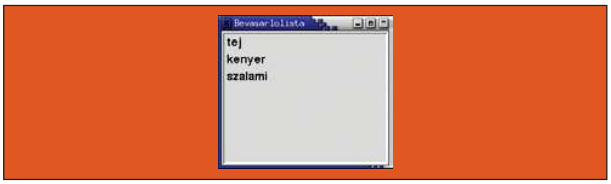
# Listaablakok röptében a Tk::HList segítségével

A bevásárlólistától a /etc/passwd-ig grafikus külsőt kaphat bármi, ha Perl alatt kipróbálod a HList elemet.

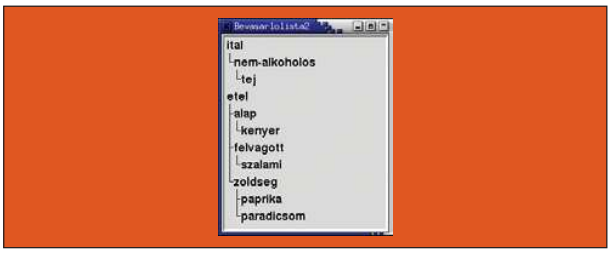
Ezúttal rendhagyó módon szeretnék eltekinteni az unalmas történelmi bevezetőtől. Miután teljesen nyilvánvaló, hogy a kedves Olvasó előbb a képernyőfotókat nézi meg, majd azonnal a kódra kíváncsi, térjünk rögtön a lényegre. Elöljáróban csak annyi feltételezéssel élek, hogy olvasóimnak nem teljesen új az a módszer, amivel *Perl/Tk*-ban egy ablakot létre lehet hozni . Ha ez nem jelent gondot, akkor kattogjon az a klaviatúra, mert máris jön az első lista.

```
#!/usr/bin/perl -w

use strict;
use Tk;
use Tk::HList;

my $foablak = new MainWindow (-title =>
              ➥"Bevasarlolista");

my $lista = $foablak -> HList (-width => 25);
$lista -> pack;

my @vasarlas = ("tej", "kenyer", "szalami");
my $dolog;

foreach $dolog (@vasarlas) {
    $lista -> add ($dolog, -text => $dolog);
}

MainLoop;
```

A szkript három modult használ. A már jól ismert `strict` szigorú ellenőrzési módot ír elő a parancsértelmezőnek. A *Tk* a grafikus eszközkészletet (*ToolKit*) teszi elérhetővé egy jól átgondolt objektum-orientált felületen keresztül. Végül a `Tk::HList` a *Tix* (*Tk Interface Extension*) `HList` nevű elemét biztosítja. Ez a kiegészítés minden *Tk* csomag része, így ennek a használatával jó eséllyel nem veszítünk a felület-függetlenségből.
Első lépésben példányosítjuk a `MainWindow` osztályt, és a konstruktornak egy – jelen esetben egyetlen párból álló – névtelen asszociatív tömböt adunk át. Ezzel úgy hozzuk létre az ablakot, hogy azonnal be is állítjuk a címét. A kapott

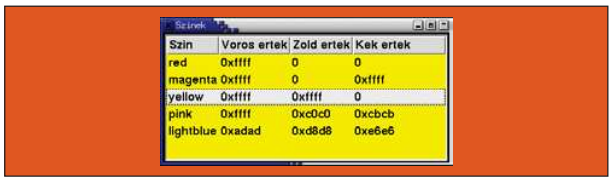



*1. ábra* A bevásárlólista

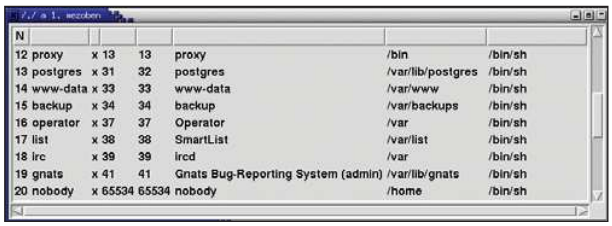



*2. ábra* Bevásárlás több szinten...

`$foablak` objektum `HList` elemfüggvényével hívunk életre egy `HList` ablakelemet, melynek szintén adunk kezdőértéket, ezáltal meghatározzuk a szélességét. Ezután az új, `$lista` objektumra meghívjuk a `pack` elrendezés-kezelőt. Semmi gond nem származik abból, hogy anélkül hívtuk meg a `$lista pack` elemfüggvényét, hogy az akár egyetlen listaelemet is tartalmazna. Az ablak, és minden, ami benne van, csak akkor jelenik meg, amikor a `MainLoop`-ot meghívjuk. A lista feltöltése a `@ennivalok` tömbből fog történni. Egy `foreach` szerkezetben a $dolog változó sorra felveszi ezen tömb elemeit. A ciklus minden lefutásában hozzáad egy új tételt a bevásárlólistához.
Ezen a ponton álljunk meg egy pillanatra. A `HList` egy igen sokoldalú ablakelem, és sokkal többre képes egy egyszerű lista megjelenítésénél. Kialakíthatunk egy hierarchiát az adatelemeink között, és akár egy faszerkezettel is elkápráztathatjuk felhasználóinkat. Az adattagokat a hierarchiában egy egyedi név azonosítja, amely ponttal elválasztva tartalmazza az elem elérési útját. Például az `etel.zoldsegek.paprika` azt jelenti, hogy a paprika a harmadik szinten helyezkedik el, és a közvetlen szülője a `zoldsegek`, ami az `etel` alatt található.
Természetesen nem kötelező kihasználnunk ezt a lehetőséget, tehetjük az összes elemünket egy szintre. Ebben a pél-




*3. ábra* Színes oszlopok




*4. ábra* Szöveges adatbázis megjelenítése

dában pontosan ez történik. Az add elemfüggvény első paramétereként megadjuk az elérési utat, továbbá kulcs-érték párral az elemhez tartozó megjelenítendő szöveget. Ez a két dolog jelen helyzetben ugyanaz, de ez nem szükségszerű. Egy többszintű ábránál szinte biztos, hogy nem is így van. Lássunk erre is egy példát!

```
#!/usr/bin/perl -w

use strict;
use Tk;
use Tk::HList;

my $foablak = new MainWindow (-title =>
              ➥"Bevasarlolista2");

my $lista = $foablak -> HList (
                        -width => 25,
                        -height => 15
                        ) -> pack;

my @vasarlas = (
    ["ital", "nem-alkoholos", "tej"],
    ["etel", "alap", "kenyer"],
    ["etel", "felvagott", "szalami"],
    ["etel", "zoldseg", "paprika"],
    ["etel", "zoldseg", "paradicsom"]
);
my $dolog;

foreach $dolog (@vasarlas) {
  for (my $elem = "", my $i = 0; $i < 3; $i ++) {

      $elem .= "." unless ("" eq $elem);
      $elem .= $$dolog [$i];

        unless ($lista -> infoExists ($elem)) {
          print "Elem: " . $elem . " (" . $$dolog
          ➥[$i] . ") ";
          $lista -> add ($elem, -text => $$dolog
          ➥[$i]);
          print "hozzaadva.\n";
        }

    }
}

MainLoop;
```

A $foablak HList elemfüggvényének meghívásakor egyrészt megadtunk egy további magasság paramétert (height), másrészt a pack-et közvetlenül a létrejövő objektumra hívtuk meg. Mivel a nyíl operátor balról jobbra értékelődik ki, ez az állítás teljesen egyenértékű az eredetivel, és talán egy kicsivel rövidebb. A @vasarlas tömbünk itt már névtelen, három elemű tömbök referenciáit tartalmazza. Ezeket a referenciákat veszi fel sorra a $dolog változó a foreach ciklusban.
Minden egyes lefutásban három lépést teszünk. A második tömbreferencia feldolgozásakor például előbb az etel, majd az etel.alap, végül az etel.alap.kenyer elemet vizsgáljuk. Erre szolgál a belső for ciklus. Minden lépésben az infoExists elemfüggvény segítségével megnézzük, hogy az elem létezik-e, és ha nem, létrehozzuk. Itt látható, hogy az elérési út különbözik a megjelenített névtől. A végén adódó fa egyes ágai nem bezárhatóak. Ha ilyesmire lenne szükséged, nézz utána a Tree elemnek, amely a HList-ből származik.
A következő szkriptben egy több oszlopból listát láthatunk, sőt, még azt is megtanulhatod, hogyan lehet eseménykezelést rendelni az ablakelemhez.

```
#!/usr/bin/perl -w

use strict;
use Tk;
use Tk::HList;

my $foablak = new MainWindow (-title => "Szinek");

my $lista = $foablak -> HList (
                        -columns => 4,
                        -header => 1,
                        -width => 25,
                        -height => 15,
                        -command => \&hatter_beallit
);

$lista -> headerCreate (0, -text => "Szin");
$lista -> headerCreate (1, -text => "Voros ertek");
$lista -> headerCreate (2, -text => "Zold ertek");
$lista -> headerCreate (3, -text => "Kek ertek");

$lista -> pack (
          -expand => 1,
          -fill => "both"
);

my @szinek = ("red", "magenta", "yellow", "pink",
➥"lightblue");
my $szin;
```

```
foreach $szin (@szinek) {

    my ($voros, $zold, $kek) = $foablak -> rgb
                               ➥($szin);

    $lista -> add ($szin);
    $lista -> itemCreate ($szin, 0, -text =>
              ➥$szin);
    $lista -> itemCreate ($szin, 1, -text =>
              ➥sprintf “%#x”, $voros);
    $lista -> itemCreate ($szin, 2, -text =>
              ➥sprintf “%#x”, $zold);
    $lista -> itemCreate ($szin, 3, -text =>
              ➥sprintf “%#x”, $kek);

}

MainLoop;

sub hatter_beallit {
    my ($szin) = @_;
    $lista -> configure (-background => $szin);
}
```

Ez a szkript már valamivel összetettebb az előzőekben bemutatottaknál. Színek egy rögzített halmazát jeleníti meg táblázatos formában. Minden szín vörös, zöld és kék összetevőjét közvetlenül a szín neve mellett 16-os számrendszerben mutatja. Ha a felhasználó duplán kattint valamelyik elemen, a teljes lista az adott háttérszínt veszi fel. A színek közismert angol nevét használjuk a feladat egyszerűsítése végett.
A főablak objektum `HList` elemfüggvényének paraméterezése már közel sem olyan kurta, mint a legelső példában. A `-columns` kulccsal lehetőségünk van az oszlopok számának beállítására. Ez alapértelmezésben 1, ezért hagyhattuk el eddig. Új opció továbbá a `-header`, mely azt határozza meg, hogy legyen-e fejléce az egyes oszlopoknak. Alapértelmezésben 0, vagyis nincs fejléc, ezt bíráljuk felül az 1-es értékkel. Végül a `-command` egy függvényt tartalmaz, melyet a `MainLoop` minden egyes alkalommal meghív, ha bármely elemen kettős kattintás történik.
A fejléc tartalmának létrehozására a lista objektum `headerCreate` elemfüggvénye szolgál. Paraméterezése önmagáért beszél. Először az oszlop sorszámát kell megadnunk, ahol az első oszlop a 0. indexű. Utána egy már jól ismert `-text` kapcsolóval beállíthatjuk a megjelenítendő szöveget. Ezúttal a `pack`-et sem üres paraméterlistával hívtuk meg. A megadott két beállítás hatására az ablakelem önműködően kitölti a főablakot, azaz a felhasználó bárhogy módosítja az ablak méretét, a listánk széltében és hosszában is követi a változtatásokat.
A lista feltöltése ismét egy `foreach` szerkezettel történik, ám most fontos, hogy a `@szinek` tömb szabványos angol színneveket tartalmaz. A ciklus hasának első utasításában ugyanis a tömb sorra következő elemével meghívjuk a főablak `rgb` függvényét. Ez utóbbi a paraméterként kapott szín három összetevőjét tartalmazó tömbbel tér vissza. Így a `$voros`, `$zold` és `$kek` változók a megfelelő értéket kapják.
Ezután az add elemfüggvénnyel csak létrehozzuk az elemet, szöveget még nem rendelünk hozzá. Ezt a feladatot ugyanis az `itemCreate` függvényre hárítjuk, amely egy meglévő listaelem celláinak kitöltésére használható. Első paramétere a listaelem azonosítója, második az oszlop sorszáma, utána pedig bármi, ami az add-nél is használható. C-hez szokott szemeknek nem meglepő, hogy a színösszetevők 16-os számrendszerbeli értékének előállításához a `sprintf` függvényt használtuk.
Végül térjünk ki egy kicsit az eseménykezelésre. Mint említettem, még a `HList` ablakelem létrehozásakor meghatároztunk egy eseménykezelőt a `-command` kapcsolóval. Ezt a `MainLoop` hívja meg amennyiben a felhasználó duplán kattintott. Azon elem azonosítóját, melyen a kattintás történt, paraméterként megkapja a függvényünk. Vagyis a `hatter_beallit` függvény első utasításában a `@_` nevű, paraméterlistát tartalmazó tömb első elemének kinyerésével annak a színnek a nevéhez jutunk, melyre a felhasználó kettőt kattintott. Ezután egy `configure` függvénnyel érvényre juttatjuk a kívánt hatást.
Utolsó példánkban egy általános, szöveges alapú adatbázisok megjelenítésére használható szkriptet szeretnék bemutatni.

```
#!/usr/bin/perl -w

my $hatarolo = “:”;

die “Hasznalat: “ . $0 . “ {kulcs} {regkif}\n”
unless @ARGV == 2;

use strict;
use Tk;
use Tk::HList;

my @talalatok = (); my $mezoszam = 0;

while (<STDIN>) {
    chomp; my @mezok = split $hatarolo;
    if ($mezok[$ARGV[0] - 1] =~ /$ARGV[1]/) {
      push @talalatok, $. . $hatarolo . join
      ➥$hatarolo, @mezok;
      $mezoszam = @mezok if ($mezoszam < @mezok);
    }
}
$mezoszam ++;

my $foablak = new MainWindow (
    -title => “/” . $ARGV[1] . “/ a “ . $ARGV[0] .
    ➥“. mezoben”, -borderwidth => 2

);

my $lista = $foablak -> Scrolled (
    “HList”,
    -scrollbars => “se”,
    -columns => $mezoszam,
    -header => 1,
    -width => 25,
    -height => 15,
    -command => \&duplakatt
);
```

```
$lista -> headerCreate (0, -text => "N");
$lista -> pack (
          -expand => 1,
          -fill => "both"
);

foreach (@talalatok) {
    my @mezok = split $hatarolo;
    $lista -> add ($mezok[0]);
    for (my $i = 0; $i < $mezoszam; $i ++) {
      $lista -> itemCreate ($mezok[0], $i, -text =>
      ➥$mezok[$i]);
    }
}

MainLoop;

sub duplakatt {
    my ($azonosito) = @_;
    for (my $i = 1; ; $i ++) {
      print $lista -> itemCget ($azonosito, $i,
      ➥-text);
      last if ($i == $mezoszam - 1);
      print $hatarolo;
    }
    print $/;
    exit;
}
```

A szkript a határoló karaktert egy változóban tárolja, így az egyetlen sor módosításával változtatható. A szabványos bemenetről olvassa az adatokat, és önállóan meghatározza az oszlopok számát. Az adatsorok közül kizárólag azokat jeleníti meg, melyeknek adott sorszámú mezője illeszkedik a szintén adott szabályos kifejezésre. Az oszlopok sorszámozása 1-től indul. A grafikus megjelenítés után a felhasználó duplán kattinthat bármely elemen. Amennyiben így kiválasztott egyet, az megjelenik a szabványos kimeneten, és véget ér a program.

A `use`-al kezdődő soroktól kezdve nézzük át a program működését. Két fő ciklus köré szervezzük a működést. Előbb beolvassuk az összes adatot egy tömbbe, majd egy következő ciklusban megjelenítjük valamennyi elemet. Az első lépésben két változót töltünk fel: a `@talalatok` tömböt és a `$mezoszam` változót. A `@talalatok` a keresés eredményét tartalmazza, minden találatot kiegészítve egy sorszámmal. A `$mezoszam` a találatok legtöbb mezőt tartalmazó sora mezőinek számát tartalmazza.

A `while` ciklus addig olvas a szabványos bemenetről, amíg talál ott adatot. Minden lefutásban egy sort dolgozunk fel. Először levágjuk a sorvége jelet, majd a határoló karakterek mentén felszeleteljük a sort, és a mezőket kigyűjtjük egy `@mezok` tömbbe. Ezután ellenőrizzük, hogy a rekord kielégíti-e az általunk támasztott feltételeket. A szkript első paramétereként kapott mezőszámot eggyel csökkentjük, és a `@mezok` ezen indexű elemét vizsgáljuk.

Ezáltal ha a paraméter 1 volt, az itt 0-át jelent, ha 2, akkor 1-et, stb. Azaz a felhasználó számára a mezők sorszámozása 1-től indul, és pont ez volt a cél. Tehát a `@mezok` megfelelő elemének kiválasztása után mintaillesztést végzünk rá a második paraméterrel. Ha ez sikeres volt, veremként használva a `@talalatok` tömböt, betesszük a visszaépített sort, az elején kiegészítve a találat helyével. Erre azért van szükség, mert ez biztosan egyedi azonosítója az adott rekordnak.

A `$mezoszam`-ot minden sikeres mintaillesztés után hozzáigazítjuk a mezők számához, amennyiben az szükséges.

A ciklus végén pedig megnöveljük az értéket eggyel, hiszen minden rekordot kiegészítettünk egy, az eredeti állományban elfoglalt sor számát tartalmazó mezővel, így a legszélesebb rekord is biztosan eggyel szélesebb lett. Ezzel beolvastuk az összes adatot, jöhet a megjelenítés.

Először is létrehozzuk a főablakot. A címben jelezzük a szkript paramétereit, vagyis a keresés szempontjából kulcs mező számát és a reguláris kifejezést. Továbbá a `-borderwidth` kapcsolóval meghatározzuk az ablak széle és a hozzá legközelebb eső ablakelem közti távolságot képpontokban. Vagyis jelen esetben az ablakban használt egyetlen elem, a lista és az ablak széle között lesz egy 2 képpont vastag üres keret.

Mivel görgetősávokat is szeretnénk létrehozni, ezúttal nem a `HList` elemfüggvényt hívjuk meg. A `Scrolled` segítségével az áhított görgetősávokkal kiegészített elem jön létre, mindössze azt kell meghatároznunk, hogy hol helyezkedjenek el a sávok. Az elem neve után a `-scrollbars` kapcsolóval égtájak angol nevének kezdőbetűivel adhatjuk meg a pozíciót. Jelen esetben a dél (south) és a kelet (east) beállítással élünk. A további paraméterekkel már korábban megismerkedtünk. Csak az első oszlopnak adunk nevet, ez jelzi az adatok forrásában elfoglalt sor számát.

Ezután egy `foreach` szerkezettel végiglépkedünk a `@talalatok` tömbön. A kevesebb változó használata érdekében az alapértelmezett változót, a `$_`-t használjuk futó változóként, ahogy ezt a `while` ciklusnál is tettük. Minden lefutásban újból szétbontjuk a rekordot, hozzáadjuk a listaelemhez, és kitöltjük a cellákat. Ezáltal a listánk pontosan a keresésnek megfelelő elemekből fog állni.

Végül a `duplakatt` függvény meghatározása következik. Ezen eseménykezelő a szkript élete során jelen esetben csak egyszer fut le, hiszen egy kettős kattintás után a rekordot kiírja a szabványos kimenetre, és kilép. A kiíratásnál már nem jelenítjük meg a korábban felhasznált sorszámot, ezért indul 1-től a `for` ciklus. Az `itemCget` segítségével kinyerjük a megadott cella tartalmát, és az utolsó lefutást leszámítva egy elválasztókarakterrel együtt írjuk ki. Legvégül egy sorvége karakter nyomtatását követően kilépünk.

Remélem, hogy a példák kellően érzékletesek voltak ahhoz, hogy a `HList` nyújtotta lehetőségeket megfelelő részletességgel bemutassam. Kisebb jártasság megszerzése után valóban percek alatt lehet grafikus külsőt készíteni egyszerűbb szkripteknek, s mivel a ***Perlt*** alapvetően szöveges feladatok megoldására készítették, a `HList` igen sok helyen alkalmazható. Sok sikert a további kísérletezéshez, akinek pedig kérdése van, írjon bátran!

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