

„Ezt egy egysoros programmal meg lehet oldani” Hatékony szkriptnyelvek Unixon 25 éve és ma

Szabó Péter
<szabo.peter@szszi.hu>

Kivonat

A Unix a kezdetektől fogva olyan eszközökkel kényeztette felhasználóit, melyek együtt lehetővé tették az ismétlődő feladatok gyors automatizálását, ezzel a Unixot hatékony munkakörnyezetté téve a programozók, a programozni szerető felhasználók és a rendszeradminisztrátorok számára. Ezen klasszikus eszközök: a csővezetékek és az adatok szöveges, újrabeolvasható tárolási módja és a szkriptnyelvek megtalálhatók a mai Linux rendszereken is, és a a stabilitás és a biztonság mellett ők is hozzájárulnak a Linux népszerűségéhez a programozni szeretők körében.

A cikk egy vázlatos, forráskód-részletekkel illusztrált körképet vázol fel a ma népszerű szkriptnyelvekről. Célja, hogy a Linux iránt érdeklődők ízelítőt kapjanak az automatizálási lehetőségekről, beleértve azt is, hogy milyen feladatnak milyen nyelvben érdemes neki-fogni, továbbá hogy szélesítse azok látókörét, akik néhány szkriptnyelvet már ismernek. A bemutatott nyelvek egy része már 25 éve is jelen volt (Bourne shell, AWK, sed, Make, C shell), de jöttek modern trónkövetelők is (Lua, PHP, Perl, Pike, Python, Ruby, TCL).

A cikkben, az egyes nyelvek szintaxisát illusztrálandó, visszatérő motívumként jelennek meg kettő vagy több különböző nyelven is működő programok, melyek egy gyakorlati hasznára (az interpreter elérési útvonalától független #! shebang) is fény derül.

Tartalomjegyzék

1. Szkriptnyelvekről általában	146
2. Régi nyelvek	149
2.1. Bourne shell: Unix folyamatok integrációja	149
2.2. C shell: a Bourne shell alternatívája	151
2.3. Make: inkrementális szoftverfordítás	152
2.4. sed: stringműveletek	154
2.5. AWK: stringműveletek strukturált programban	155
3. Mai nyelvek	156
3.1. PHP: könnyen elsajátítható webprogramozás	157
3.2. Perl: svájcbicska minden rendszeren	158
3.3. Python: jól olvasható, objektumorientált szkriptek	160
3.4. Ruby: a programozók kedvence	161
3.5. Pike: webprogramozás C-szerű szintaxissal	162
3.6. TCL: grafikus felületeket gyorsan	163
3.7. Lua: letisztult, beágyazott szkriptnyelv	165

1. Szkriptnyelvekről általában

A *szkriptelés* [14] egy olyan szoftverfejlesztési fázist jelent, melyben a már létező, különféle komponenseket kapcsoljuk össze egy új feladat megoldása érdekében, a *szkriptnyelv* pedig egy olyan (általános vagy speciális célú) programozási nyelv, melyen a szkriptelés történik, a *szkript* pedig a szkriptelés során előálló program (forráskódja). A *szkript* eredeti angol megfelelője színpadi szövegkönyvet, forgatókönyvet jelent, utalva arra, hogy az is pontosan leírja, hogy kinek mikor mit kell tennie.

Szkriptelhető lehet például egy stratégiai játék, melynek létező komponensei az egyes épületeket, járműveket és az ellenséges katonákat vezérlő szoftverdarabok, és ezeket szkripteléssel összekapcsoljuk úgy, hogy együtt, összehangoltan megvalósítsák a játék egy pályáját. A Quake 3D lövöldözős játék (FPS) egyike volt az első olyan játékoknak, melynek egy része szkriptnyelven íródott. A fejlesztők dokumentálták, hogyan lehet a játékot szkriptelni, így bárki számára lehetővé vált saját ellenségek, pályák stb. létrehozása. A Quake saját, speciális célú szkriptnyelvet használt, a QuakeC-t. A mai, szkriptelhető játékok körében egyre népszerűbb a Lua beágyazott programozási nyelv, melyről a 3.7. szakaszban részletesebben szólnunk.

Unix rendszereken a szkriptelés főleg ismétlődő feladatok automatizálására használatos. A szkript általában egy összetett feladatot hajt végre, a konfigurációs fájlokban és a parancs-sori paramétereiben megadott paraméterekkel, a létező unixos szoftverek felhasználásával, összekapcsolásával. A Unix-filozófia [17] része, hogy az egyes programoknak olyan szöveges kimenetet kell produkálniuk, melyek nem csak ember számára olvashatóak, hanem könnyen feldolgozhatóak más programmal. Annak idején a Unixokon vált népszerű a csővezeték fogalma. A csővezeték (angolul *pipe*) lehetővé teszi, hogy egy folyamat a kimenetét ne a terminálra vagy fájlba írja, hanem közvetlenül egy másik program bemenetére küldje. Az összekapcsolás folytatható: a másik program is csővezetéken küldi tovább kimenetét egy harmadiknak stb. Csővezeték-hálózatok (angolul *pipeline*) szkriptelés segítségével könnyen kialakíthatók. Megjegyezzük, hogy programokat nem csak csővezetéken lehet összekapcsolni (hanem például kliens–szerver architektúrában), ez azonban Unix alatt körülményesebb, mint az egyszerű csővezeték. Unixon nem csak rendszeradminisztrációs szkriptek vannak, hanem a programozni szerető felhasználók gyakran saját munkájukat automatizálják szkripteléssel.

Egy általánosabb, laza definíció szkriptnyelvnek tart minden olyan programozási nyelvet, melynél fordítás és linkelés nem szükséges (tehát a forráskód elkészítése után közvetlenül futtatható), továbbá a nyelv és a hozzátartozó programkönyvtárak bizonyos feladattípusok programozását jelentősen leegyszerűsítik más nyelvekhez képest. Fő eltérés az előző, komponensintegrációs definícióhoz képest, hogy a laza definíció megengedi egyetlen komponensből álló szoftverek készítését az adott szkriptnyelven.

Cikkünk a Unixon használt legelterjedtebb szkriptnyelveket tárgyalja – ezeken felül is több tucat egyéb, széles körben használt szkriptnyelv létezik Unixra. A tárgyalt nyelvek listája az első verzió kiadásának évével és Wikipedia-beli [19] címszavukkal együtt mutatja az 1. táblázat. Külön meg vannak jelölve azok a nyelvek, melyek csak a laza definícióba férnek bele. A tárgyalt nyelvek mindegyikének létezik szabad szoftver implementációja, és többnyire egyetlen ilyen implementáció terjedt el.

Interpretált nyelvek ♦ Programozási nyelvek között különbséget szokás tenni a forráskód beolvasása szerint. Az ún. *tiszta interpretált* nyelvek esetén (a cikkben tárgyaltak közül a Bourne shell és C shell) egy utasítás beolvasása után az rögtön végrehajtódik, és a következő utasítás beolvasása csak ezután következik. Az egyéb interpretált nyelvek az egész forrásfájlt beolvassák (és esetleg bájtkódot vagy végrehajtási fát építenek), majd a kód memóriabeli, teljesen felépített változatát hajtják végre. A végrehajtás *interpretált*, vagyis a programot a CPU nem közvetlenül hajtja végre, hanem egy olyan segédprogramot, ún. *interpretert* futtat, amely végigmegegy a program utasításainak memóriabeli reprezentációján, és minden utasítás-

1. táblázat. Elterjedt szkriptnyelvek Unixon

25 éve is megvoltak:

Bourne shell	1977–	http://en.wikipedia.org/wiki/Bourne_shell
AWK	1977	http://en.wikipedia.org/wiki/AWK_programming_language
C shell	1979	http://en.wikipedia.org/wiki/C_shell
Make	1977	http://en.wikipedia.org/wiki/Make
sed [†]	1974	http://en.wikipedia.org/wiki/Sed

Újkeletűek, aktív fejlesztés alatt állnak:

Lua	1994–	http://en.wikipedia.org/wiki/Lua_programming_language
PHP [†]	1995–	http://en.wikipedia.org/wiki/PHP
Perl	1987–	http://en.wikipedia.org/wiki/Perl
Pike [†]	1994–	http://en.wikipedia.org/wiki/Pike_programming_language
Python	1990–	http://en.wikipedia.org/wiki/Python_%28programming_language%29
Ruby	1995–	http://en.wikipedia.org/wiki/Ruby_%28programming_language%29
TCL [†]	1988–	http://en.wikipedia.org/wiki/Tcl

[†] komponensintegrációra nem használatos

nál az utasítás típusának megfelelően ágazik el. Ezzel szemben ún. *fordított* (angolul *compiled*) végrehajtás esetén a fordítóprogram (angolul *compiler*) az architektúrának megfelelő, a CPU által közvetlenül végrehajtható kódot állít elő (az ún. *binárist*), a futtatáshoz csak erre van szükség, a forráskódra nem. A fordítás igen időigényes, például a Linux kernel egy mai modern PC-n kb. 15 perc alatt lefordul, apróbb eszközök fordítása is igénybe vehet néhány másodpercet. A cikkben tárgyalt nyelvek egytől egyig interpretáltak. Fordított nyelvre példa a C, a C++ és a Delphi, és fordítás történik C# és a Java esetén is.

Az interpretálás gyorsabb betöltést és lassabb végrehajtást tesz lehetővé, mint a fordítás. Egy interpretált program éles változatának futtatásakor felmerül az igény a nagyobb sebességre. Erre vannak különböző technikák (bájt kód, futás előtti fordítás (JIT), beolvasás után cache-elés), ám a gyorsított végrehajtási sebesség is gyakran elmarad az azonos célú, eleve fordított programok sebességétől.

Sok interpretált nyelvre jellemző (és a legtöbb fordított nyelvre nem) az automatikus memóriafelszabadítás, vagyis hogy a futtatókörnyezet a használaton kívüli memóriát felszabadítja. Ennek tipikus eszközei a hivatkozásszámlálás és a szemétyűjtés. Az automatikus memóriakezelés egyszerűsíti a programkódot, de a plusz adminisztráció miatt lassítja a futást.

Nagy szoftverek szkriptben ♦ A szkriptnyelvek tehát több szempontból segítik a gyors szoftverfejlesztést (pl. jól használható programkönyvtárak, speciális célú nyelvi elemek, a gyors újraindítás az interpretáltság miatt). Vigyázni kell azonban, mert ha a nagy kódolási sietségben nem figyelünk a forráskód minőségére, egy bizonyos méret fölött a kód érthetlenné, karbantarthatatlanná válik. A szkriptnyelveket sok kritika éri, hogy könnyű bennük rossz minőségű kódot írni. E sorok írója (a tárgyalt szkriptnyelvek közül) 1000 sor fölött nem ajánlja a Bourne shellt, a C shellt és a sedet, 10000 sor fölött pedig az AWK-t, a Make-et és a TCL-t. A többi nyelv (Lua, PHP, Perl, Pike, Python és Ruby) megfelelő eszközöket kínál nagy szoftverek fejlesztéséhez. A lehetőség persze nem garancia: különösen Perlben, PHP-ban és Lua-ban kell tudatosan lassítani és odafigyelni kódolás közben – Python, Ruby és Pike esetén általában természetesebben, kevesebb odafigyeléssel készül a jó minőségű kód. A legnagyobb Perl-modulgyűjteménybe, a CPAN-be [5] kerülő modulok legtöbbje ékes példája a jó minőségű Perl-kódnak.

Egysoros szkriptek ♦ A cikk címében szereplő felkiáltás („ezt egy egysoros programmal meg lehet oldani!”) lehetne akár a tapasztalt, hatékonyan dolgozó Unix-felhasználók jelmon-

data. Ők ugyanis a napi feladataik során nemcsak gyakran használnak, hanem gyakran írnak is szkripteket, sokszor igen rövid, 80 karakteren elérő egysorosakat, mindig a megfelelő szkriptnyelv(ek)en. Vegyük például azt a feladatot, hogy meg kell találni a legtöbb helyet foglaló felhasználót. Ez az alábbi paranccsal megoldható¹:

```
$ du -sk /home/* | sort -rn | less
```

A fenti parancs a felhasználókat a *home* könyvtárak méretének csökkenő sorrendjében jeleníti meg. A használt szkriptnyelv a Bourne Shell, amely elvégzi a `/home/*` kifejtését, és három programból álló csővezeték-hálózatot hoz létre (a parancsbeli `|` karakterek mentén). Elképzelhető azonban, hogy egyes felhasználók *home* könyvtárai nem közvetlenül a `/home`-on belül található (pl. `/home/tanar/hannibal`). A javított megoldás, ami a `/home` alkönyvtáraiban található *home* könyvtárakat is megjeleníti, a következő:

```
$ </etc/passwd awk -F: '$6~/^\/home\/\/{print$6}' |
  xargs du -sk | sort -rn | less
```

(A parancs épp 80 karakter lett.) Ha a felhasználói adatok NIS-adatbázisban találhatóak, akkor `</etc/passwd` helyett `ypcat passwd |` írandó. Tárolásfüggetlen megoldás is létezik, itt `„getent passwd |”`-re kell cserélni. Ha a `getent` parancs nem elérhető, akkor a megoldás Perlben:

```
$ perl -le 'while(@L=getpwent){print$L[7]if$L[7]=~/^\/home\/\}' |
  xargs du -sk | sort -rn | less
```

A fenti három megoldás jól példázza a unixos gyors problémamegoldás menetét. A gyakorlott felhasználó ismeri a használható eszközöket (`du`, `sort`, `less`, `xargs` stb.) és a szkriptnyelveket (Bourne Shell, AWK, Perl stb.), és töprengés nélkül, folyamatosan írja be a csővezeték-hálózatot létrehozó shell-parancsot. (Esetleg a manban utánanézés a programok kapcsolóinak.) Majd, ha a parancs nem pont azt csinálja, amit kéne, akkor módosít rajta (pl. szükség esetén áttér AWK-ról Perlre).

E cikknek nem célja, hogy minden kódrészletet teljes mértékben megmagyarázzon. Ha valaki a régebbi eszközök (Bourne shell, AWK, Make, sed) tömör, velős tárgyalására kíváncsi, olvassa el a klasszikusnak számító [9]-et, vagy egyszerűen Linux alatt olvassa el a megfelelő kézikönyvdalt (angolul *man page*): *bash(1)*, *mawk(1)*, *sed(1)*, Make esetén pedig az Info oldalt (`info make` paranccsal). A modern szkriptnyelvek mindegyikéhez van tutorial, megtekinthető az adott nyelv honlapján.

Kívülállóak elfogadhatatlanul nagyok találhatják azt az erőfeszítést, amivel el lehet jutni arra a tudás- és készségszintre, hogy a fentiekhez hasonló egysoros parancsok gépelése folyékonyan történjen. Alternatív megoldásnak javasolhatják a grafikus menedzsment-felületek használatát (pl. Vezérlőpult), kattintgatni végül is sokkal kevesebb tudással is lehet. Vannak azonban hátrányai is a grafikus felületen történő munkának:

- Egy grafikus felület felhasználója nem élvez teljes szabadságot: elképzelhető például, hogy az ő felületén nincs olyan funkció, mellyel a használt tárterület csökkenő sorrendjében fel lehet sorolni a felhasználókat. Ekkor vagy egyenként, kézzel végignézi az összes felhasználót (sok, értelmetlenül elvesztegetett idő), vagy lekérdez minden adatot a felhasználókról, majd egy grafikus táblázatkezelővel feldolgozza az eredménytáblát (még ez is jóval lassabb a szkriptelésnél).
- Mennyire növekszik az idő, ha ugyanezt az információt távolról, lassú kapcsolaton kell megszerezni? A távoli grafikus felületen való kattintgatás órákig is eltarthat, míg pl. a szkriptek könnyedén futtathatók SSH-n keresztül.

¹ a kezdő dollárjel a promptot jelöli – ne gépeljük be

- Ha a vezetőség naponta jelentést kér a foglalt tárterületről, akkor a grafikus felületen dolgozó alkalmazottra ez minden nap plusz terhet ró, míg a Unix alatt szkriptelő rutinos felhasználó 1 perc alatt megírja a szkriptet, és még 5 perc, amíg létrehozza a cron jobot, ami gondoskodik a szkript esténkénti lefuttatásáról, és az eredmény e-mailes elküldéséről a vezetőség számára.
- Ha az adott grafikus felületnek új verziója jelenik meg, sok minden máshol lesz rajta, az alkalmazói tudás egy része elavul, a szokásos munkamenetet újra és újra ki kell dolgozni. A unixos eszközök és a Unix alatt használatos programnyelvek viszonylag lassan változnak: az 1975-ben írt shell-szkript kevés változtatással ma is futtatható (a `du`, `xargs` és társai már akkor is megvoltak), és az 1995-ben írt Perl-szkript is szinte ugyanúgy működik ma is, mint írásakor. A szkripteket író Unix-felhasználó tehát számíthat arra, hogy az egyszer megszerzett tudásának, készségeinek évtizedek múlva is hasznát veszi, és nem kell évente felülrírnia őket új ismeretekkel. Továbbá lehetősége van lassan, fokozatosan tanulni, a Unix használatának egyre hatékonyabb módjait elsajátítani.

2. Régi nyelvek

A régi nyelvek közös tulajdonsága, hogy az 1970-es évek Unix rendszereihez fejlesztették őket, egyszerűek, manapság is elérhetők szinte minden Unixon, és a sok implementációnak van egy széles körben elfogadott metszete, egy résznyelv, mely minden implementációban azonosan működik.

2.1. Bourne shell: Unix folyamatok integrációja

A *shell* (magyar fordításban *burok* vagy *hég*) az a program, amely bejelentkezéskor és terminálablak nyitáskor elindul, feladata a felhasználói parancsok fogadása (a sor szerkesztése és a korábban kiadott parancsok visszaadása), a parancsok végrehajtása (átírányítás, fájlnevek kiegészítése `*`-ra és `?`-re, csővezetékek létrehozása, külső program(ok) futtatása), és az indított programok vezérlése (angolul *job control*; háttérbe rakás, előtérben futó program kiválasztása).

Egy egyszerű parancs kiadásakor a shell a szóközők mentén argumentumokra bontja, az első argumentumot a program nevének veszi, és a programot elindítja (szükség szerint megkeresve a `$PATH`-on), átadva neki a további argumentumokat. Vannak a shellnek belső parancsai is (pl. `echo`, ami kiírja argumentumait), ezeket ő maga hajtja végre külső parancs meghívása nélkül. Az elindított program alapértelmezésben a terminálról ír és onnan olvas, és a hibaüzeneteket is a terminálra írja. A program által kilépéskor visszaadott státusz kód a shelltől lekérdezhető. A shellben lehetőség van ezek átírányítására, például a Bash shell a `foo <be >>ki 2>/dev/null` parancs hatására a `foo` programot futtatja úgy, hogy az a `be` fájlból olvasson, a `ki` fájl végére fűzze hozzá kimenetét, és hibákat a `/dev/null`-ra írja (amit a Unix eldob). A shell ezen felül még kiegészíti (angolul *globbing*) a fájlneveket, így például a `foo ?a*` parancsot futtatás előtt kifejtű `foo bar ba tart-tá`, ha az aktuális könyvtárban `bar`, `ba` és `tart` azok a fájlok, melyeknek második betűje `a`.

A shell nem csak egyszerű parancsot fogad el, hanem ezekből `|` karakterrel összefűzött csővezeték-hálózatot is (és az egyes programok elindításáról, ki- és bemenetük összekapcsolásáról ő gondoskodik), továbbá vannak vezérlőszerkezetek (pl. `if`, `while`), és egyes shellekben függvényeket írhatunk (a függvény egyetlen státusz kódot adhat vissza, ami nemnegatív egész szám). A shell kezel string típusú változókat, melyek parancsok építéskor (általában `$` után) behelyettesítődnek. Az újabb shellek (pl. Bash), ellentétben az eredeti Bourne shelllel, egydimenziós tömböket is kezelnek, és egész számokkal is tudnak számolni.

Az első Bourne shell a kezdeti unixos shellekből leszűrt tapasztalatok után keletkezett 1977-

ben. A mai Linux rendszerek alapértelmezett és egyben legelterjedtebb shellje, a Bash felülről kompatibilis az eredeti Bourne shelllel. Az újabb shelltípusok (pl. Korn shell és Zsh) is ilyenek, de egymástól is vettek át nyelvi elemeket. Az ash egy apró, az eredeti Bourne shell szolgáltatásait megvalósító shell, interaktív parancsbevitelt kényelmessé tevő rutinok nélkül – ideális telepítőkészletek, RAM-ból futó beágyazott rendszerek shell-szkripteléséhez (pl. BusyBoxban [4]).

Mint említettük, a Bourne shell tiszta interpretált. Emiatt egy hosszan futó, fejlesztés alatt álló szkript futása közben előfordulhat, hogy módosul a forrásfájl, és a shell a következő utasítást már az új fájlból akarja venni. Ez általában nem kívánatos. Megelőzhetjük, ha a szkript érdemi részét körbe vesszük `if true; then ... exit; fi`-vel.

Stringkezelés ♦ Az összefűzésen kívül az összes többi stringkezelő művelet shellben igen keserves, például egy olyan eljárást, ami a `V` változóban az `aaa` stringet felváltva `bbb`-re és `ccc`-re cseréli, túl körülményes csak belső parancsokkal megírni. Az alábbi megoldás nem működik minden Bourne shellben, de megy Bash-ben, Zsh-ban és pdksh-ban:

```
repaaa() {
    local X="$V" Y=bbb Z="{V#*aaa}"; V=""
    while [ "$Z" != "$X" ]; do
        V="$V${X%aaa}$Z"$Y"
        if [ "$Y" = bbb ]; then Y=ccc; else Y=bbb; fi
        X="$Z"; Z="{X#*aaa}"
    done
    V="$V$X"
}
```

Ugyanez Perlben sokkal rövidebb, gyorsabb, és jóval kevesebb gondolkodással előáll:

```
sub repaaa() { my $C=1; $V=~s/aaa/($C^=1)?"bbb":"ccc"/ge }
```

A Bourne shell tehát remek eszköz átirányításra, valamint külső programok integrálásra csővezeték-hálózatokkal és egyszerű vezérlessel, ám nehézkessé válik, amikor stringkezelésre vagy számolásra kerül sor. További probléma, hogy a shell-szkriptek igen lassúak. A fenti problémás esetekben érdemes áttérni pl. Perlre; Perlben ugyanis – a megfelelő kiegészítő modulokkal – minden feladatot meg lehet oldani, és csak súlyos CPU- vagy memóriaínség esetén kell Perlről valamely fordított nyelvre váltani. (Ízlés szerint Perl helyett egyéb modern szkriptnyelvvvel is dolgozhatunk.)

Kalandoz kedvűek kipróbálhatják a Bash helyett a Zsh nevű shellt, amely számos előnyét ötvözi a Bourne, a C és a Korn shellnek. A legfontosabb előnyei a Bash-sel szemben: interaktív lehetőségei (prompt, fájlnev-kiegészítés) jobban testreszabhatók, egyszerre több kimeneti fájlba is át tud irányítani, továbbá a `**` mintával az alkönyvtárakat (rekurzívan) is ki tudja egészíteni. Mindezek olyan szolgáltatások, melyek kényelmesebbé, hatékonyabbá teszik a mindennapi munkát. A Zsh hátránya például, hogy csak a legújabb verziók működnek jól UTF-8-as terminálon.

Shell-szkriptek írása ♦ A shell használata esetén szinte észrevétlenül válik a felhasználó programozóvá. Például a 148. oldalon található listázó parancsot (amely a home könyvtárbeli helyfoglalást szerint rendez), könnyen szkriptté alakíthatjuk:

```
#!/bin/bash --
# Használat: listaz.sh [<hossz>]
# a foglalt terület szerinti csökkenő sorrendben listázza ki a
# felhasználók home könyvtárait. Csak az első <hossz> db könyvtárat
# mutatja (vagy 20-at, ha nincs <hossz>).
```

```
</etc/passwd awk -F: '$6~/^\home\/\{print$6}' | xargs du -sk |
  sort -rn | head -"${1:-20}"
```

A szkriptet futtathatóvá tétele (`chmod +x listaz.sh`) után a `./listaz.sh 3` paranccsal próbálhatjuk ki.

A Bourne shell a `#` jeltől figyelmen kívül hagyja a sorok tartalmát, így megjegyzés helyezhető el. A `"${1:-20}"` kifejezés pedig a szkript első argumentumát (`$1`) adja vissza, vagy annak hiánya (vagy üressége) esetén `20`-at.

A fájl első, `#!`-lel (az ún. *shebang*-gel) kezdődő sorát a Bourne shell a `#` jel miatt figyelmen kívül hagyja, ám ez a sor nem fölösleges: a kernel számára azt írja elő, hogy az adott fájl futtatásához egy külső programot kell indítani – tehát a `./listaz.sh 3` parancs helyett `/bin/bash -- ./listaz.sh 3` fog lefutni, vagyis a Bash interpreter fogja futtatni a `./listaz.sh` shell-szkriptet az egyelemű 3 argumentumlistával. Ez pont az, amit szeretnénk.

Rendszerműködtető shell-szkriptek Linuxon ♦

- `/etc/init.d/*`: a szolgáltatásokat indító és leállító szkriptek,
- `*.ebuild`: Gentoo disztribúcióban a csomagkészítést és -fordítást vezérlő szkriptek (Bash alapú, saját keretrendszerrel).
- `preinst`, `postrm` stb.: Debian-alapú disztribúciók csomagjainak felrakásakor és leszedésekor lefutó szkriptek.
- `/etc/cron.*/*`: a cron által periodikusan (naponta, hetente stb.) futtatott rendszerkarbantartó szkriptek.
- A 2.4-es Linux kernelben a `make menuconfig`-ra lefutó kernelfordítás előtti konfiguráló program is shell-szkript volt.
- A Debian telepítője is sok egyedi shell-szkriptet tartalmaz.
- A Knoppix Linux LiveCD indulásakor shell-szkriptek vezérlik a hardverfelismerést stb.

GNU-s szabad szoftverek lefordításában is sok shell-szkript vesz részt (AutoConf, AutoMake, LibTool, lásd még [7]), a fordítás előtt lefutó `configure` szkript is Bourne shell-szkript: őt az AutoConf generálja a `configure.in` és egyéb fájlokból M4 makrónyelv felhasználásával.

2.2. C shell: a Bourne shell alternatívája

A C shell a Bourne shell egyik elődjéből fejlődött ki, mert akkoriban az elődből számos kényelmi elem hiányzott: indított programok vezérlése, parancstörténet (angolul *command history*), korábbi parancsok felhasználása (angolul *history substitution*), tömbkezelés, home könyvtárak kifejtése `~`-re, álnevek (angolul *alias*), számolás, fájlnev-kiegészítés *Tab*-bal. Ezek manapság a Bash-ben mind megvannak, ezért a C shell választása nem jár számottevő előnnyel.

Unixra a legerterjedtebb, továbbfejlesztett implementáció a `tcsh`.

A C shell a C nyelvről kapta a nevét, mert szintaxisa a C nyelvéhez hasonló (ami kicsit könnyebben olvasható, mint a Bourne shell-szkriptek) Ezért az apró esztétikai előnyért azonban nagy árat fizetett: nem tudja futtatni se a Bourne shellre, se a Bourne shell elődjére írt szkripteket. A C shell tehát megosztja a shell-szkriptek fejlesztőit: vagy Bourne shellre fejlesztenek, ami minden Unixon futtatható, vagy C shellre, ami sokkal kevésbé ismert és

használatos (manapság a legtöbb Linux-disztribúció nem rak fel alapból C shell implementációt). Emiatt többnyire Bourne shell-szkriptek születnek.

A C shell szintaxisában levő precedencia miatt kritika érte. Például az

```
if ( ! -e foo ) echo bar >foo
```

utasításban a >foo átirányítás akkor is végrehajtodik (üres fájlt hozva létre), ha a feltétel nem teljesül.

C shell és Bourne shell szétválasztása ♦ Manapság a /bin/sh általában egy Bourne shellre mutat (Linux alatt a Bash-re, beágyazott Unixokon általában a BusyBox-ba [4] épített ash-ra). Csak évtizedekkel ezelőtt volt szokás egyes rendszereken C shellt berakni /bin/sh-nak. Ma már nem szükséges tehát, hogy a #! /bin/sh shebanggel kezdődő shell szkriptek felkészüljenek, hogy esetleg Bourne shell helyett C shellben futnak.

Am – a lényesen eltérő szintaxis ellenére – kis trükközéssel elérhető, hogy ugyanaz a szkript fusson mindkét shelltípusban. Persze csak a lelegeje fut mindkettőben: a kód egy elágazással kezdődik, amely vagy a csak Bourne shellben, vagy a csak C shellben működő részre ugrik. A helyzetet bonyolítja, hogy már az elágazó utasítás (if) szintaxisa is lényegesen különbözik a két shelltípusban. Azért van megoldás:

```
eval '(exit $?0)' && eval '#\
echo Ez csak Bourne-kompatibilis shellben, pl. ash, bash, ksh, zsh; exit'
echo Ez csak C shellben
```

A szétválasztás azon alapul, hogy a \$?0 C shell esetén 1-et ad vissza, míg Bourne shell esetén 00-t (feltéve, hogy a legutoljára befejeződött parancs státuszkódja 0). A többi utasítás (eval és exit) és szintaktikai elem mindkét shellben ugyanazt csinálja.

2.3. Make: inkrementális szoftverfordítás

A Make nyelv több, egymástól függő műveletre bontható munka leírását teszi lehetővé. Ilyen munka például egy szoftver lefordítása (fordítás: forrásfájlokból objektumfájlok készítése, majd linkelés: az objektumfájlokból bináris linkelése, majd a dokumentáció generálása) vagy egy L^AT_EX-dokumentum lefordítása (pl. forrás → DVI → PostScript). A leírás alapján a Make program (Linux rendszereken általában a GNU Make) az egyes műveleteket automatikusan sorrendezi: ha egy *B* művelet függ az *A* művelettől, akkor a sorrendben *A* meg fogja előzni *B*-t. Ezután a Make a kialakított sorrendben végrehajtja a műveleteket. A Make inkrementálisan dolgozik: ha a munka elvégzése után néhány fájl megváltozik (melyektől függ a munka eredménye), és újrafuttatjuk a Make-et, akkor csak azokat a műveleteket hajtja végre, melyekre a megváltozott fájlok hatással vannak. Ily módon egy szoftverfejlesztés (módosítás, lefordítás, kipróbálás) során lerövidíthető a fordítással töltött idő, ha azt Make-vel végezzük: ugyanis csak a megváltozott fájlok (és a tőlük függő esetleges egyéb fájlok) kerülnek újrafordításra. Az időnyereség manapság is számottevő: nem mindegy, hogy 500 db C++ fájlból 1-et vagy 500-at kell újrafordítani, ha egy fájl fordítása 2 másodpercig tart.

A Make által végrehajtható műveleteket szabályok írják le. Példa szabályra:

```
# A harmadik sort tabulátorral kell kezdeni.
foo.o: foo.c foo.h bar.h
    gcc -c -o foo.o foo.c
```

A fenti szabály jelentése: ha a foo.o fájl nem létezik, vagy régebbi, mint a foo.c, foo.h és bar.h fájlok bármelyike, akkor a gcc -c -o foo.o foo.c shell-paranccsal készíten-dő el e fájlokból a foo.o új változata.

A Make-ért szerzője 2003-ban *ACM Software System Award* díjban részesült [1].

A GNU Make implementáció számos kiegészítő szolgáltatást nyújt: string típusú változók kezelése, stringműveletek, preprocesszor (a Makefile egyes sorainak feltételes kihagyása), függvény külső parancs hívására (a parancs kimenetét stringként adja vissza), szabály az összes adott kiterjesztésű fájlra, változók felülbírállása parancssorból stb.

A Make fontos szerepet kap a GNU szoftverek fordításában [7]. Egy GNU szoftver fordítása így történik:

1. Az első lépés a forráskód letöltésére és kicsomagolása.
2. A `./configure` parancsot kell futtatni (szükség esetén a fordítást befolyásoló argumentumokkal, például a telepítési célkönyvtárral és a használandó programkönyvtárakkal kiegészítve), amely nagy vonalakban ezt teszi: detektálja a rendszer néhány jellemzőjét (melyik függvény melyik `.h` fájlban és melyik programkönyvtárban érhető el), felderíti a függőséget a fordításhoz szükséges forrásfájlok között, majd legenerálja a `Makefile.in`-ből a `Makefile`-t (a Make számára) és a `config.h.in`-ből a `config.h`-t (a C fordító számára).
3. A `make` parancs a `Makefile`-ban található szabályokat követve levezényli a szoftver lefordítását.
4. A rootként kiadott `make install` parancs telepíti (felmásolja) a lefordított szoftvert.

A Make-et használják még a Linux kernel fordítására, a NIS felhasználó-adatbázis frissítésére, Debian-csomagoknál a program fordítására és a csomag különböző fájljainak elkészítésére is. \LaTeX -dokumentumok Make-vel automatizált fordítására nincs általánosan elfogadott séma, a szerzők szükség esetén egyedi `Makefile`-okat írnak.

Nem véletlen, hogy a függőségek felderítését nem a Make, hanem a `configure` szkript végzi. Erre ugyanis a Make nem képes. További hátránya a Make-nek, hogy körkörös függőségek esetén hibát jelez (pl. \LaTeX -dokumentumok többmenetes fordításakor tipikus a körkörös függőség), és nem támogatja az elosztott fordítást sem (vagyis nem lehetséges egy nagy szoftvert több gépen, párhuzamosítva fordítani).

A fenti problémák orvoslására számos szoftver született [15], például `dmake` (elosztott fordítást támogat, az OpenOffice.org fordításához használják), `Ant` (Javaban, főleg Java-programok fordítására), `SCons` (Pythonban), `PBS` (Perlben), `Rake` (Rubyban).

Hordozható Make shebang ♦ A Make alapesetben az aktuális könyvtár `Makefile`-jából olvassa a szabályokat. Ezt `-f` kapcsolóval bírálhatjuk felül. A `#!/usr/bin/make -f shebang` segítségével elérhető, hogy a Make-szabályokat tartalmazó fájl közvetlenül futtatható legyen. De mit tegyünk, ha a `make` parancs elérési útját nem ismerjük (lehet például `/usr/local/bin/make` is)? Futtassuk a szkriptet `#!/bin/sh shebang`-gel, és bízzuk a shellre az elérési útvonal megtalálást. Igen ám, de ekkor úgy kell a fájl első néhány sorát megírunk, hogy shell-szkriptként és Make-szabályfájlként is működjön. Íme a megoldás, kezdjük így a szabályfájlt:

```
#!/bin/sh
#\
eval '(exit $?0)'&&eval '\
M=make;>/dev/null type -p gmake&&M=gmake;exec "$M" -f "$0" "${1+"$@"}"; \
>/dev/null which gmake&&exec gmake -f "$0" $argv;q;exec make -f "$0" $argv;q
# Don't touch lines 1--6: http://www.inf.bme.hu/~pts/Magic.Make.Header
```

A fenti megoldás először a `gmake` parancsot keresi (hasznos például FreeBSD-n, ahol a `make` parancs a BSD Make-jét futtatja, a `gmake` pedig a GNU Make-et), majd a `make`-et.

Működik Bourne és C shellben is (a már ismert `$?0`-s trükkel választva szét a két shellt). A shell és a Make szétválasztásának alapötlete az, hogy a `#\`-t tartalmazó sor shellben egysoros megjegyzés, Make-ben viszont a következő sorban folytatódó megjegyzés. Tehát a Make a fenti egész fejléceket megjegyzésnek tekinti, figyelmen kívül hagyja.

Megjegyezzük, hogy azért kell Bourne és C shell esetén más kódot futtatnia a fejlécnek, mert a kapott argumentumok továbbadása máshogy történik (Bourne shell esetén a hordozható megoldás a `${1+"$@"}`, C shell esetén pedig a `$argv:q`).

Van egy rövidebb, a `env` programot használó megoldás is, amivel egy futtatható szkriptfájl elkezdhető. Bash-szkript esetén például a `#!/usr/bin/env bash` sorral kezdve a szkriptet biztosak lehetünk benne, hogy a `bash` program fogja végrehajtani, bárhol is legyen a `$PATH`-on – feltéve, hogy van `env` program a `/usr/bin`-ben. (Ez például modern Linux, FreeBSD és Solaris rendszerekre igaz is.) Ugyanez a trükk Make esetén `#!/usr/bin/env make -f shebang`-sорт eredményezne, ami nem működne, mert a kernel az argumentumokat nem vágja fel szóközök mentén, tehát a `make_ - f` parancsot próbálná elindítani, ahelyett, hogy a `make` parancsot futtatná az `- f` kapcsolóval. Hátrány még, hogy `gmake`-kel nem is próbálkozik.

2.4. sed: stringműveletek

A `sed`-et az 1970-es években az igény hívta életre, hogy a shell nem kínált elég hatékony eszközöket szövegfeldolgozásra. Később ez a helyzet javult, ám a mai shellekben is kifejezetten kényelmetlen egy string belsejéből adatot kinyerni. A Unix-filozófiát [17] követve nem a shellt bővítették, hanem egy céleszközt fejlesztettek ki.

Az `sed` általános szövegfeldolgozási modellje a következő. A `sed`-szkript egy szabálylista. Minden szabály egy feltételből és a feltétel teljesülésekor végrehajtandó (esetleg összetett) utasításból áll, melyek módosíthatják az adott sort (és használhatnak változókat, melyek értéke két sor közt megmarad). A `sed` a bemenetet soronként olvassa, minden sorra végigmegy a szabálylistán, és az eredményül kapott sort kiírja a kimenetére. Ugyanezt a modellt követi az `AWK`, továbbá a `Perl`, a `Ruby` és `PHP` is meghívható olyan kapcsolókkal, hogy a kapott szkriptet soronként dolgozza fel.

A `sed`-et shell-szkriptekből szokás hívni, a `sed`-szkriptet parancsargumentumban megadva. Például az alábbi Bourne shell-részlet a `MACI` változóban cseréli az összes barnamedvét és jegesmedvét mosómedvére:

```
MACI=" `echo "$MACI" | sed 's/(barna|jeges)\(medve\) /mosó\2/g' `"
```

Hasonló bonyolultságú parancsokkal lehet a shell-szkriptben fájlneveket átalakítani, pl. a kiterjesztésüket megváltoztatni.

A `sed` leghatékonyabb eszköze, a reguláris kifejezés mind szűrésre (a feltételrészben), mind cserére (az utasításrészben), mind csere utáni feltételes elágaztatásra használható. További eszközök: karaktercsere, string másolása az aktuális sorból az egyetlen változóba és vissza, elágaztatás, sorszám-intervallumok megadása a feltételben.

A `sed` korlátai: nem lehet benne számolni, az aktuális soron kívül csak egy változója van, csak soronként tudja a bemenetet feldolgozni, nem támogatja a strukturált programozást (ciklus, függvény), a szintaxisa nehezen olvasható. Shell-szkriptből hívva további hátrány, hogy lassú, mert minden stringkezelő művelethez új `sed` folyamatot kell indítani.

Az ismert korlátok ellenére a `sed` azért van még ma is használatban, mert biztosan lehet számítani rá, hogy minden Unixon fent van, és a szabványos utasításai ugyanúgy működnek benne mindenhol. Így tehát a `GNU AutoTools configure` shell-szkriptje is sok helyen használ `sed`-et. Bonyolultabb szövegfeldolgozási műveletekre a minden Unixon jelen levő eszközök közül inkább az `AWK` ajánlott (lásd a 2.5. szakaszban).

Bár a `sed` nem tud számolni, reguláris kifejezéseit vezérlési szerkezeteivel kombinálva írható benne számológép (ami például a kisiskolás módon, számjegyenként ad össze és szo-

roz). Az elvi lehetőséget illusztrálандó egy ilyen számológép készült 1996-ban, ami a dc fordított lengyel sorrendű, tetszőleges pontosságú számológépet emulálja (tehát a $2 + 3 \cdot 4$ kiértékeléséhez a `2 3 4*+p` parancsot kell küldeni a bemenetére). A számológép forráskódja megtalálható `dc.sed` néven a Debian `sed` csomagjában és itt [6] is.

Hordozható sed shebang ♦ A Make-nél felvetett problémát oldjuk meg `sed` esetén is: hogyan lehet olyan szkriptet írni, amelynek első shebang sorába nincs bedrótozva az interpreter elérési útja. A megoldás útja is a szokásos: a szkriptet `#!/bin/sh shebang` segítségével shellben futtatjuk, és a szkript eleje arra utasítja a shellt, hogy keresse meg, és indítsa el az interpretert az adott szkripttel. A szkript eleje nem csak shellben fut, hanem adott interpreterben is, és ott nem csinál semmit. A megoldás `sed` esetén (Bourne és C shellekkel is működik) az alábbi fejléc alkalmazása:

```
#!/bin/sh
\false>true;eval '(exit $?0)&&eval `
: f{bin;case c in esac;exec sed -n -f "$0" "${1+"$@"}'; \
exec sed -n -f "$0" $argv:q
:in} # Don't touch lines 1--6: http://www.inf.bme.hu/~pts/Magic.sed.Header
```

A fejléc shell-szkriptkénti értelmezése nem okoz nehézséget. A sedes értelmezés kicsit bonyolultabb: `\f`-től `f`-ig egy reguláris kifejezést láthatunk, utána a kapcsos zárójelek közt a `bin` utasítás elugrik a `:in`-re (kihagyva a `c...`, `e...` és `e...` utasításokat, tehát összességében a fenti fejléc sedben semmit nem csinál – és épp ez a feladata.

Az `-n` kapcsoló azt eredményezi, hogy a (módosított) sort tartalmát a sor feldolgozása után a `sed` ne írja ki. Ezt a hatást elérhetjük `-n` nélkül is (vagyis akkor is, ha a szkript `sed -f prog.sed`-ként van meghívva), ha egy `d` utasításból álló sort helyezünk el a szkript végére.

2.5. AWK: stringműveletek strukturált programban

Az AWK a `sed` sor-feltétel–utasítás modelljét követő, klasszikus unixos szövegfeldolgozó nyelv. Szintaxisa a C nyelvhez hasonló, és ennek megfelelően képes egész és lebegőpontos számokkal számolni, tartalmaz strukturált vezérlési szerkezeteket (`if`, `while`), kezel tömböket, és lehet benne saját függvényt definiálni. Eltérés a C nyelvhez képest, hogy a string beépített típus (a kapcsolódó memóriakezelést az AWK-értelmező végzi), az értékeket egymás mellé írással (vagyis az üres operátorral) lehet konkatenálni, a tömbök asszociatívok (vagyis lehet például stringgel indexelni őket), a reguláris kifejezés nyelvi elem (tehát nem kell duplázni a backslasheket), a sor végén pontosvessző nélkül is be lehet fejezni az utasítást, a függvénydefiníciónak más a szintaxisa, és hogy a program nem csak függvény- és egyéb definíciókból és deklarációkból, hanem függvénydefiníciókból és feltétel–utasítás párokból áll. Újdonság a `sed`hez képest, hogy a sorokat szóközök mentén mezőkre bontja (és ezután ízlés szerint hivatkozhatunk az egész sorra vagy annak mezőire), hogy mind a sorhatárt, mind a mezőhatárt jelző karakter átállítható, továbbá hogy külső fájlokól és folyamatokból érkező adatokat is tud fogadni (és írni is).

Linux rendszereken a `mawk` és `gawk` (GNU AWK) szabad implementációk elterjedtek, az utóbbi kicsit több funkciót tartalmaz, például TCP-kapcsolat nyitását.

Az AWK volt az első széles körben elterjedt unixos szkriptnyelv, melyben hosszabb, több-bezer soros szövegfeldolgozó szkripteket is kényelmes volt írni, és az eredményül kapott forráskód általában mások számára is érthető, átlátható lett. Eközben az egysoros szkriptek írása is kényelmes maradt. Az AWK hatással volt több modern szkriptnyelvre is.

Megfigyelhető, hogy egyes Unix-parancsok funkciói integrálódtak a szkriptnyelvekbe. Például a `sed` képes kiváltani a `grep`-et, az AWK a `cut`-ot, és némi programozással a `wc`-t is. A számolás megjelenése az AWK-ban fölöslegessé tette az `exp`-et. Ezek a funkciók modern

szkriptnyelveinkben, kissé eltérő szintaxissal tovább élnek.

Az AWK fő hátrányai: bináris fájlokat nem tud kezelni (a nullás kódú karakterrel és a fájl végén a soremelés hiányával lehetnek gondok), a beépített függvények köre nem bővíthető (például külső programkönyvtárak beemelésével), valamint nagy szoftverek írásához nem ad külön támogatást.

Az AWK, a sed, a bc, a dc is azon egyszerű unixos szkriptnyelvek egyike, melyek teljes leírása egyetlen kézikönyvoldalon elfér.

Hordozható AWK shebang ♦ A szokásos feltételeket teljesítő fejléc, melyet a futtathatóvá tett szkript elejére kell írni, az alábbi:

```
#!/bin/sh
false && 0>/dev/null; true; \
eval '(exit $?0)'&&eval '\
exec awk -f "$0" -- "${1+"$@"}'; #\
exec awk -f "$0" -- $argv:q #/
# Don't touch lines 1--6: http://www.inf.bme.hu/~pts/Magic.AWK.Header
```

A shell-szkriptként történő értelmezés annyi újdonságot tartogat, hogy a true-ra azért volt szükség, hogy a \$? Bourne shellben 00-t adjon vissza (és ne 10-t). A fejlécet AWK-szkriptként ilyen: változónév && 0>/regexp/, ami mindenképpen hamis, mert egy /regexp/ mintaillesztés eredménye sosem negatív – tehát a fejléc AWK-ban az elvárásoknak megfelelően működik: nem csinál semmit.

3. Mai nyelvek

A tárgyalt mai nyelvek közös jellemzői:

- tanultak elődjeik és kortársaik hibáiból, és igyekeznek jobb megoldást adni;
- hatékony stringkezelést biztosítanak reguláris kifejezésekkel;
- lehetőség van bennük objektumorientáltan programozni;
- aktív fejlesztés alatt állnak, az új verziókban nyelvi újítások is várhatók;
- Unixra egyetlen implementációjuk terjedt, és sok esetben azt tekintik az adott nyelven helyes programnak, amit a referencia-implementáció elfogad (egyres nyelveknek van még Java JVM-es vagy .NET-es implementációja is);
- Windowsra és más rendszerekre is van implementációjuk, ezek általában a Uniximplementáció portjai;
- C nyelven bővíthetőek (és ezáltal illeszthetőek hozzájuk C API-jű programkönyvtárak);
- lehetővé teszik nagy szoftverek fejlesztését, és – a Lua kivételével – különböző segéd-eszközökkel (pl. teszt-keretrendszer, dokumentációgeneráló, debugger) támogatják is;
- a Lua kivételével, a megfelelő kiegészítőkkal támogatják mind a szöveges, mind a grafikus (Gtk, Qt vagy saját), mind a webes alkalmazások írását;
- lehetőséget biztosítanak webalkalmazás írására, és az alkalmazás perzisztens futtatására (például a webserverral CGI helyett FastCGI protokollon kommunikálva);
- a Lua, a Ruby és a PHP kivételével fejlett Unicode-támogatást nyújtanak;

- dinamikus kötést használnak, és a Pike kivételével nem ellenőrzik a változók típusát;
- éves vagy nagyobb gyakorisággal az adott nyelvvel foglalkozó nemzetközi konferenciákon mutatják be a legújabb fejlesztéseket.

A mai szkriptnyelvek között konvergencia figyelhető meg: bár a nyelvek szintaxisa és filozófiája közt sok a különbség, az adat- és vezérlési szerkezetek, a beépített és kiegészítő programkönyvtárak közt nagy az átfedés. A második és a harmadik szkriptnyelv elsajátítása már jóval könnyebb az elsőnél.

3.1. PHP: könnyen elsajátítható webprogramozás

A PHP eredeti célja az volt, hogy a weboldalak egy része a szerveren, dinamikusan generálódhasson, és ezzel a felhasználók élőbbé tehesék a honlapjaikat. A nyelv és az implementáció is sokat fejlődött az első elterjedt verzió, a PHP/FI óta. A PHP5 egy objektumorientált programozást is lehetővé tévő, több webszerverrel és több protokollon keresztül is működni képes, széles körben népszerű, könnyen elsajátítható, sok kiegészítő modullal (pl. adatbázis-illesztők) rendelkező, AJAX-os fejlesztésre is alkalmas szkriptnyelv. Népszerűségét jelzi, hogy több webszolgáltató (köztük ingyenesek is) lehetővé teszi PHP-szkriptek futtatását a szerveren.

A PHP rendkívül könnyen tanulható, azonnal használatba vehető, a bonyolultabb szolgáltatásait elég csak akkor megismerni, amikor szükség van rá. A PHP-ről szóló könyvekből Dunát lehet rekeszteni (magyar fordításban is több tucat jelent meg), csakúgy, mint a webes tutorialokból. Rendkívül hasznosak a PHP honlapján levő, függvényekre lebontott referenciakódokhoz fűzött felhasználói megjegyzések, ahol jó eséllyel megtalálható a megoldás, ha egy adott függvény nem úgy működik, ahogy szeretnénk (vagy éppen nem a megfelelő függvénnyel próbálkozunk).

A PHP-s webfejlesztést számos eszköz támogatja, köztük például az Eclipse szabad IDE és a Zend Studio fizetős IDE, de vannak egyéb eszközök, pl. dokumentáció-generálók is.

A PHP szintaxisa a C nyelvéhez hasonló, azzal a különbséggel, hogy a változónevek előtt dollárjelet kell használni, a változókat nem lehet deklarálni (és a függvényargumentumok típusát sem kell megadni). A kódot `<?php` és `?>` közé kell tenni (`include`-olt fájlok esetén is), az ezeken kívüli szövegrészeket a PHP változatlanul a kimenetre másolja. A kimenet webalkalmazás esetén a böngészőnek küldött oldal, parancssori futtatásnál pedig a szabványos kimenet.

A PHP alapból minden HTTP-kéréskor újra betölti és értelmezi a kérést kiszolgáló kódot és a kód által használt, PHP nyelvű programkönyvtárakat is. Az emiatt bekövetkező lassulás ún. *source cache* segítségével megelőzhető. Szabad *source cache* például az APC, és több kereskedelmi implementáció is van. Olyan szoftverek is letölthetők, melyek a szkript végrehajtását gyorsítják (a bajtkód optimalizálásával vagy eltérő futtatási stratégia alkalmazásával).

A PHP-t számos hátránya akadályozza abban, hogy webes felületen kiszorítsa az egyéb szkriptnyelveket (gondolunk itt főleg a Perlre; Rubyra és a Pythonra). Egyes hátrányok a nyelvi természetűek (pl. nincs *closure* a nyelvben; a lokális változók helyett a globálisakat kell külön deklarálni; függvény által visszaadott tömböt nem lehet közvetlenül indexelni; a string- és tömbkezelő műveletek nem fogalmazhatók meg olyan tömören; mint más nyelvekben; külön odafigyelést igényel, hogy egy argumentumot referencia vagy érték szerint adunk át egy függvénynek; nincs megfelelő Unicode-támogatás stb.), mások pedig a beépített függvények és a kommunikációs interfész hiányosságai (pl. túl sok a beépített függvény; kevés a beépített osztály; a függvények elnevezése és paramétersorrendje nem konzisztens, ezért nehezen megjegyezhető; a hiba esetén visszaadott érték nem konzisztens; nincs jól átgondolt, hivatalos út kivételkezelésre és adatbázis-kezelésre; nem lehet minden kivételt elkapni (pl. ha nem létező függvényt hívunk, akkor a hibaüzenetet mindenképpen megkapja weboldal

látogatója); nagy fájlok feltöltéséhez túl sok memóriát (!) használ; a feltöltés folyamata nem szkriptelhető; a komponensek nem kombinálhatók szabadon, például FTP-ről részlegesen letöltött fájlt nem lehet közvetlenül a kimenetre írni). A fentiek miatt egyes, más szkriptnyelvekhez szokott fejlesztőknek kifejezetten kényelmetlen PHP-ben dolgozniuk, mert folyamatosan azt érzik, hogy az adott feladatot a saját, kedvenc nyelvükön sokkal gyorsabban, javítgatások és idegeskedés nélkül megoldanák.

A hátrányok nagy része abból adódik, hogy a mai PHP egy hosszú, szerves fejlődés eredménye, és a fejlesztők az új szolgáltatásokat legtöbbször csak hozzáadták a meglévőkhöz, és ahol csak lehet, nem változtattak, hogy a régi PHP-szkriptek továbbra is működjenek. Emiatt a mai PHP nem pont a mai igényeket szolgálja ki.

3.2. Perl: svájcbicska minden rendszeren

A Perl egy általános célú, hatékony szkriptnyelv. Eredetileg szövegfeldolgozásra készült, pontosabban szövegfájlokból információ-kinyerésre és jelentések generálására. Azóta kiegészült a rendszerhívások és a Unixokon elérhető programkönyvtárak használatának lehetőségével, és számos modul született, amely ezeket kényelmessé és hatékonyvá teszi. Megalapozottá vált a mondás, hogy Perlben mindent meg lehet csinálni, sőt – a Perl jelmondata szerint – „többféleképpen is meg lehet csinálni”. Ez nem is meglepő, hisz a Perl kitalálója nyelvész, és az emberi beszédhez hasonló programnyelvet kívánt alkotni, és a beszéd jellemzője, hogy ugyanazt a gondolatot sokféleképpen ki lehet fejezni, mindenki az izlése és a képességei szerint formálja beszédét.

A Perl szintaxisa csak távolról hasonlít a C nyelvére. A C-hez képest újdonság, hogy a változóneveket – a változó típusától függően – \$, @ vagy % jellel kell kezdeni, a változók típusát nem lehet deklarálni, a változókat nem kell deklarálni (de ajánlott), a függvénydefiníció szintaxisa más, vannak reguláris kifejezések, és stringen belül is van változó-behelyettesítés. A Perlről egy rövid, magyar nyelvű bevezető leírás olvasható [18]-ban. A Perl sokan kritizálják amiatt, hogy nehezen olvasható a kód. A nehéz olvashatóságnak részben az az oka, hogy túl sok nyelvi elem kínálkozik, melyek nagy része írásjelekből áll, így egész hosszú kód-részletek is keletkezhetnek, melyben betűt csak elvétve találni. Jobban olvasható, a Perlhez hasonló tudású szkriptnyelv a Ruby, és nagyon jól olvasható szkriptnyelv a Python.

Egyes Perl-programozók érdekességképpen közzétesznek szándékosan elbonyolított, értelmetlenre írt, rövid programkódokat [16]. Az alábbi kód [11] például

```
not exp log srand xor s qq qx xor
s x x length uc ord and print chr
ord for qw q join use sub tied qx
xor eval xor print qq q q xor int
eval lc q m cos and print chr ord
for qw y abs ne open tied hex exp
ref y m xor scalar srand print qq
q q xor int eval lc qq y sqrt cos
and print chr ord for qw x printf
each return local x y or print qq
s s and eval q s undef or oct xor
time xor ref print chr int ord lc
foreach qw y hex alarm chdir kill
exec return y s gt sin sort split
```

csak Perl kulcsszavakból áll – és ezek a kulcsszavak együtt (írásjelek nélkül) egy működő Perl-programot alkotnak, amely kiír egy rövid üzenetet. Még szerencse, hogy „többféleképpen is meg lehet csinálni”, és a nagy perles projektek nem az elbonyolított utat választják.

Talán a Perl a legtöbb operációs rendszeren használható szkriptnyelv. Fő implementáci-

óját, a Perl 5-öt több tucat Unix és számos nem Unix rendszerre portolták, a legtöbb Linux-disztribúció alapértelmezésben feltelepíti. A Perl 6 megjelenése lassan egy évtizede húzódik, de az előzetes leírások és félkész implementációk azt mutatják, hogy a Perl 5-től lényegesen különböző, teljesen átdolgozott nyelvre és implementációra számíthatunk, Perl 5-ös szkripteket futtatni tudó, kompatibilitási móddal.

A Perlt sokféle célra használják: egysoros szkriptekkel megoldható feladatokra (a sed és az AWK nyomdokain), szövegfeldolgozásra, bináris fájlok feldolgozására, rendszeradminisztrációs műveletekre, egyszerűbb grafikus felületekhez (pl. grafikus Linux-telepítő), webalkalmazásokban, IRC-kliens szkriptelésére (pl. Irssi), szövegszerkesztő szkriptelésére (pl. Vim).

A Perlben írt, újgenerációs, modell–nézet–vezérlő alapú webalkalmazás-keretrendszerek közül a Catalystet és a Maypole-t kell megemlítenünk. Ezen keretrendszerek olyan modellt, programkönyvtárakat és futtatókörnyezetet biztosítanak, melyek jobb kódot eredményező és hatékonyabb webalkalmazás-fejlesztést tesznek lehetővé, mint a korábban megszokott módszer („elkezdem írni a CGI-szkriptet, és folyamatosan bővítem minden funkcióval, ami csak kell”). Az említett keretrendszerek célkitűzésben és tudásban a Ruby on Railshez hasonlók.

A Perl annak idején reguláris kifejezéseivel *de facto* szabványt teremtett. Később megszületett a PCRE, ami a Perl-kompatibilis reguláris kifejezések Perl-től független megvalósítása egy programkönyvtárban. Használja a PHP, az Apache, a Patsfix és az Nmap is. A reguláris kifejezésekkel történő szövegfeldolgozás hatékonysága azonban nemcsak a reguláris kifejezések szintaxisától, hanem azoknak a programozási nyelvbe való integrálásától is függ. A PCRE csak az előbbit veszi át a Perl-től, az integrálás szoftverenként más és más. (A további megjegyzések nem a PCRE-re vonatkoznak.) AWK, Perl és Ruby esetén például a reguláris kifejezések a nyelv részei, és van beépített illesztés, csere és feldarabolás művelet. PHP, Python, Lua, Pike és Java esetén viszont az integráció nem teljes, ezért minden regex-művelet kényelmetlen, nehézkes, és fölöslegesen bonyolítja a kódot ezekben a nyelvekben pl. a Perlhez képest.

Minden fejlesztő számára nyitott a lehetőség, hogy az általa készített Perl-modulokat az egész közösség számára szabadon elérhetővé tegye. Ehhez biztosít infrastruktúrát a CPAN [5], ahol a legtöbben publikálják a saját fejlesztésű moduljaikat. Az olyan nagy projektek legfrissebb kiadott verziói, mint maga a Perl 5, a Perl 6, a Catalyst és a Maypole is a CPAN-ről érhetők el. A CPAN a Perl-modulok kifogyhatatlan tárháza. Egyik fő szolgáltatása, hogy webes felületén kereshetünk a Perl és a Perl-modulok dokumentációjában, a Perl-modulok metaadataiban (pl. modulnév, szerző, disztribúció). A megtalált modul forráskódját a weben le is lehet tölteni, ám erre inkább a CPAN shell ajánlott, amely a Perl telepítésekor felkerül a rendszerre, és a CPAN-ről tudja magát és más modulokat is frissíteni. Részletes magyar nyelvű leírás a CPAN-ről [13]-on.

Szkriptek perzisztens futtatására használható a Perl-specifikus SpeedyCGI és az általános FastCGI protokoll is.

A Perlt – a rosszul megírt forráskód nehezen olvashatósága mellett – kritika szokta érni azért, mert nem kínál egy egységes utat az objektumorientált programozásra, továbbá, hogy a memóriafelszabadítás nem szemétyűjtésen, hanem referenciaszámláláson alapul. Ezeket a hátrányokat a Perl 6 javítani fogja.

Hordozható Perl shebang ♦

```
#!/bin/sh
eval '(exit $?0)' && eval 'PERL_BADLANG=x;PATH="$PATH.";export PERL_BADLANG\
  PATH;exec perl -x -S -- "$0" "${1+"$@"}";# 'if 0;eval 'setenv PERL_BADLANG x\
;setenv PATH "$PATH";exec perl -x -S -- "$0" $argv;q;#'.q
#!/perl -w
+push@INC, '.';$0=~/(.*)/s;do(index($1,"/")<0?"/$1":$1);die@if$@__END__+if 0
```

```
:#Don't touch/remove lines 1--7: http://www.inf.bme.hu/~pts/Magic.Perl.Header
```

Ez a megoldás kicsit hosszabb a megszokottnál. Részletes magyarázata angol nyelven [12]-ben olvasható. A `PERL_BADLANG=x` értékadásra azért volt szükség, hogy a Perl ne mutasson figyelmeztető üzenetet, ha a locale beállítás hibás a rendszeren. A Perl, ha az első, shebang sorban nem látja a `perl` stringet (mint esetünkben), akkor a programot a shebang sorban látott parancsnak adja át. Ez esetünkben nem jó, mert végtelen ciklust kapnánk (a shell meghívja a Perlt, ami visszahívja a shellt stb.). A végtelen ciklust úgy kerüli el a megoldást, hogy a Perl a `-x` kapcsolóval hívja, melynek hatására a Perl a `#!perl`-es sortól kezdve kezdi értelmezni a szkriptet. Ekkor viszont a hibüzenetben elromlanak az oldalszámok, ennek orvoslására a következő sorban a szkript `do`-val betölti és futtatja önmagát (majd `__END__`-del kilép, hogy ne fusson kétszer). Amikor viszont előlről kezdi futtatni a szkriptet, akkor a `do`-t tartalmazó sort végre se hajtja, mert azt el van rejtve egy `q+...+ if 0`; utasításban.

A fentieket figyelembe véve a fenti fejléc négy különböző módon képes futni (és mindegyik módnak van haszna): Bourne shell-szkriptként, C shell-szkriptként, Perl-szkriptként előlről és Perl-szkriptként a `#!perl` sortól.

Hordozható SpeedyCGI shebang ♦

```
#!/bin/sh
eval '(exit $?0)'||eval 'setenv PERL_BADLANG x;set S="-w";>/dev/null\
which speedy&&exec speedy $S "$0" $argv;q;exec perl $S -eshift=-/\(.\*\)/"\
;do(("\$0=\$1")=-m,^.?.?/,?"\$0:qq,./\$0,\)\);die\$@if\$@ "$0" $argv:q#'if 0;
eval 'export PERL_BADLANG=x;type -p speedy>/dev/null&&exec speedy -w \
"$0" ${1+"$@"};exec perl -w -e"shift=-/(.*)/;do((\$0=\$1)=-m,^.?.?/,?"\$0
:qq(./\$0));die\$@if\$@ "$0" ${1+"$@"};:'if 0;BEGIN{delete$INC{+__FILE__};q
#! perl -w
+${0}~/(.*)/;do((\$0=\$1)=-m,^.?.?/,?"\$0:qq(./\$0));die\$@if\$@;__END__+}
## Don't touch lines 1--10: http://www.inf.bme.hu/~pts/Magic.Speedy.Header
```

Lényegesen új ötletet nem tartalmaz a Perl shebanghez képest. A fő különbség az, hogy a szkriptet először a `speedy` programmal próbálja futtatni, és ha ez nem sikerül (például azért, mert a `SpeedyCGI` nincs telepítve), akkor a `perl`-t használja. Sajnos a Bourne és C shellek annyira különböznek, hogy ugyanazt a funkciót külön-külön kellett bennük megvalósítani.

3.3. Python: jól olvasható, objektumorientált szkriptek

A Python egy általános célú, objektumorientált szkriptnyelv, amely főleg nyelvi tisztaságával, egyszerűségével tűnik ki a modern szkriptnyelvek közül. Emiatt első nyelvként oktatásra is kiválóan alkalmas. A fő eltérés a többi szkriptnyelv szintaxisához képest, hogy a program struktúráját nem kapcsolos zárójelekkel, hanem beljebb kezdéssel jelzi. Például 1-től 10-ig a számokat az alábbi programmal lehet kiírni:

```
i=1
while i<=10:
    print i
    i=i+1
print "Vége"
```

További fontos különbség, hogy értékadás nem szerepelhet utasítás közepén. Ez is hozzájárul a jó olvashatósághoz, mert emiatt nem lehet a fenti ciklust egyetlen sorba összevonni, mint például Perl esetén: `$I=0; while ($++I<=10) { print "$I\n" }`.

A legismertebb Python-alkalmazások a Zope webes tartalomkezelő, a Zorp magyar fejlesztésű, az alkalmazási rétegben működő tűzfal, a Gentoo Linux Portage nevű csomagkeze-

lője és a Trac webes projektmenedzsment eszköz (SVN-vizualizációval, hibajegykezelővel és wikivel). Egyes szoftverek, például a XEN, konfigurációs fájljait Pythonban kell elkészíteni.

Hordozható Python shebang ♦

```
#!/bin/sh
"""true"; eval '(exit $?0)'&&eval '\
exec python -- "$@" ${1+"$@"}'
exec python -- "$@" $argv:q #"""
# Don't touch lines 1--5: http://www.inf.bme.hu/~pts/Magic.Python.Header
```

Az alapötlet az, hogy Pythonban a többsoros stringkonstansokat három idézőjel határolja, míg a shell-szkriptben a két nyitó idézőjelnek semmi hatása, a harmadik zárópárja pedig rögtön a `true` szó után található. A Python figyelmen kívül hagyja az önmagában álló stringkonstansokat, tehát a fejléc a Pythonban nem csinál semmit.

Megjegyezzük, hogy a C shell egy parancsot külső programként futtat, ha a parancs neve idézőjelet vagy backslashot tartalmaz. Tehát a fenti `"""true"` C shellben a `/bin/ttrue`-t fogja futtatni, Bourne shellben pedig a shell belső `true` parancsát. Hatásuk ugyanaz.

3.4. Ruby: a programozók kedvence

A Ruby is egy új, objektumorientált, általános célú szkriptnyelv, amely igyekszik ötvözni más szkriptnyelvek jó tulajdonságait. Szintaxisára hatással volt a Python tisztasága (de nem érzékeny a beljebb kezdésre: a blokk végének jelzésére az `end` kulcsszó használatos), és ugyaninnen vette a stringek és a listák intervallummal történő indexelését is. A Smalltalktól vette át azt az elvet, hogy minden érték objektum, még a számok is (ettől még nem pazarolja a memóriát számok tárolásakor), a Perl-től és az AWK-től örökölte a reguláris kifejezések nyelvbe jól integrált kezelését.

A jó olvashatóság érdekében a konstansokat (beleértve az osztályok neveit is) nagybetűvel kell kezdeni, a lokális változókat kisbetűvel (és külön jele van a globális változóknak és a példányváltozóknak). Ez kódoláskor nem jelent nagy terhet, viszont mások kódját sokkal gyorsabban olvashatóvá és érthetővé teszi.

Nyelvi újdonság Rubyban az *iterátor* fogalma. A Ruby-beli iterátor egy kényelmes szintaxisú *closure*, vagyis olyan utasításblokk, amely a környezetének lokális változóit használja, de nem a környezetéből kerül meghívásra. Például az alábbi Ruby-szkript az argumentumait adja össze egész számként:

```
sum = 0
ARGV.each { |arg| sum += arg.to_i }
p sum
```

A program kapcsos zárójelbe tett része az iterátor: ez egy olyan kódrészlet, ami nem közvetlenül kerül meghívásra, hanem az `each` metódus hívja az `ARGV` tömb minden elemére egyszer. Az iterátor nem forradalmian új ötlet; a Ruby újítása abban áll, hogy kényelmes szintaxist biztosít rá, és előszeretettel használja standard programkönyvtárában.

Egyedi megoldás a Rubyban, hogy feltételben csak a `false` és a `nil` érték számít hamisnak, és például a `0` és a `0.0` és az üres string is igaz. A Ruby alkotója nem véletlenül döntött így: ezáltal tömör és kifejező kód írható az alábbihoz hasonló feladatokra: „ha a kép szélessége nem ismert, akkor próbáld meg GIF fájlként betölteni, és kinyerni a szélességet, és ha ez nem sikerül, akkor próbáld meg JPEG fájlként betölteni, és kinyerni a szélességet”. A kód a következő:

```
width ||= get_gif_width(image) || get_jpeg_width(image)
```

A megoldás akkor működik, ha a hívott függvények nil-t adnak vissza, ha nem ismerik a formátumot, és egy szélességet jelölő számot (akár 0-t is!), ha ismerik. Hasonlóan tömör megoldást nem lehet adni olyan nyelvekben, melyek a nullát hamisnak tekintik.

Rubyban bizonyos függvényeknek több nevük van. Például nem kell emlékeznünk, hogy egy string hosszát `length` vagy a `size` metódus adja-e vissza: bármelyiket használhatjuk.

A Ruby ötletesen nevezi el a *getter/setter* metódusokat. Például az `img.width=42` értékadás az `img` objektum `width=` metódusát, a `img.width` lekérdezés pedig a `width` metódusát hívja. Ezáltal a Ruby megelőzi, hogy a kód tele legyen `get`-tel és `set`-tel.

Az osztályokhoz bármikor felvehető új metódus. Ez az alapsztályokra is igaz, így például az egész számokra és a stringekre alkalmazott metódusok körét is bármikor bővíthetjük.

A fentihez hasonló apró ötletek együttese eredményezi, hogy Rubyban keveset kell gépelni, nem kell fölösleges dolgokon gondolkodni, és jól olvasható a kód.

A Ruby nyelv a Ruby on Rails nevű, modell–nézet–vezérlő alapú webalkalmazás-keretrendszer megjelenésével és elterjedésével került be a köztudatba. A Railszel egyszerű és kényelmes az adatbázis alapú webes alkalmazások fejlesztése. A Rails egy olyan modellt definiál, melyben nagy webalkalmazások párhuzamos fejlesztése is lehetővé válik a kód túlzott elbonyolódása nélkül. A [2] cikkben rövid magyar leírás olvasható a Railsről. A Rails hamar a programozók kedvencévé vált, és hatással volt más programnyelvek webalkalmazás-keretrendszerének fejlődésére is (például hasonló perles keretrendszer a Catalyst). A Rails aktív fejlesztés alatt áll.

A Ruby saját szálkezeléssel rendelkezik, amit sajnos nem elég stabil, gyakran beragadnak a többszálú szkriptek.

A Ruby nem csak webfejlesztéskor remekel, hanem szövegfeldolgozásra, hálózati programozásra (pl. másolatás több TCP/IP kapcsolat mögött), XML-feldolgozásra is kiváló, sőt: Tk- és GTK-kötése segítségével grafikus alkalmazások is fejleszthetők. Ideális továbbá gyors prototípuskészítésre. Tiszta objektumorientált szemlélete miatt programozási nyelvek tanulásakor első objektumorientált nyelvnek is jó.

Hordozható Ruby shebang ♦

```
#!/bin/sh
eval '(exit $?0)##'+/&&eval '#/ if 1<1&&'
PATH="$PATH:"; exec ruby -x -S -- "$0" "${1+"$@"}"
setenv PATH "$PATH":.;exec ruby -x -S -- "$0" $argv:q
#!/ruby -w
##Don't touch/remove lines 1--6: http://www.inf.bme.hu/~pts/Magic.Ruby.Header
```

A shell és a Ruby szétválasztása az `eval`-lal kezdődő sorban történik. Ezt a sort a shell így látja: `eval '(exit $?0)##' && eval '#\`, a Ruby pedig így: `eval 'string' + /regexp/ if 1<1&&'parancs'`. A parancsot záró `'` az utolsó sorban fejeződik be. A Ruby a vérehajtást az `1<1` feltételrész kiértékelésével kezdi, ami hamis, tehát nem értékeli ki se a `'parancs'`-ot (ami külső program futtatását eredményezné), se a `'string'`-et, se a `/regexp/`-et. Elértük tehát a kívánt hatást: a fejléc Rubyban semmit nem csinál.

3.5. Pike: webprogramozás C-szerű szintaxissal

Az LPC a MUD-ok² egy nagy részének (az LPMud-oknak) objektumorientált szkriptnyelvé volt, vagyis LPC nyelven alakították ki a fejlesztők a MUD világot: a helyszíneket, az ellenségeket (és a barátságos lényeket, például kereskedőket), a felszerelési tárgyakat és a harcrendszert is. A játék logikája is LPC nyelven volt leprogramozva: a rendszer milyen pa-

² az 1990-es évek elejének telneten játszható szöveges szerepjátékai, ahol a fő cél az egyre erősebb ellenségeket legyőzése egyre jobb képességekkel és fegyverekkel, esetleg a többi játékos segítségével

rancsokra hogy reagál, egy páncél milyen hatással van viselője harcértékére, mikor és milyen mértékben fejlődnek a játékos képességei stb. Egy világot általában sokan fejlesztettek: nem volt ritka, hogy a játékot indító 3–5 fős csapat mellé később akár tucatnyian csatlakoztak, akik általában kevesebb jogosultsággal rendelkeztek: csak egy-egy küldetés vagy világrész kidolgozása volt a feladatuk, a játék logikáját nem módosíthatták. LPC-ben az egyes objektumokat (pl. játékosok) ki lehetett menteni, hogy a játék újraindításakor tulajdonságaik megőrződjenek.

Manapság a MUD-okat már kiszorították a 3D-s szerepjátékok, és ezáltal az LPC nyelv is veszített jelentőségéből. A változás nem érintette azonban a Pike-ot, amely az LPC által ihletett, de azóta továbbfejlesztett, önállóan (nem MUD-on belül) használatos, általános célú, objektumorientált szkriptnyelv. Első és legfontosabb ipari alkalmazása Roxen Challenger webszerver volt, amely C-ben és Pike-ban vegyesen volt írva, és mind a szkriptelést, mind a webalkalmazások fejlesztését támogatta Pike-ban. A programról Caudium néven szabad szoftver projekt ágazott le, és a mai napig ez az egyik legnagyobb Pike-ra épülő projekt.

A Pike szintaxisa – az LPC-n keresztül – a C nyelvből származik, és megőrizte azt, hogy a változókat típusal deklarálni kell, továbbá gyűjtemények (pl. tömb, asszociatív tömb) az elemek típusát is meg kell adni. Lehetséges viszont típusmegkötés nélküli változót (mixed) is deklarálni. Pike-ban, hasonlóan a szkriptnyelvek többségéhez, van `string` típus, automatikus a memóriakezelés, és az összetett típusok referencia szerint kerülnek átadásra.

Hordozható Pike shebang ♦

```
#!/bin/sh
#if 0
eval '(exit $?0)&&eval '\
exec pike -- "$@" ${1+"$@"}'
exec pike -- "$@" $argv:q
#endif // Don't touch lines 1--6: http://www.inf.bme.hu/~pts/Magic.Pike.Header
```

A fejléc azt használja ki, hogy a Pike rendelkezik preprocesszorral, így az `#if 0` és az `#endif` közötti sorokat figyelmen kívül hagyja. Ugyanezzel a trükk például C nyelven is bevethető, tehát lehet olyan C forrásfájl írni, ami shell-szkriptként futtatva lefordítja önmagát a megfelelő beállításokkal.

3.6. TCL: grafikus felületeket gyorsan

A TCL nyelv szorosan kötődik a vele együtt fejlesztett Tk grafikus eszközkészlethez. A Tk volt az 1990-es évek közepén az egyik első könnyen szkriptelhető eszközkészlet (angolul *toolkit*) X11 grafikus felületre. Később kiadták a TCL/Tk-t Windowsra és Mac OS X-re is, ezzel lehetővé vált a támogatott rendszerek között hordozható (ám nem teljesen azonos kinézetű) grafikus ablakozó alkalmazások írása. A TCL/Tk az első szkriptnyelvek egyike volt, melyek a Unicode-támogatást nyújtottak nem csak stringekben, hanem reguláris kifejezésekben és a grafikus vezérlőelemeken is. Az eseményvezérelten programozható Tk grafikus eszközkészletet kiegészítették szintén eseményvezérelt socketkezeléssel, ezáltal lehetővé vált TCP/IP kliensek és szerverek írása TCL-ben. A fenti előnyök miatt a TCL/Tk népszerűvé vált gyors prototípus készítésre, főleg grafikus felületek esetén.

A TCL nyelv egyedi szintaxissal rendelkezik, ami (a LISP-hez hasonlóan) zárójelezésen alapul, és az operátorok és a vezérlési szerkezetek teljesen hiányoznak belőle, pontosabban: ezen szolgáltatásoknak nincsen szintaktikus megfelelője. A TCL-program utasításokból áll, melyeket soremelés vagy pontosvessző választ el egymástól. Minden utasítás egy szóközzel elválasztott lista, melynek első eleme a parancs neve, a további elemek pedig az argumentumok. Ha kapcsos zárójelbe teszünk egy listaelemet, akkor egyben marad, nem bontódik fel

szóközök mentén. Ha szögletes zárójelbe tesszük, akkor TCL-utasításként lefut, és az általa visszaadott string kerül behelyettesítésre. A szintaxisban szerepel még dollárjel után változó-behelyettesítés és stringképzés idézőjellel. A TCL fő adatstruktúrája a string – ha egy a lista stringként van megadva, akkor ugyanúgy bontandó elemeire, mint egy utasítás. Például az alábbi utasítás

```
puts [lindex {a {b {c d} e} f} 1]
```

hatására először a kétargumentumú `lindex` parancs fut le, ami az első argumentumában kapott a `{b {c d} e}` f listának választja ki az első elemét, és mivel a TCL a listaelemeket nullától számozza, az első elem a `b {c d} e` string lesz (ami értelmezhető listaként is, ha listakezelő parancsnak adjuk), de jelen esetben a `puts` parancs kapja meg, ami soremeléssel lezárva kiírja a képernyőre. (A fenti utasítás beírható `tclsh` program indítása után.) Vegyük észre, hogy mind az utasításvégrehajtás, mind az `lindex` eltüntetett egy kapcsoszárójel-párt.

Példa ciklusszervezésre:

```
set i 1; while {$i<=10} {puts $i; incr i}
```

A fenti programrészlet 1-től 10-ig írja ki a számokat. Láthatjuk, hogy a TCL a stringeket több célra is felhasználja: számok és listák is megadhatók és felhasználhatók stringként, továbbá aritmetikai kifejezést (pl. `$i<=10`) és a kódrészletet is (pl. `puts $i; incr i`) is stringként kell átadni.

A TCL szintaxisa volt az egyik fő akadályja annak, hogy a nyelv általános célú szkriptnyelvként elterjedjen. Gyakori, hogy a TCL-programozó nem tud a feladat megoldására koncentrálni, mert a kapcsos és szögletes zárójelek pontos elhelyezésére kell figyelnie, melyek gyakran 5 vagy még nagyobb mélységben, nehezen követhetően ágyazódnak egymásba – más szkriptnyelvekben ugyanazt a feladatot zárójel nélkül, sokkal gyorsabban meg lehetett volna oldani, és az eredmény is tömörebb és érthetőbb lenne.

A TCL alapértelmezésben nem objektumorientált, de [*incr Tcl*] néven van hozzá ilyen kiegészítő.

A Tk grafikus eszközkészlet szorosan kötődik a TCL-hez. A Perl/Tk és Ruby/Tk párosítások is használnak TCL-t a belsejükben, de ezt, amennyire csak lehet, elrejtik, és az eseménykezelők megírását Perl illetve Ruby nyelven szorgalmazzák. A GTK és Qt eszközkészletek megjelenésével a Tk visszaszorult: a két új eszközkészlet nem csak szebb és intuitívabb a mai felhasználó számára, hanem jóval több beépített vezérlőelemet tartalmaz, és egyedi vezérlőelemek írása sem túl bonyolult. A modern szkriptnyelvekhez van GTK- vagy Qt-illesztés.

A nem grafikus felhasználására példa az egyik legnépszerűbb IRC-bot, az Eggdrop, amely TCL-ben szkriptelhető.

Hordozható TCL shebang ♦

```
#!/bin/sh
# Don't touch lines 1--5: http://www.inf.bme.hu/~pts/Magic.TCL.Header \
eval '(exit $?0)'&&eval '\
exec tclsh "$0" "${1+"$@"}";#\
exec tclsh "$0" $argv:q
```

Az alkalmazott alaptrükk ismerős: a szétválasztás azt használja ki, hogy TCL-ben a megjegyzés a sor végére tett backslashsel a következő sorban folytatódik, míg shellben nem. Ha TCL/Tk grafikus alkalmazásainkhoz szeretnénk hasonló fejléct, akkor a fentiben a `tclsh`-t cseréljük `wish -f-re`.

3.7. Lua: letisztult, beágyazott szkriptnyelv

A Lua kilóg a tárgyalt modern szkriptnyelvek sorából, mivel beágyazott környezetre tervezték. Ez egyszerűbb nyelvet, rövidebb kódot, és ezzel együtt kisebb funkcionalitást von maga után. (Összehasonlításképpen: a Lua 5.0 programkönyvtárának mérete Linux alatt 118 kB, a Perl 5.8.4-é 1150 kB, a Ruby 1.8.2-é pedig 748 kB.) A Luáról bevezető jellegű, magyar nyelvű leírás [3].

Fontosabb eltérések a többi tárgyalt modern szkriptnyelvtől:

- Nincs teljes, objektumorientált kivételkezelés (de azért lehet dobni és elkapni string típusú kivételeket). Hiba esetén a hívási verem elérhető.
- Az objektumorientált programozás támogatása szokatlan. Módszere [8] JavaScript prototípusos megoldáshoz áll közel. Az öröklődést ún. *metatáblák* segítségével valósítja meg. Az objektum metódusait tartalmazó táblához csatolt metatábla lehetővé teszi, hogy hiányzó metódus hívása esetén a Lua a metódust a szülő osztály táblájában keresse. (Metatáblákkal egyébként tetszőleges objektum tetszőleges operátora felüldefiniálható.)
- A reguláris kifejezések egyszerűbbek, kevésbé kifejezőek a Perlben és a POSIX ERE-ben megszokottnál.
- A szintaxisa kicsit fura – például nem fogad el kifejezést utasítás helyén.
- Szokatlansága ellenére kerek, letisztult, átgondolt egészet alkot. Minden, a szkriptprogramozáshoz szükséges nyelvi elemnek megvan benne a helye.

A Lua első nagy felhasználói a játékprogramok voltak (RPG-k és lövöldözős játékok egyaránt). Azóta több szövegszerkesztő, IDE, webszerver (pl. `lighttpd`) és peer-to-peer kliens (pl. `BCDC`) is szkriptelhető Luában. Érdekesnek ígérkezik a `LuaTeX`, amely a `TeX` és a Lua tudását egyesíti. Részletes lista a Luában szkriptelhető programokról [10]-ben.

Megjegyezzük, hogy a webszerver szkriptelése nem ugyanaz, mint a webalkalmazások írása egy az szkriptnyelven a webszerver felhasználásával. A webalkalmazások általában a tartalmat generálják, ezzel szemben a webszerverhez írt szkriptek pl. a kérés kiszolgálása előtt végeznek jogosultságellenőrzést, az URL-átírást valósítják meg, és arról döntenek, hogy ez oldal változott-e a böngészőben levő cache-elt változathoz képest. Az Apache `mod_perl`, `mod_python` és `mod_ruby` moduljai és a Caudiumban futó Pike mind szkriptelésre, mind webalkalmazások írására alkalmasak, míg az Apache PHP-modulja inkább csak webalkalmazásokra, a `lighttpd` Lua-modulja pedig csak szkriptelésre jó.

A Lua megjelenése előtt a komolyabb szövegszerkesztők vagy saját szkriptnyelvet tartalmaztak (pl. `Vim`, `Emacs`, `Epsilon`), vagy – kiegészítőkként – Perlben vagy Pythonban voltak szkriptelhetők (pl. `Vim`). Mi lehet az oka, hogy újabban egyre több szoftver választja a Luát az általános célú, „nagy” szkriptnyelvek helyett? A kis mérete mellett a telepítés egyszerűsége (főleg Windows alatt) is fontos ok. Emellett a Lua C-interfésze és forrása jobban átlátható és érthető, mint egy nála sokkal nagyobb szkriptnyelvé – ezzel csökken a nehezen reprodukálható, verziófüggő és rendszerfüggő hibák valószínűsége. Az alkalmazásfejlesztők számára tehát kisebb teher és kisebb kockázat a Luát beemlenni, mint a „nagy” szkriptnyelveket.

Hivatkozások

- [1] ACM Software System Award díjak.
URL <http://awards.acm.org/software%5Fsystem/>.
 - [2] Bártházi András: Ruby on Rails, 2005. augusztus.
URL <http://weblabor.hu/cikkek/rubyonrails>.
 - [3] Bevezető leírás a Lua-ról, elsősorban BCDC-seknek.
URL http://ro.4242.hu/cgi-bin/yabb2/YaBB.pl?board=bcdc_help.
 - [4] BusyBox: a beágyazott Linux svájcbicskája. URL <http://www.busybox.net/>.
 - [5] CPAN: a Perl-modulok kifogyhatatlan tárháza. URL <http://www.cpan.org/>.
 - [6] dc implementáció sed-ben.
URL <http://sed.sourceforge.net/grabbag/scripts/dc.sed>.
 - [7] GNU szoftverfordítási rendszer.
URL http://en.wikipedia.org/wiki/GNU_build_system.
 - [8] Roberto Ierusalimsky: Objektum-orientált programozás Lua-ban, 2003.
URL <http://www.lua.org/pil/16.html>.
 - [9] Brian W. Kernighan – Rob Pike: *A UNIX operációs rendszer*. 1987, Műszaki Könyvkiadó, Budapest.
 - [10] Luában szkriptelhető programok listája. URL <http://www.lua.org/uses.html>.
 - [11] Mike Rosulek: Csak kulcsszavakból álló Perl-szkript, 2003.
URL <http://perlmonks.org/?node=0bfuscated%20Code>.
 - [12] Szabó Péter: A magic header for starting Perl scripts. 2003. április., *The Perl Journal*, 13–15. p. URL <http://i.cmpnet.com/ddj/tpj/images/tpj0304/0304tpj.pdf>.
 - [13] Szabó Péter: Perl-modulok telepítése a CPAN-ról, 2005.
URL <http://www.szsi.hu/wikidev/PerlModulokCPANr%C5%91l>.
 - [14] Szkriptnyelvek. URL http://en.wikipedia.org/wiki/Scripting_language.
 - [15] Szoftverfordítás automatizálása.
URL http://en.wikipedia.org/wiki/Build_Automation.
 - [16] Szándékosan nehezen érthetőre írt rövid Perl-szkriptek.
URL <http://perlmonks.org/?node=0bfuscated%20Code>.
 - [17] UNIX-filozófia. URL http://en.wikipedia.org/wiki/Unix_philosophy.
 - [18] Verhás Péter: Perl röviden.
URL <http://www.szabilinux.hu/verhas/perl/index.html>.
 - [19] Wikipedia: szabad enciklopédia. URL <http://en.wikipedia.org/>.
-