkább felépítünk egy fát és kétszer bejárjuk. Az első bejárás alkalmával kiírjuk a tartalomjegyzéket és a második bejárás során a teljes tartalmat. Lássuk (ez ugyancsak megtalálható az 53. CD-melléklet Magazin/XML_Perl/olvas2.pl könyvtárában)!

```
#!/usr/bin/perl -w
use strict;

use XML::Parser;
use Text::Wrap;

my ($indlevel,@sectnums);

die "Usage: ".$0." file\n" unless @ARGV == 1;
my $parser = new XML::Parser
➥(Style => 'Tree',ErrorContext=>2);
my $tree = $parser->parsefile($ARGV[0]);
die "<dokumentum> nem a gyokerelem"
➥unless $tree->[0] eq "dokumentum";
```

Ha minden rendben van, a `$tree` egy hivatkozás egy kételemű tömbre, amelynek az első tagja „dokumentum" (a gyökérelemünk neve), a második pedig egy hivatkozás egy tömbre a `<dokumentum>` tartalmával.

```
$indlevel = -1;
$sectnums[0] = 0;
for my $pass (0..1) {
  my $p = $tree->[1];
  for (my $i = 1; $i < @$p; $i += 2) {
      if (not $p->[$i]) {
        warn "szoveg nem allhat itt"
        ➥unless $pass or $p->[$i+1]=~/^\s*$/;
        next;
      }
      warn "<".$p->[$i]."> nem allhat itt"
      ➥unless $pass or $p->[$i]
      ➥eq "fejezet";
      process_sect($p->[$i+1],$pass);
  }
}
```

Átugorjuk a `<dokumentum>` tulajdonságait tartalmazó asszociatív tömböt (hiszen elvileg nincsenek is neki tulajdonságai, az esetleges hamis tulajdonságokat pedig figyelmen kívül hagyjuk). A megmaradt tagokon párosával megyünk végig. Ha a dokumentum nyelvtanilag helyes volt, vagy üres szövegcsomópontokból, vagy `<fejezet>`-ekből tevődik össze. Az utóbbiak feldolgozását a `process_sect()` függvényre bízzuk, mivel egy `<fejezet>`-ben előfordulhat újabb `<fejezet>`, s ezt a legegyszerűbben rekurzívan oldhatjuk meg.

```
sub process_sect {
  my ($p,$pass) = @_;
  my ($href,$ind);
  if ($pass == 0) {
    ++$sectnums[++$indlevel];
    $sectnums[$indlevel+1] = 0;
    $href = join('.',@sectnums[0..$indlevel]);
    $ind = $indlevel;
    $p->[0]{_href} = $href;
    $p->[0]{_ind} = $ind;
    print ' ' x (4*$ind),$href,
    ➥" ".$p->[0]{cim}."\n";
```

Első körben kiszámoljuk a fejezetszámot és a bekezdés nagyságát, és a `<fejezet>` elem külön tulajdonságaként tároljuk. A tulajdonságok a tömb első tagja által mutatott asszociatív tömbben tárolódnak, ezzel a módszerrel dinamikusan bővítettük az értelmező által visszaadott fát, ugyanis csökkenti a létrehozandó és karbantartandó adatszerkezetek számát. Ezután a tartalomjegyzékhez kiírjuk a fejezet számát és a címét.

```
    } else {
      $href = $p->[0]{_href};
      $ind = $p->[0]{_ind};
      print ' ' x (4*$ind),$href,
      ➥" ".$p->[0]{cim}."\n\n";
    }
```

A második körben előkaparjuk a fejezet számára és a bekezdés nagyságára vonatkozólag az első körben elrejtett adatokat. Ezt használjuk fel a kiíratáshoz a teljes dokumentum megjelenítésekor is.