C Scripting Language avagy a lusta C programozók öröme

A CSL egy a C nyelv után gyorsan megtanulható szkriptnyelv. Egy informatikai rendszerben sokszor adódnak olyan programozással is megoldható feladatok, melyekhez az ember nem szívesen tölt órákat egy hagyományos programozási nyelven kódolva a gép előtt. A szkriptnyelvek csekély kompromisszumok árán segíthetnek ilyenkor – már ha egyszer megbarátkoztunk velük. Akiknek ez még nem sikerült teljes mértékben, azoknak ajánlom a CSL-t, mely csak néhány lépésnyire található a C/C++ stílusú nyelvektől, de azoknál jóval engedékenyebb a programozóval szemben.

A programozási nyelvek közül egyre nagyobb népszerűségnek örvendenek az ún. szkript-nyelvek, de leginkább csak néhány divatossá vált képviselőjüket ismeri a nagyközönség. Titkuk talán – a programozók egészséges lustaságán kívül – abban rejlik, hogy gyorsan és könnyen lehet velük meghatározott kérdésekre frappáns választ adni. Rugalmasságuk ugyanakkor korlátozottabb az úgynevezett compiler típusú (lefordítás után futtatható kódot generáló), általános célú nyelveknél, inkább egy adott környezethez igazodva, rész- vagy célfeladatok megoldásában jeleskednek.

A ***CSL*** nyelvet – mely a ***GNU*** licensz szerint szabadon letölthető és használható a ➲ ***http://csl.sourceforge.net*** címről – elsősorban azok érezhetik szívükhöz közelállóknak, akik már rendelkeznek némi gyakorlattal a ***C/C++*** vagy ***Java*** szintaxisához hasonló nyelvekben, ugyanakkor nem szeretnének egyszerű feladatok megoldására túl sok időt fordítani. E nyelvet használva nem kell azon töprengeniük, vajon szükség van-e explicit típuskonverziókra, meg kell-e semmisíteniük egy használaton kívüli objektumot, hogyan foglaljanak memóriát egy dinamikus tömbnek, és nem okoz fejtörést a kiútkeresés a mutatók erdejéből sem. Utóbbi a szerző állítása szerint nincs is a nyelv kelléktárában – ezzel azért ne értsünk egyet azonnal.

Pingvinek alatt is működő öszvérmegoldás

Ugyanakkor a szkript jellegű nyelvekhez képest jobban ragaszkodik a „klasszikus” programozási szintaktikákhoz, fogásokhoz. Kevésbé szokatlan első látásra a ***C*** vagy ***Pascal*** nyelveken felcseperedett programozóknak, mint az objektumorientáltságban is jeleskedő ***Python*** vagy az elegáns héjprogramok (shellszkriptek); képes kihasználni a reguláris kifejezések erejét – noha nem olyan mértékben, mint például a ***Perl*** – , s az adatbázisokhoz kapcsolódás is megoldható segítségével, még ha ebben tudása nem is fogható ***PHP***-hoz.

Sajnos svájci szerzője 2002-ben abbahagyta a projekt továbbfejlesztését, és a honlapról elérhető fórumon sem tapasztalhatunk újabb aktivitást, viszont eszközkészlete átlátható, és rugalmassága elegendő a legtöbb feladat megoldásához.

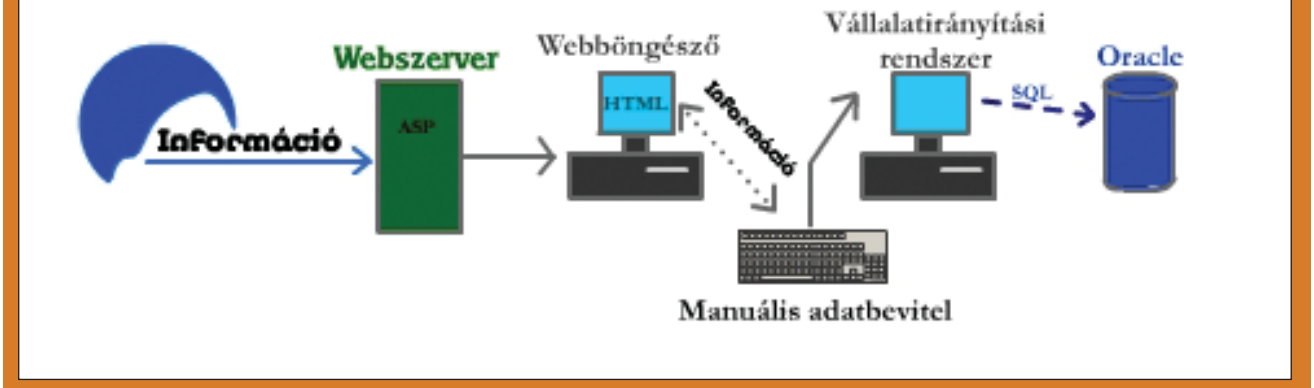
Általános célú programozási nyelv voltát hangsúlyozza a legfontosabb célterületeket lefedő eszközkészlete: sztring- és fájlkezelés, kommunikációs portok, reguláris kifejezések, matematikai műveletek, regisztrációs adatbázist kezelő valamint rendszerfüggvények, és adatbázis-támogatás egyaránt található függvényeinek sorában. Ráadásul ezek többsége egyaránt elérhető a ***Linux/BSD/Unix***, az ***OS/2*** és ***Windows*** környezetekben, hiszen a ***CSL*** ezen platformokon fut. Ráadásul képes egy lefordított típusú nyelvbe makróként beágyazódva is működni; a legfontosabb ***C/C++*** fordítókhoz ***API*** illetve osztály szintű kapcsolódási felületet nyújtva.

Szerezzük meg és használjuk

Telepítése egyszerű és jól dokumentált. A letöltött tömörített fájlt a szokásos módon használhatjuk:

```
tar xzf <fájlnév>
cd csl_verzió_platform
tar -C /usr/local -xf files.tar
ldconfig
```

Ügyeljünk arra, hogy root felhasználói jogokkal kell rendelkeznünk mindezekhez, illetve ha nem az ajánlott ***/usr*** vagy ***user/local*** könyvtárak alá telepítünk, akkor az elérési útvonalat is be kell állítanunk rendszerünk számára.



1. ábra Az információáramlás útja CSL szkript nélkül

Önálló szkripteket így futtathatunk vele:

```
csl  script_neve.csl  [paraméterek_ha_vannak]
```

Kezdő lépéseinket mi is egy *„Helló világ!”* programmal tegyük meg.

```
#loadLibrary 'ZcSysLib'
main()
{
  sysLog("Helló világ!");
}
```

Látható, hogy szintaktikáját nagymértékben a *C* nyelvtől örökölte; az `#include<függvény.h>` szerepét itt általában a fentebb látható függvényhívás tölti be, valamint az elmaradhatatlan `main(){…}` főfüggvény is ismerős. Megjegyzés: ahhoz, hogy a beépítendő fejlécfájlokat használni tudjuk, a `CSLPATH` környezeti változóban a */share/csl* könyvtár elérési útvonalát be kell állítani.

Alapelemek és használatuk

Változók terén viszont inkább az egyszerűsödő, gyorsabb kódolást szem előtt tartó nyelveket idézi: egyedüli változótípusa a `var`, ez tárolhat karakterláncot és számot egyaránt. Ebből következően értelmét vesztené, s ezért nincs is struktúra típus, valamint mutatókkal sem találkozhatunk a nyelvben.
Itt azonban egy kis pontosítást tehetünk, mert függvények hívása esetén a nyelv dokumentációja is kitér arra a tényre, hogy a *CSL* támogatja és javasolja is a cím szerinti értékátadást az & operátor segítségével (`fuggveny(var& x, var& y[])`), sőt, tömbök esetén ez még nyilvánvalóbban látszik. Nem csoda, ha ilyenkor a *C* nyelvben közismerten kezdő tárhelycím szerint, mutatók (pointerek) segítségével is elérhető tömb képe sejlik fel az olvasóban.
Még pontosabban a *C++* használatakor is sűrűn igénybe vett referencia típusú paraméter-átadás történik (a függvény meghívásakor változókat és nem mutatókat adunk meg, míg a függvénydefinícióban az & operátorral tudatjuk, hogy nem a változó értékét, hanem címét használjuk), mely által egyszerűen megváltoztatható a függvényblokkban az adott változó értéke, és nem bonyolódunk bele a mutatókba.

A *C++*-tól eltérően nem a függvénytörzsön belül kell definiálni a `static` kulcsszóval bevezetett globális változókat, és a külső deklarációknak csak akkor van igazán értelme, ha segítségével futási időben – és nem fordításkor – töltünk be változókat.
Amennyiben egy lokális tömböt inicializálunk, úgy az dinamikusan foglal magának tárhelyet (itt érezhető az interpreter jellegű nyelv előnye), míg globális esetben a szokásos statikus helyfoglalás történik. Ezért lehet érvényes a következő kódrészlet *CSL* nyelven, míg *C/C++*-ban nem:

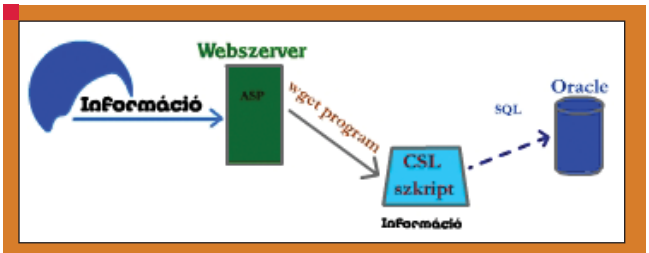
```
var x = 100;
  var tomb[++x];       // a tomb[] 101 elemet
                       // tud tárolni!
  var  masiktomb[x*2]  // a masiktomb[] 202
                       // elemet tárol
```

A dinamikus helyfoglalás előnyét szemlélteti a következő kódrészlet is, ahol a tömb méretét a `resize` utasítással növeljük meg, így már 5 sort tudunk feltöltetni kedvenc betűhármasainkkal:

```
var tomb = {
{'a', 'b', 'c'},
{'d', 'e', 'f''}
}; // eddig a tomb 2 sornyi betűhármast tudott
   // tárolni
resize tomb[5][3]; // ezzel a művelettel már 5
                   // sornyit - dinamikusan nőtt!
```

Való világ

Ennyi bevezető után nézzünk egy életből vett példát!
Cégünk vállalatirányítási rendszere *Oracle* alapokon nyugszik, s egy mezőben képes tárolni a különböző nemzetközi fizetőeszközök és a forint közti váltószámot. Ezt az információt később a kimenő nemzetközi számláknál használjuk, automatikusan átváltva az összeget például euróra, cserébe viszont a váltószámokat valakinek napi rendszerességgel frissítenie kell.
A konkrét feladat röviden tehát így hangzik: tudjuk meg a napi valutaárfolyamot, és „mondjuk meg” egy *Oracle* adatbázis-kezelőnek.



2. ábra CSL szkript segítségével rövidebb az út

Mivel az ***MNB*** honlapján megtalálhatóak a szükséges árfolyamok, a ***CSL*** és egy kis külső segítség igénybevételével a folyamat automatizálható. Nézzük, hogyan tehetjük ezt meg. A példában néha a kitekintés kedvéért lemondok az optimális megoldásról.
Betöltjük a használni kívánt függvényeket:

```
#loadLibrary 'ZcSysLib' //Kiiratásokhoz.
#loadLibrary 'ZcStrLib' //Sztringekhez.
#loadLibrary 'ZcRexLib' //Reguláris
                        //kifejezésekhez.
#loadLibrary 'ZcFileLb' //File műveletekhez.
#loadLibrary 'ZcDaxLib' //Adatbáziskapcsolathoz.
```

Tudjuk, hogy mi a pontos neve annak a ***HTML*** oldalnak, ahol a szükséges információ található. Ezért a linuxos környezetben méltán népszerű ***wget*** program segítségéhez fordulunk a letöltésben – egész egyszerűen elindítjuk kis programunkból a letöltésvezérlőt. A függvény így néz ki:

```
letoltes()
{
  //Ha már létezik a fájl, akkor töröljük, mert
  //különben a szemfüles wget más nevet adna neki.
fileDelete('engine.aspx@page=napiarfolyamok');
  //Letöltjük a html fájlt a (külső) wget program
segítségével:
sysCommand('wget www.mnb.hu/engine.aspx?page=
➥napiarfolyamok');
}
```

Íme egy részlet a letöltött fájlból elrettentésül (ebből kell kinyernünk a napi árfolyamokra vonatkozó információt):

```
<span class="MNB_Heading4">EUR</span></td><td
➥class="MNB_DgBorder" align="Right"><span
➥class="MNB_Heading5">1</span></td><td
➥class="MNB_DgBorder" align="Right"><span
➥class="MNB_Heading5">245,70 </span></td>
```

Könnyű dolgunk lenne, ha például az ***awk*** nyelvet használnánk, hiszen csak azt kellene megmondani neki, hogy mezőhatárolónak melyik ***HTML*** tag-et tekintse, s ugrálhatnánk a mezők között. Mi azonban inkább a mintaillesztést hívtuk segítségül, s a ***C*** nyelvhez hasonlóan fájlból olvasunk be változóba:

```
var mintakereses()
{
  const mit_keres = '\>EUR\<';
  const arfolyam = '[0-9]\{3},[0-9]\{2}';
  ➥//000,00 alakú tizedestörtet keres
  var minta = rexOpen(mit_keres);
  var ar = rexOpen(arfolyam, rexOpenExtended);
  ➥//okosabb regexp
  var talalat[5][2];   //>EUR< minta
                       //megtalálásakor kell.
  var artalalat[3][2]; //Eur/Ft árfolyamát ha
                       //megtaláljuk, ebbe kerül.
```

A reguláris kifejezések a `rexOpen()`, `rexMatch()` és `rexClose()` utasítások segítségével használhatók. Tudjuk, hogy a keresett pénznem `<html_tag>EUR<html_tag>` alakban található a fájlban, s azt is, hogy az árfolyam `000,00` alakú tizedestört. Utóbbira a mintaillesztés csakis olyan számokat keres, melyek egy hármas csoport, majd vessző – ezt jelzi a `[0-9]\{3}` a programkódban – , végül egy kettes csoport szám – ezt pedig a `[0-9]\{2}` – alakjában fordulnak elő. Az `EUR` szót követő első ilyen előfordulás lesz a keresett árfolyam.
Folytassuk a fájl megnyitásával:

```
//Fájl megnyitása.
var megnyit = fileOpen('engine.aspx@page=
➥napiarfolyamok', fileOpenRead
+fileOpenOld);➥//a fileOpenOld módosító azt
jelzi, hogy elvárjuk, létezzen a fájl
//Ellenőrizzük, hogy nem volt-e gond a fájl
➥megnyitásával (ill. tényleg létezik-e):
  if (!megnyit)  //hibakezelés - ezt rábízhatjuk
                 //így egy C++ programra
  {
    const hiba[3] =
    {
'Nyitasi hiba!', 'Hibas, vagy nem letezo fajl.',
fileInfo('engine.aspx@page=napiarfolyamok')
  };
  throw hiba;  //throw itt, try és catch C++
               //oldalon
  }
  //Hibakezelés vége
```

Kivételek márpedig vannak…

A hibakezelést itt most a ***CSL*** kommunikációkészségének bemutatása kedvéért nem „helyben” végezzük, hanem átadjuk azt egy képzeletbeli ***C++*** programrészletnek, mely körülbelül így nézhetne ki:

```
void Kivetelkezelo()
{
  try {
  ZCsl csl;
  csl.loadScript("programom.csl");
  ZString ret = csl.call("kivetelkezelo.sh");
  } // try vége, kivételek figyelése.
  catch (const ZException& err) {
  cerr << err.text;
  } // catch vége
} // Kivetelkezelo vege.
```

Természetesen ugyanez fordítva is lehetséges, egy *C/C++* program által „dobott” kivételt „el tud kapni” a CSL program is. Akik eddig jobbára csak C nyelvet használtak, valószínűleg `if() then() else()` szerkezetek segítségével próbálnák meg lekezelni az esetleges hibákat. A *CSL* ehhez képest a *C++/Java*-ból is ismerős, kulturáltabb és nagyobb programokban is hatékony módszert használja: a kritikus részeket „próbáló” programblokkokban (`try{}` blokkok) „dobunk” egy hibajelzést (`throw()` függvény), ha bajt észlelünk, és ezeket a programblokkot követően akár az összes kritikus rész hibaüzenet dobására figyelve „elkapjuk” (`catch()` függvény). Rövid programunkat azonban most nem tűzdeljük tele – az egyébként erősen ajánlott – kivételkezelésekkel.

Aki eurót keres, >EUR<-t talál

Folytassuk a sort a keresett `>EUR<` mintával:

```
for (var i=1; !fileEof(megnyit) ; i++)
➥//fájl beolvasása kereséshez
  {
  var sor[i] = fileReadLine(megnyit);
  var letezik = rexMatch(minta, sor, 1, talalat);
➥//megtalálható-e a minta
```

A `rexMatch` mintaillesztést végez, a mintát a sztringben keresi (egy előfordulás), s ha talált valamit, azt a `talalt[][]` tömb segítségével tudjuk lokalizálni: megkapjuk, hogy hányadik karakterpozícióban kezdődik a keresett minta, és azt is, hogy milyen hosszúságú:

```
if (letezik)
  {
  sysLog('A keresett ' + mit_keres + ' mintazat
  ➥a fajlban a(z)');
  sysLog(i + '. sorban, a ' + talalat[0][0] + '.
  ➥karakterpoziciotol kezdodik.');
```

Így könnyedén leszűkíthetjük a keresést, hiszen a fájl elejére nem vagyunk már kíváncsiak, csak a mintától a végéig érdekes számunkra:

```
var szukebb = strSubString(sor,talalat[0][0]);
➥//sztringnek az >EUR< -tól kezdődő részét
➥tároljuk
var letezik_ar = rexMatch(ar, szukebb, 1,
➥artalalat);  //ha talál 000,00 alakú  részt
```

Ha megtaláltuk a keresett mintára illeszkedő részt, akkor kiíratjuk:

```
if (letezik_ar)
{
var mettol = artalalat[0][0]; //hányadik
➥karakterpozíciótól kezdődik az ár
var meddig = artalalat[0][1]; /hányadik
➥karakterpozícióig tart az ár
sysLog('\t' + mit_keres + ' arfolyama: ' +
➥strSubString(szukebb,mettol,meddig));
return (strSubString(szukebb,mettol,meddig));
```

```
➥//fájl elejével nem foglalkozunk már
} //if(letezik_ar) vége
else return(0);
} //if(letezik) vége
} //for ciklus (fájl beolvasáshoz kellett) vége
```

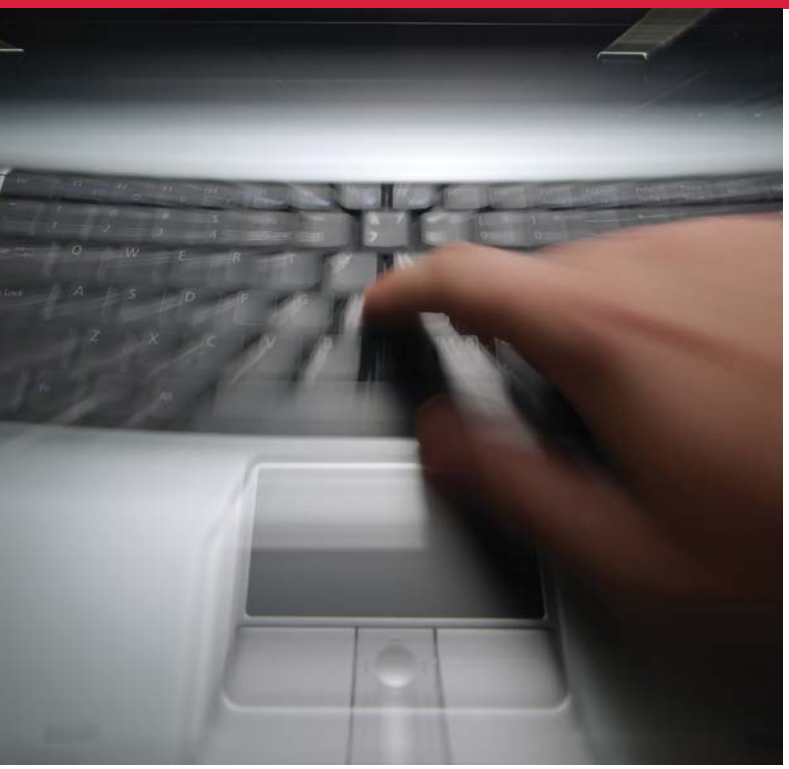
Végül felszabadítjuk a memóriát. Erre azért van szükség, mert az univerzális és gyorsabb használat (több sztringen is végezhetnénk ugyanarra a mintára illeszkedésvizsgálatot) érdekében a mintaillesztőt „lefordítottuk” a `rexOpen` függvény meghívásával.

```
rexClose(minta);       //memória felszabadítása
➥mintaillesztésből
rexClose(ar);  //memória felszabadítása
➥mintaillesztésből
fileClose(megnyit);    //olvasott fájl lezárása
}       //mintakeresés vége
```

Utolsó lépések: adatot az adatbázisba

Az adatbázishoz kapcsolódáshoz a parancssori argumentumokból szerzünk információt, ezért programunkat a következőképp kell elindítani:

```
csl program.csl
➥Felhasznalonev/Jelszo@Adatbazisnev
```



Minden más esetben hibát kapunk, ugyanis ezek feldolgozásában segít minket egy beépített függvény, mely kifejezetten az ilyen alakban elindított, adatbázis-kapcsolatokat is használó programokból képes kivágni a felhasználónevet, s egyúttal a másik két adatot is változókba teheti:

```
var adatbazis, nev, jelszo;
//parancssori argumentumokból "kitaláljuk",
➥melyik a név
nev = strSplitConnectString(mainArgVals[2],
➥jelszo,adatbazis);
```

A csatlakozás előtt át kell alakítani az árfolyamot tartalmazó `arfolyam` változót, hiszen `000,00` alakban kaptuk meg a kívánt adatot, adatbázisunk pedig `000.00` alakban várja tőlünk:

```
arfolyam = strChange(arfolyam,',','.');
➥//Kicseréli a 000,00 alakot 000.00 -ra
```

Majd meghívjuk a kapcsolódást végző függvényünket, átadva neki a szükséges bejelentkezési információkat, valamint a kinyert árfolyamot:

```
kapcsolodas (adatbazis, nev, jelszo, arfolyam);
```

Íme a függvény:

```
kapcsolodas(var& adatbazis, var& nev, var&
➥jelszo, var&arfolyam)
{
var belepes = daxConnect('oracle',adatbazis,
➥nev,jelszo);
```

Leegyszerűsítjük a megoldást azzal, hogy a `kapcsolodas` függvényen belül *SQL* utasítást hajtunk végre, a `belepes` nevű, gyakorlatilag mutató szerepű változóval hivatkozunk az éppen használt adatbázis-kapcsolatra, lekérdezéskor és jóváhagyáskor, valamint lekapcsolódáskor egyaránt:

```
daxSimple(belepes, 'UPDATE PENZUGYEK SET Arfolyam
➥= ' + arfolyam + ' WHERE Penznem LIKE
➥\'EUR%\'');
daxCommit(belepes); //jóváhagyjuk a műveletet
daxDisconnect(belepes); //adatbáziskapcsolat
➥lezárása
} //kapcsolodas függvény vége
```

Mindezt persze a beépített *CSL* függvények segítségével is megoldhattuk volna, hiszen `daxParse()`, `daxCheckCursor()`, `daxFetch()` stb. utasítások segítenek a kurzor pozicionálásában. Itt is fontos lenne a `try-throw-catch` blokk használata kivételkezelés céljából. Aki még nem sokat programozott *Linux* alatt, bizonyára furcsállja az itt is előforduló, látszólag felesleges \ jeleket. Ezek segítségével azt tudatjuk – itt most a *CSL*-lel, de általában a parancsértelmező burokkal -, hogy a \ jelet követő karaktert szó szerint értelmezze, s ne tulajdonítson neki speciális jelentést. Ha ezt elmulasztjuk, az Oracle hibát jelez, mert a programtól nem a kívánt `LIKE 'EUR%'` sztringdarabot kapja meg.
Munkánk eredményét és a *CSL* szkript ténykedését érdemes naplózni, ezt könnyedén meg is tehetjük a main függvényen belül:

```
//Minden tevékenységünket naplózzuk.
  sysDateFormat(sysDateFormatISO); //magyar
                                   //dátumformátum
  sysLogFile( sysDate() + '.log' ); //logfájl
➥neve a mai_dátum.log alakú lesz
```

Tovább bővíthetjük e kis program tudását, ha például a linuxos *cron* segítségével időzítjük a program futtatását, esetleg előtte beágyazzuk egy shellszkriptbe. Így akár komolyabb ellenőrzésekre is lehetőség nyílik, együttműködve például egy *SMTP* szerverrel küldhetünk levelet az illetékeseknek a beimportált árfolyamról naponta, vagy a hibagyanús eseteket (túl nagy vagy túl kicsi érték, kapcsolódás elutasítva stb.) és a naplófájlokat is eljuttathatjuk hozzájuk. Korántsem merítettük ki a nyelv adta lehetőségeket, s kellően elszánt és sok szabadidővel megáldott programozók néhány hatékony szkript és hagyományos program vegyítésével jól működő, komplex feladatok megoldására alkalmas, paraméterezhető szoftvereket hozhatnak létre akár a *CSL* segítségével is. Legfőbb előnyét abban látom, hogy a *C* nyelv ismerete után könnyen tanulható, kényelmesen használható, sokoldalú és hordozható eszköz birtokába kerülünk.

Tóth Virgil Zoltán (m_v@c2.hu)
Szoftverfejlesztő informatikus és rendszergazda, kedvence a Debian disztribúció. Szabadidejét legszívesebben felesége és szépirodalmi regények társaságában tölti. Lenyűgözőnek tartja a Linux rugalmasságát, és a vele dolgozók aktivitását.