Az XML és a Perl (2. rész)

Amikor a nyelv találkozik az értelmezővel…

Amikor egy nyelv értelmezéséről beszélünk, arra az elemzésére gondolunk, amikor a nyelv szabályait követve apró részekre bontjuk fel a bemeneti adathalmazt. Az értelmezés alapvetően szükséges minden olyan program esetén, amelyik strukturált adatot használ bemenetként. Az előző részben az egyszerű adatnyelvek között említést tettem egy olyan módszerről, amelyik vesszővel elválasztott mezőket tartalmazó sorokból álló állományt használ adatforrásként. Perlben az utóbbit valami ehhez hasonló módon értelmeznénk:

```
while (<INPUT>) {
       # @mezok tomb felhasznalasa
       @mezok = split /,/;

}
```

A fenti kód azonban csak ráhúzza a szerkezetet az olvasott állományban tárolt karaktersorozatra. Ebben az esetben a szerkezet implicitnek nevezhető. Az az adat, hogy a rekordok első mezője mondjuk valakinek a vezetékneve, míg a második a keresztneve, explicit módon nem derül ki az állományból. A programnak kell tudnia, hogy melyik micsoda.

Hogyan ne értelmezzünk XML-t?

Kezdő, sőt még komolyabb Perl-programozók is gyakran beleesnek abba a hibába, hogyha az adatforrás egy HTML-dokumentum vagy egy XML-alkalmazás, azonnal sorról sorra értelmezik és szabályos mintaillesztéssel keresik meg a címkéket. Ez nem túl jó módszer szerkesztett jelölőnyelvek esetében! Először is a sortöréseknek semmi jelentőségük nincs az SGML/XML-alkalmazások többségénél, mindössze a szerkesztés kényelmessé tételét szolgálják.
Másodszor, aki olvasta a Perl GYK-t, az tudja, hogy a Perl mintaillesztési kifejezések nem használhatók egymásba ágyazott zárójelek esetén (ugye, mind olvastátok a GYK-t?). „Mi köze van ennek az XML-hez?" – kérdezheted. XML-t értelmezni egy az egyben egymásba ágyazott zárójelek sokaságának értelmezését jelenti.
Harmadszor az XML-ajánlás engedélyezi logikailag egy dokumentum fizikai értelemben több részben történő tárolását.
Az SGML, amire a HTML is épül, ugyanazt a beszúrási módszert teszi lehetővé, csak sajnos igen kevés böngésző ismeri. Senki sem szeretné, hogy XML-alkalmazása ugyanilyen nehézségektől szenvedjen. A beszúrás módszere temérdek egyszerű, egyúttal unalmas kódot követel a programozótól.

Akkor hogyan értelmezzünk XML-t?

Az eddig leírtak figyelembevételével beláthatjuk, hogy XML-értelmezőt írni nem egy leányálom. A W3C egyik eredeti szándéka az XML kapcsán az volt, hogy egy diplomás informatikus egy hét alatt képes legyen megírni egy értelmezőt. Úgy néz ki, ez azoknak a céloknak az egyike volt, amit nem sikerült elérni, s aminek a megvalósítását az XML fejlesztésével megjelent új lehetőségek, például a névtér használata, még lehetetlenebbé tették.
Egy XML-értelmező felépítésének összetettsége ennek ellenére sem fogott vissza több tehetséges, magányos programozót és csoportot – ők megírták a saját értelmezőjüket. Ezek közül igen sok szabadon hozzáférhető. Így majdnem mindenki, aki XML-t használ bemenetként, egy előre megírt értelmezőt használ, ami az összes piszkos munkát elvégzi. A Perl világában ez olyan értelmezők használatát jelenti, amelyek modulok formájában elérhetőek.

Az értelmezők típusai

Az XML-dokumentumot az összes értelmező az alábbi összetevőkre bontja:

- elemek,
- egy tulajdonság-érték párokból álló lista minden elemhez (ez üres is lehet),
- az elemek tartalmát jelentő karakterek (szöveg),
- feldolgozói utasítások, amelyek egy bizonyos programnak szólnak, és nem az adatot írják le, XML-ben ezek `<?` és `?>` között találhatók; egy feldolgozói utasítás tartalma egy tetszőleges név, amit célnak is hívunk, és azt az alkalmazást jelöli, amelynek az utasítás szól; ezt egy egyenlőségjel követi, majd egy karakterlánc, amit már az alkalmazás fog feldolgozni, és
- megjegyzések, azaz olyan szövegek, amelyek `<!--` és `-->` között találhatók.

Valójában egy XML-értelmezőnek nem kötelessége az alkalmazásnak a megjegyzéseket átadnia, de nagyon sok megteszi. Ezért ne számíts erre a viselkedésre, ne akarj így utasításokat átadni egy programnak. Erre használd a feldolgozói utasításokat.
Az XML-értelmezők osztályozásakor két független szempontot kell figyelembe vennünk: vannak ellenőrző és nem ellenőrző, illetve folyamalapú és faalapú értelmezők.

Ellenőrző és nem ellenőrző értelmezők

Az előző cikkben megemlítettem, hogy az SGML-értelmezőknek a dokumentum helyes értelmezéséhez szükségük van a jelölőnyelv nyelvtanára, a DTD-re (Document Type Definition). A DTD írja le a használható elemtípusokat, az ezekhez tartozó tulajdonságokat (attributum), ha vannak, és azt, hogy hogyan lehet az egyes elemeket egymásba ágyazni. A HTML DTD-je többek között meghatározza, hogy a `<TD>` elem kizárólag a `<TR>`-en belül helyezkedhet el.
Az XML tervezésének köszönhetően egy jól formázott XML-dokumentumot DTD nélkül is értelmezni lehet. A jól formázott XML-dokumentummal szembeni követelmények az XML-ajánlásban találhatók. Így minden elemnek van kezdő és záró címkéje, az elemek egymásba ágyazhatók, de nem nyúlhatnak át egymáson, és minden dokumentum csak egy gyökérelemet tartalmazhat.
Bár a DTD nem kötelező egy XML-dokumentumban, egy ellenőrző értelmező használhatja a DTD-t arra, hogy megbizo-

nyosodjon az XML alkalmazásszabályainak betartásáról. Egyfajta séma ez – ha megszeged, az ellenőrző értelmező hangosan tiltakozik. Ez igen hasznos lehet akkor, ha külső forrásból származó XML-dokumentumot kell használnod. Például a beszállítód XML-formátumban küldte el a számlát – az utóbbi esetben nem árt ellenőrizni, hogy a megfelelő elemek a megfelelő sorrendben szerepelnek-e. A DTD továbbá meghatározhat alapértelmezett értékeket bizonyos tulajdonságokhoz, amit egy ellenőrző értelmező felhasználhat, ha az adott elemnél ez nincs megadva.
Egy nem ellenőrző értelmezőnek viszont nincs másra szüksége, mint hogy a dokumentum jól formázott legyen. A nem ellenőrző értelmezők sokkal egyszerűbbek ellenőrző társaiknál, emiatt a legtöbb ingyenes értelmező nem ellenőrző. A nem ellenőrző értelmezők kielégítő megoldást jelenthetnek, ha a dokumentum saját forrásból származik, vagy annyira összetett nyelvtana van, hogy nem lehet DTD-vel leírni, és az alkalmazásnak kell eldöntenie, hogy helyes-e. Létezik szabadon hozzáférhető ellenőrző XML-értelmező modul Perlhez `XML::Checker` néven, de sajnos még nem mondható üzembiztosnak.

Folyamalapú és faalapú értelmezők

Egy értelmező alapvetően kétféleképpen tálalhatja az adatot az alkalmazásnak. Az egyik módszer az, hogy a dokumentum olvasása közben minden egyes alkalommal, amikor új összetevőre bukkan, egy jelzést küld az alkalmazásnak. A másik lehetőség, hogy az egészet beolvassa, majd egy, a dokumentum felépítésével összhangban álló faszerkezetet ad át a programnak. Az elsőt folyamalapú vagy eseményvezérelt értelmezőnek, míg a másodikat faalapú értelmezőnek hívjuk. Két fogalommal gyakran fogsz találkozni a SAX és a DOM kapcsán. A SAX (Simple API for XML) egy, az xml-dev levelezési lista tagjai által létrehozott nem hivatalos ajánlás arról, hogy egy folyamalapú értelmező hogyan kell, hogy beszélgessen az alkalmazással.
A DOM (Document Object Model) a W3C hivatalos ajánlása, amelyben azt írja le, hogy egy alkalmazás hogyan érheti el és módosíthatja egy dokumentum faszerkezetét.
Melyiket használd? Ez a feldolgozás természetétől és a dokumentum méretétől függ. Egy faalapú értelmezőnek az egész dokumentumot be kell töltenie a memóriába, ami nem kifejezetten hasznos nagy szótárak vagy más adatbázisok esetén. Egy folyamalapú értelmezővel átugorhatod azokat az elemeket, amelyek nem érdekesek számodra. Egy adott szócikkre történő keresés esetén kifejezetten az utóbbi módszer javasolt. Viszont ha olyan elemeket keresel, amelyek kapcsolatban állnak más elemekkel (például azokat az írókat keresed, akik egy témában legalább három cikket adtak le), egy faalapú értelmezővel sokkal könnyebben boldogulhatsz.
Érdemes megjegyezni, hogy egy faalapú értelmező használható egy folyamalapú tetején, illetve egy faalapú értelmező kimenete bejárható úgy, hogy folyamalapú felületen keresztül érje el az alkalmazás.

XML-értelmező modulok Perlhez

Jelenleg négy főbb XML-értelmező érhető el modulok formájában. Ezek mind szabadon hozzáférhetőek és letölthetőek a CPAN oldaláról (➲ http://www.cpan.org).

XML::Parser

Ez az összes értelmező nagypapája. ***James Clark*** C-ben írt Expat értelmezőjén alapul, és eredetileg nem más, mint ***Larry Wall*** írta. Jelenlegi karbantartója ***Clark Cooper***.
Az `XML::Parser` alapvetően folyamalapú, nem ellenőrző értelmező; lényegében függvényeket kell rendelnie az egyes eseményekhez. Az esemény kiváltásakor a megadott függvény meghívásra kerül.

XML::DOM

Ez a faalapú értelmező a W3C 1. szintű DOM-felületét valósítja meg. ***Enno Derksen*** írta; az `XML::Parser` képezi az alapját.

XML::Parser::PerlSAX

Ken MacLeod viszonylag új folyamalapú értelmezője a SAX-felületet valósítja meg. A felülete sajnos nem annyira perles, mint az `XML::Parser`-é.

XML::Grove

Ken MacLeod faalapú értelmezője. Míg az `XML::DOM` külön függvényhívásokat igényel a fa bejárásához, az XML::Grove olyan fát hoz létre, amelyik szokványos Perl-tömbműveletekkel használható. Emiatt gyorsabb lehet, ha sokszor kell bejárni a fát, mivel Perlben a függvényhívás meglehetősen erőforrás-költséges.
Ezek a modulok egytől egyig objektumközpontú felülettel rendelkeznek. Ha még nem barátkoztál meg az objektumközpontú programozással Perl alatt, itt az ideje, hogy átnézd a `perltoot(1)` és `perltootc(1)` kézikönyvoldalakat – Debian alatt a ***perl-doc*** csomag tartalmazza őket.

Következzen egy valódi alkalmazás!

Eleget beszéltünk arról, hogy mit lehet beszerezni, kezdjük el használni is! Az XML egyik lehetséges felhasználási módja – habár nem az egyetlen –, hogy leírja egy dokumentum szerkezetét, így a nézegető már emberi fogyasztásra alkalmas formában jelenítheti meg. Ebben a példában egy rendkívül egyszerű XML alapú jelölőnyelvet hozunk létre, utána egy hihetetlenül rövid programot írunk a dokumentum megfelelő formázására, illetve a tartalomjegyzék kiíratására.

Dokumentumunk jelölőnyelve

A dokumentum egy vagy több fejezetből áll, amelyek egymást követő bekezdéseket és beágyazott fejezetek tartalmaznak. Minden fejezetnek van egy címe, és minden bekezdés természetesen karakterekből épül fel. Az első feladatunk ezeket az állításokat XML-es fogalmakkal leírni. A dokumentum egyértelműen egy elem, nevezetesen a gyökérelem. Ez alapján a dokumentumunk így néz ki:

```
<dokumentum>
<!-- valami meg jon ide -->
</dokumentum>
```

Továbbá minden fejezetnek külön elemnek kell lennie, ezeket hívjuk `<fejezet>..</fejezet>`-nek. A fejezetcím lehetne külön elem a `<fejezet>` elemen belül, de ezúttal legyen a `<fejezet>` elem tulajdonsága. Végül a bekezdéseket a `<bekezdes>` elemmel fogjuk jelölni. Ezek szerint egy teljes dokumentum így fest:

```
<dokumentum>
     <fejezet cim="elso">
       <bekezdes>Ez az elso fejezet</bekezdes>
       <fejezet cim="elso alfejezet">
            <bekezdes>Ez az elso fejezet
elso alfejezete
            </bekezdes>
```

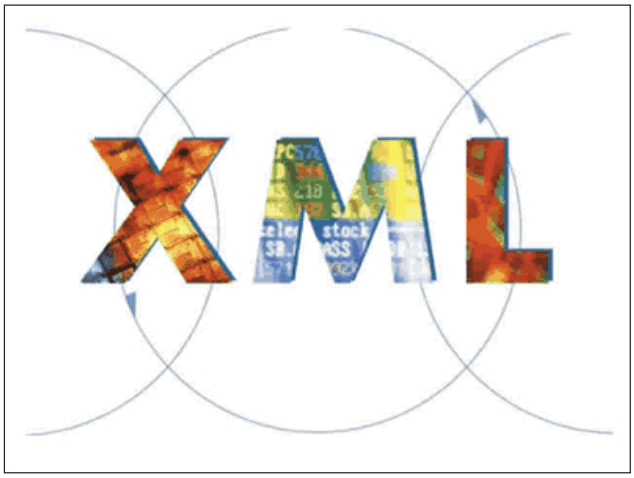
```
          <fejezet cim="elso al-alfejezet">
            <bekezdes>Ez az amit a cim is
sugall</bekezdes>
            <bekezdes>Ez meg a masodik
bekezdes</bekezdes>
          </fejezet>
          <bekezdes>Ez tovabbi szoveg
az elso alfejezethez. Jol nez ki, ugye?
          </bekezdes>
      </fejezet>
      <fejezet cim="masodik alfejezet">
          <bekezdes>Ez az elso fejezet
masodik
alfejezeteben szerepel.
          </bekezdes>
      </fejezet>
    </fejezet>
    <fejezet cim="masodik">
      <bekezdes>Ez a nagyon egyszeru program
az XML-
dokumentumok folyamalapu ertelmezeset
mutatja be. Beolvassa az allomanyt es
megfelelo sorszamozassal, bekezdesekre
tordelve talalja a tartalmat.
      </bekezdes>
    </fejezet>
</dokumentum>
```

Nézegetőprogramunk

Most egy olyan programot szeretnénk írni, ami a fentihez hasonló dokumentumot ekképp jeleníti meg:

```
1 elso

Ez az elso fejezet

    1.1 elso alfejezet

    Ez az elso fejezet elso alfejezete

        1.1.1 elso al-alfejezet

        Ez az amit a cim is sugall

        Ez meg a masodik bekezdes

    Ez tovabbi szoveg az elso alfejezethez.
Jol nez ki, ugye?

    1.2 masodik alfejezet

    Ez az elso fejezet masodik alfejezeteben
szerepel.

2 masodik

Ez a nagyon egyszeru program az XML-
dokumentumok folyamalapu ertelmezeset
mutatja be. Beolvassa az allomanyt es
megfelelo sorszamozassal,
bekezdesekre tordelve talalja a tartalmat.
```

Esetleg így:




```
1 elso
    1.1 elso alfejezet
        1.1.1 elso al-alfejezet
    1.2 masodik alfejezet
2 masodik
```

Figyeld meg, hogy XML-dokumentumunk sehol sem tartalmazza, hogyan kell sorszámozni, illetve bekezdésekre tördelni a szöveget. Kizárólag a fejezetek és bekezdések helye van megadva. Ez is azt a nézetet tükrözi, hogy a tartalmat és a kinézetet el kell választani egymástól – ez nagy előnye a strukturált jelölőnyelveknek.
Mivel a feldolgozás módszere nem igényel véletlen elérést a dokumentumban, a folyamalapú `XML::Parser`-t fogjuk használni.

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

Ugye, mindig használod a `-w`-t és a `strict`-et?

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

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

A `$indlevel` a beágyazás mélysége, a `@sectnums` a fejezetszámokat tároló tömb, a `$parabuf` pedig mindig a pillanatnyi bekezdést tartalmazza.

```
getopts('c',\%opts);
```

Ha a `-c` kapcsoló meg van adva, akkor csak a tartalomjegyzéket írjuk ki.

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

Létrehozunk egy új értelmező objektumot, és meghatározzuk, hogy a `'Stream'` stílust akarjuk használni. A stílusokról bővebben az `XML::Parser` POD-jában olvashatsz.

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

Az objektumnak átadunk egy állományt, és hagyjuk, hogy végezze a feladatát. Ezzel az utasítással a program véget ért. Az igazi érdekesség azonban az események kiváltásakor életbe lépő függvényekben rejlik:

```
sub StartTag {
    my ($expat,$eltype) = @_;
```

Amikor a `'Stream'` stílussal dolgozunk, az értelmező – akárhányszor csak meglát egy nyitócímkét – meghívja a `StartTag` függvényt. Az első átadott érték egy hivatkozás az értelmező objektumra, a második pedig egy karakterlánc a címke nevével. A `%_` a tulajdonság–érték párok listáját tartalmazza, míg a `$_` a teljes címkét magába foglalja a kisebb és nagyobb jelekkel együtt.

```
    if ($eltype eq "dokumentum") {
      $indlevel = -1;
      $sectnums[0] = 0;
      $parabuf = "";
```

Változóinkat kiindulási értékekkel látjuk el, amikor a gyökérelemmel találkozunk. Ezt egy `StartDocument` függvényben is megtehettük volna, ami akkor kerül meghívásra, amikor az értelmező hozzálát az elemzéshez.

```
    } elsif ($eltype eq "fejezet") {
      ++$sectnums[++$indlevel];
      $sectnums[$indlevel+1] = 0;
      print ' 'x(4*$indlevel),join
      ➥('.',@sectnums[0..$indlevel]),
      ➥" ",$_{cim},"\n";
      print "\n" unless $opts{c};
```

A `<fejezet>` típusú elemnél meg kell növelnünk a beágyazás mélységét és a fejezetszámot, illetve a következő fejezetszámot nulláznunk kell. Továbbá a címet is ki kell íratni a megfelelő bekezdéssel és számozással.

```
    } elsif ($eltype eq "bekezdes") {
```

Ebben a példában a bekezdések elején nem teszünk semmit, de egy kifinomultabb program esetén erre az ágra is szükség lehet.

```
    } else {
      die "Ismeretlen elem: ",$eltype,"\n";
    }
```

Mivel nem ellenőrző értelmezőt használunk, legalább egy kis ellenőrzésre szükség van az alkalmazás részéről. Ismeretlen elem esetén azonnal megszakad a program futása.

```
}

sub Text {
    tr/\n/ /;
    s/^\s+//;
    s/\s+$//;
    return if $_ eq "";
    $parabuf = $_;
}
```

Az értelmező a `Text` függvényt hívja meg, amikor karakterlánccal találkozik. A `$_` változó tartalmazza a szöveget a következő nyitó- vagy zárócímkéig, feldolgozói utasításig, illetve megjegyzésig. Egy XML-értelmezőnek az összes karaktert kötelessége átadni az alkalmazásnak, beleértve a szóközöket, a sortöréseket és a tabulátorokat. Ezért minden felesleges karaktert le kell vágnunk a szöveg elejéről és végéről. Figyelmen kívül hagyjuk azokat a karakterláncokat, amelyek például csak szóközből állnak. Ezek mindössze az XML-dokumentum könnyebb szerkeszthetőségét szolgálják. A sortöréseket szóközökké alakítjuk.
Egyedül a `<bekezdes>` elemen belüli szöveg érdekel minket. Ezért programunk a `<bekezdes>`-eken kívül minden karaktert szép csendben figyelmen kívül hagy.

```
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};
    }
}
```

Amikor a zárócímkébe ütközünk, munkánk véget ért. A bekezdéseket formázva kiíratjuk a képernyőre.

Mi vár rád a következő részben?

Ennyi volt e hónapra. A következő részben folytatjuk vizsgálódásunkat az XML-értelmezők körül, és közelebbről is megnézzük a faalapú értelmezők használatát. A fentebb szóba került összes példa megtalálható a CD-mellékleten.

Köszönetnyilvánítás

Szeretnék köszönetet mondani ***Eric Bohlman***-nak a hathatós közreműködésért.

Fülöp Balázs (admin@guardware.com)
18 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mro□ek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.