GNU rendszerben fejlesztés

Dévényi Károly Pásztor György

Utolsó módosítás: 2003. április 30.

Tartalomjegyzék

1. fejezet

I. félév

1.1. A programozáshoz használt segédeszközök

1.1.1. A programozáshoz használt szövegszerkesztő (VIM) bemutatása, „testre-szabása”, stb.

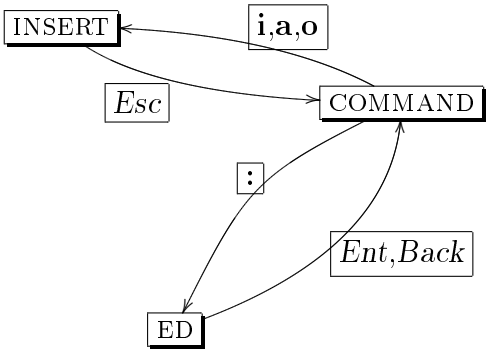
Az alap vi bemutatása

A vi használatában három üzemmód van:

insert mode Beszúró üzemmód. Szabadon gépelhetünk szöveget, ez bekerül a dokumentumba

command mode Különböző parancsokat adhatunk ki (pl. kurzor alatti karakter törlése **x**, sor törlése **dd**, beszúrás **i**, utána-írás/*append* **a**, új-sor kezdés **o**)

ed mode Különböző az ed szövegszerkesztőben megszokott parancsokat adhatunk ki (pl. fájlba írás **:w**, fájl beolvasása **:r***fájlnév*, helyettesítés **:s**)



1.1. ábra. Váltás a vi módjai közt

A command mód fontosabb parancsai

x Annak a karakternek a kitörlése, amin a kurzor áll

d Sor, vagy sorok törlése

y Sor, vagy sorok vágólapra helyezése

p Vágólap beillesztése a kurzor után

i Beszúró módba váltás, a kurzor helyére szúr be

a Beszúró módba váltás, a kurzor utáni helyre kezd beszúrni

o Beszúró módba váltás, új sort kezd, a kurzor utáni sorban

w Következő szóra ugrás

c Szövegrész cseréje

/ Reguláris kifejezés keresése.

? Reguláris kifejezés keresése a szövegben visszafele haladva.

n A következő találat megkeresése, az előzőleg már megadott regexp-hez.

h l j k Balra, Jobbra, Le, Fel mozgás a szövegben. (Általában a kurzorbillentyűk ugyanúgy használhatók)

~ Megcseréli az aktuális karaktert, ha az kisbetűs volt annak nagybetűs változatára, ill. ha nagybetűs volt, akkor a kisbetűs változatára.

. Megismétli a legutolsó vi parancsot

Megj.: a p,i,a,o parancsoknak van nagybetűs változata is. A vi „gondolkodásmódja", amikor parancsot adunk ki:
[Multiplikációs szám]PARANCS**[[Multiplikációs szám] Argumentum]**
Pl. törlés *(d)*, vagy vágólapra másolás *(y)* esetén tudatni kell a vi-jal, hogy „mit" szeretnénk kivágni. Pl. ha szeretnénk a kurzortól kezdődő 14 betűt a vágólapra tenni, akkor a következő billentyűket kell leütnünk: **d14**SPACE
További példák:

2×5 sor törlése	**2d5d**
5 sor vágólapra tétele	**5yy** vagy **y5y**
14 karakter törlése	**d14␣** vagy **14d␣** vagy **14x**
5 szó vágólapra tétele	**y5w** vagy **5yw**
2 szó kicserélése a 'broaf' szövegre	**c2wbroaf** ESC

Megjegyzések:

- ESC*-re azért van szükség, mert a c parancs is beszúró módba tesz.*
- A nagybetűs változatok:

	kisbetűs eset	nagybetűs eset
i/I	beszúr	sor elejére szúr be
o/O	következő sorba kezd új sort	előző sorba kezdi az új sort
p/P	utána beilleszt	aktuális pozíciótól kezdve beilleszt
a/A	a kurzor után beilleszt	a sor végére illeszt be

Az ed mód fontosabb parancsai

r fájl beolvasása. A kurzor utáni sorban kezdve beszúrja a megadott fájl tartalmát.

w fájl elmentése. Ha fájlnevet is megadunk, akkor a megadott fájlnévbe ment. Ha előtte megadjuk, hogy csak bizonyos sorokra vonatkozzon a parancs, akkor csak azokat a sorokat menti el. Pl. a kurzor sorát, és az utána következő még 4 sort (tehát összesen 5 sort) szeretnénk egy másik fájlba elmenteni, akkor a kiadandó parancs: **:,+4w masikfile.txt**

q Kilépés. Ha megváltozott a fájl, és nem mentettük azóta, akkor nem lép ki, hanem figyelmeztet.

wq Elmenti a fájlt, és utána lép ki.

q! Mindenféleképp kilép.

w! Ha a fájl írásvédett, akkor is megpróbálja felülírni.

s csere *(szubsztitúció)*. Reguláris kifejezést adhatunk meg cserére, és a perl-ben megszokott módon lehet kifejezéseket megadni. Részleteket lásd később. Néhány példa:

Az aktuális sorban a joe szót cseréljük ki vim-re.	**:s/joe/vim/**
Az aktuális sorban az összes joe szót cseréljük ki vim-re.	**:s/joe/vim/g**
Az aktuális és köv 2 sorban a sor első joe-ját cseréljük ki vim-re.	**:,+2s/joe/vim/**
Az első sortól az utolsó előttiig a sor első joe-ját cseréljük ki vim-re.	**:1,$-1s/joe/vim/**

A VIM-ben megjelent új parancsok

V *Visual Line*: Sorok kijelölésére szolgál, és a megadott parancsot az ily-módon kijelölt sorokra hajtja végre. Pl. Ha nem tudjuk fejből, hogy hány sort szeretnénk a vágólapra tenni, akkor csak kijelöljük őket, és utána y-t ütünk.

^**V** *Visual Block*: Hasonló az előzőhöz, de itt nem sorokat, hanem téglalap alakú területeket tudunk a szövegben kijelölni.

< Kijjebhúzza a kijelölt programblokkot.

> Beljebhúzza a kijelölt programblokkot.

A VIM-ben megjelent új ed-szerű parancsok

:sp *Split screen*: Vízszintesen kétfelé osztja a képernyőt, és ha fájlnevet is megadunk, akkor a megadott fájlnevet betölti a képernyő másik felébe.

:vsplit *Vertical Split screen*: Ugyanaz mint az előző, csak függőlegesen vágja ketté a képernyőt.

A szétosztott képernyők esetén a mozgás a képernyőrészek közt a CTRL-**w** -vel történik. Figyelem: Ez csak command módban használható!
Használata: Ha még egyszer leütjük, akkor ciklikusan a következőre ugrik. Ha megadott irányba akarunk elmozdulni, akkor a már megszokott módon leütjük a **h**, **j**, **k**, **l** billentyűk valamelyikét,

vagy a kurzormozgató nyilak valamelyikét. Ha az aktuális képszeletet szeretnénk növelni, akkor a +, ill ha csökkenteni szeretnénk akkor a **-** billentyűket kell leütni. Itt is ugyanúgy üthetünk előtte vagy a CTRL-**w** után számokat, hogy adott számszor hajtsa végre a műveletet (, vagyis ne csak 1 sorral növekedjen/csökkenjen az ablak mérete, hanem pl. 5-el: CTRL-**w 5**+).

Makrózás a vim-ben

A vim a makrókat az angol abc 26 betűjével azonosítja. Ha egy makrót rögzíteni akarunk, akkor üssük le a **q** betűt, majd a makró *nevét*. Az ezután következő vi parancsokat a vim megjegyzi. Ha a makrónkat be akarjuk fejezni, akkor adjuk ki ismét a **q** parancsot. Ha a státuszsor be van kapcsolva, akkor a makró rögzítése közben a státuszsorban a *recording* felirat látható. A makrót tartalmazó vi parancsokat elő akarjuk hívni, akkor a **@** jelet és a makró nevét kell leütni. Ha a makrót egymás után többször akarjuk végrehajtani, akkor a **@** előtt üssük le a számot is, ahányszor végre akarjuk hajtani.

Néhány hasznos alapbeállítás, .vimrc

syntax on Syntax highlighting bekapcsolása. Automatikusan megpróbálja felismerni a szerkesztett fájl típusát, és az ehhez tartozó szintaxismodult betölti, és ennek megfelelően „színezi a szöveg részeit.

filetype plugin on Ha felismerte a fájltípust, akkor ennek megfelelően betölt néhány ehhez tartozó alapbeállítást. Pl. C fájlok esetén automatikusan indentel gépelés közben. Gyakorlatban ki kell próbálni, pl. egy **switch** elágazásnál.

set showcmd A részlegesen begépelt parancsot jelzi a státuszsorban

set showmatch Ha épp zárójelet lezárunk, akkor egy pillanatra megmutatja a párját

set ignorecase Keresésnél nem számít a kis és nagybetű különbsége

set incsearch Inkrementális keresés

set ruler A státuszsorban, mindig mutatja a kurzor pozícióját, ill. hogy a szövegben hol állunk

set history=50 A vi-ban kiadott parancsok pufferét 50 parancsra visszamenőleg megőrzi

set ts=8 sts=4 sw=2 Indentelésnél 8 space-t cserél ki egy tab-ra *(tab space)*, egy tab leütésre 4 space-t tesz ki *(soft tab space)*, 2 space-el húzza beljebb a szöveget, ahol be kell húzni *(space width)*

set tw=0 Nem töri el a sorokat. Beállítható, hogy hány karakter széles legyen a szöveg *(text width)* szerkesztés közben, és ha szélesebbre nyúlik amit gépeltünk, azt eltöri a megadott szélességnek megfelelően. Pl. ha levelezőből külső editornak használjuk, akkor érdemes egy **:set tw=76** paranccsal kezdeni.

További vim információk

A vimről további információkat a [1] **vimtutor** programból szerezhetünk, valamint a vimben van *online* súgó is, ami a **:help** parancsal érhető el. Ha a súgó hivatkozhat más sugó oldalakra is. Ezek |jelek| közt vannak. Ha az adott témát akarjuk inkább olvasni, akkor üssük le a CTRL-**]** -t. Ha vissza akarunk menni az előző témára, akkor a CTRL-**t** billentyűkombinációt üssük le. Ha egy adott témát akarunk olvasni, akkor azt elérhetjük közvetlenül a **:help** *témanév* parancsal.

1.1.2. A screen bemutatása

Ide jön a Miham előadásának az anyaga

1.1.3. A CVS bemutatása

I can't imagine programming without it... that would be like parachuting without a parachute!

–Brian Fitzpatrick on CVS

El sem tudnám képzelni a programozást enélkül... Olyan mintha ejtőernyőznék ejtőernyő nélkül!

–Brian Fitzpatrick a CVS-ről

Bevezető

A cvs arra jó, hogy több ember együtt tudjon dolgozni egy projekten. Ha valaki használt már más hasonló rendszert, akkor ismerős lehet számára a zárol-módosít-elenged filozófia mely szerint, ha vki egy fájlt módosítani akar, akkor előtte zárolja, majd módosítja és a végén a zárolást elengedi. Ez avval jár, hogy miközben zárolva van a fájl, más nem férhet hozzá, stb. Ez a módszer működik ha a fejlesztők személyesen is ismerik egymást, találkoznak személyesen, és minden face-to-face kommunikációt bevetnek stb. Azonban egy projektet általában a világ számos részén levő fejlesztők fejlesztenek. Ezért helyette a CVS-nél a másol-módosít-összefésül modellt használják. Ami az alábbiak szerint működik:

1. A fejlesztő kér egy működő változatot a CVS-től. Ezt „checkout"nak is hívjuk. Képzeljük el úgy mint amikor egy könyvet kiveszünk a könyvtárból.

2. A fejlesztő szabadon szerkeszti a saját működő másolatát. Ugyanebben az időben más fejlesztők dolgoznak a saját működő másolatukkal, lévén ezek mind külön másolatok. Képzeljük úgy el, mintha mindenki kivette volna a könyvtárból ugyanazt a könyvet, és mindenki írogatná bele a saját megjegyzéseit a margóra, vagy akár teljes lapokat újraírna. (Természetesen tekintsünk el attól a ténytől, hogy ez egy könyvtáros szemében könyvrongálás ☹)

3. A fejlesztő befejezi a változtatásait és beküldi (*commitolja*) egy naplóbejegyzés (*log message*) kíséretében amelyben megmagyarázza a változtatások természetét és céljait. Mintha a könyvtárat tájékoztatnánk a változtatásainkról, és közölnénk, hogy mit és miért változtattunk. Viszont leszúrás helyett a változtatásaink bekerülnének a mesterpéldányba, ahol azok mindörökké meg vannak örökítve.

4. Eközben más fejlesztők megkérdezhetik a CVS-t, hogy lássák, hogy változott-e a mesterpéldány, és ha igen, akkor eszerint frissítse az ő munkapéldányaikat. (*update*)

A CVS felfogásában minden a projektben résztvevő fejlesztő egyenlő. Hogy mikor *commit*oljunk és mikor *update*ljünk, az lehet egyéni is, illetve előírhatja a projekt szabályzata. Egy általános stratégia hogy mindig *update*ljünk mielőtt egy nagyobb változtatásba kezdenénk, és akkor *commit*oljunk, amikor a változtatást befejeztük, az használható, és futtatható. Ilymódon a mesterváltozat mindig „futtatható" állapotban marad.

Természetesen felmerül a kérdés, hogy mi van, ha **A** és **B** fejlesztők dolgoznak a saját példányaikon, eltérő változtatásokat csinálnak, de a változtatásaik érintenek közös területeket is. Ezt *konfliktusnak* hívják, és a CVS figyelmezteti **B** fejlesztőt, ahogy megpróbálja *commit*olni a változtatásait. Ahelyett, hogy engedné **B**nek hogy folytassa, a CVS bejelenti, hogy konfliktust talált és konfliktusjelzőket (könnyen felismerhető szöveges jelzők) helyez a megfelelő pontokra.... Természetesen a konfliktus feloldása a fejlesztőkre vár, a CVS nem tudja kitalálni a helyes megoldást.

Tekintsük át részletesen mi történik a mesterpéldánnyal (, vagy ahogy a CVS terminológia hívja *Repository*val). Nézzük az alábbi szituációt:

1. Két fejlesztő, **A** és **B** *checkout*olják a projekt egy-egy munkaváltozatát ugyanabban az időpontban. A projekt még az elején van, még senki nem komitolt bele így a fájlok még eredeti állapotukban vannak.

2. **A**-ra rájön az alkothatnék, és egy csomót dolgozik a projekten, és utána ezeket *commit*olja.

3. Eközben **B** TV-t néz.

4. **A** továbbra is megihletődött állapotban folytatja a kódolást, és egy újabb adag változást *commit*ol. Így a *Repository* már tartalmazza az eredeti fájlokat, **A** első módosításait, és ezeket a módosításokat is.

5. Eközben **B** armagetronozik.

6. Villámcsapásként az égből **C** is csatlakozik a projekthez, és ő is *checkout*olja a projekt egy munkapéldányát. **C** munkapéldánya már a legutolsó módosítások szerint néz ki.

7. **A**ban még mindig dúl a munkavágy, így egy újabb adag változást *commit*el.

8. Végül **B**nek is megjön az ihlete, és rájön, hogy ideje dolgozni. Nem zavartatja magát avval, hogy *update*lje a munkaváltozatát, egyszerűen csak belepiszkít a fájlokba, esetleg olyanokba is, amiket A is módosított. Ezután **B** *commit*eli a változtatásait.

Ennél az utolsó pontnál több dolog is történhet. Ha **B** semmi olyan fájlon nem változtatott, amin **A** is, akkor a *commit* sikerülni fog. Egyébként a CVS észreveszi, hogy **B** munkaváltozatában némelyik fájlja már idejétmúlt (**outdated**), és hivatkozva a repository utolsó állapotára figyelmezteti **B**-t, hogy *update*lnie kell mielőtt *commit*olna.

Amikor **B** *update*l, a CVS beolvasztja (*merge*li) **A** összes változtatását **B** másolataiba. Esetleg ezek közül némelyik ütközhet **B** még nem *commit*olt változtatásával. Ekkor ezeket **B**-nek fel kell oldania, mielőtt commitel.

Ha most **C** *update*l, akkor számos új változtatást fog kapni a *repozitórium*ból: **A** harmadik *commit*jának eredményét, és **B** első SIKERES változtatásainak eredményét.

A CVS rendszerint változtatásokat szolgál ki, a megfelelő sorrendben. A fejlesztők munkaváltozatainak elavultságától függő mennyiségben küld változtatni-valókat a szinkronizálásokhoz. Ennélfogva a *repozitóriumnak* tárolnia kell a projekt legeleje óta történt összes változást egyenként. A CVS ezeket növekvő *diff*ekként tárolja. Ennélfogva képes, egy akármilyen régi munkaváltozathoz képes előállítani, hogy mi a különbség a jelenlegihez képest, és hatékonyan naprakésszé tenni a munkaváltozatot. Ez a fejlesztők számára lehetővé teszi azt és, hogy a régebbi munkaváltozatokat is áttekintsék, illetve megtudják a különbséget bármely két változtatás közt. (Ami hasznos lehet például, ha valamikor bekerült a *repozitóriumba* egy hiba, és ki tudják deríteni, hogy az melyik változtatásnál, és mivel kerülhetett be, stb.)

Néhány tesztfeladat:

- Elkezdeni egy projektet, és a forrását *import*álni a cvs-be.
- Minden a projekthez tartozó személy *checkout*oljon egy *munkapéldány*t a *repozitórium*ból.
- Dolgozzanak a fejlesztők, majd *commit*eljenek, updateljenek.
- Vegyenek fel a fejlesztők új fájlt a cvs-be, ill. távolítsanak el onnan már nem szükségeset.
- Nézzék meg két adott verzió közt a különbséget.

A használt kifejezések:

Revision Egy kommitelt változás egy fájl vagy egy rakat fájl történetében. A revízió egy „pillanatkép” egy állandóan változó projektről.

Repository A mester másolat ahol a CVS a projekt teljes Revíziótörténetet tárolja. Minden projektnek pontosan egy Repozitóriuma van.

Working copy Egy másolat amin éppen változásokat végzünk a projekt érdekében. Rengeteg ***munkapéldány*** lehet egy adott projektről; általában minden fejlesztőnek megvan a sajátja.

Check out Egy munkapéldány igénylése a repozitóriumból.

Commit Elküldeni a munkapéldány változtatásait a központi reporzitóriumba. (*check-in* néven is ismert.)

Log message Egy magyarázat egy revízióhoz amikor kommitelsz, leírva a változásokat.

Update Kiszedni a változásokat a repozitóriumból, és a munkapéldányban alkalmazni.

Conflict Az a szituáció, amikor két fejlesztő megpróbál változásokat kommitolni ugyanazon fájl, ugyanazon területén. A CVS észreveszi a konfliktust, megjegyzi, rámutat, és a fejlesztőknek kell megoldaniuk.

1.2. A GNU fejlesztő eszközök I. rész. Compiler, optimalizációk használata, speciális módokon való programfordítás, stb.

1.2.1. Bevezető

A **C/C++** programok lefordításához a legelterjedtebb fordító a GNU projekt egyik legfontosabb programja a gcc, ami a **GNU Compiler Collection** rövidítése. A *Compiler Collection* név jogos, hiszen nem csak C hanem C++, illetve Fortran forráskódot is le tud fordítani, sőt a 3.x verziók már a *JAVA* forrásokat is le tudják fordítani, ill. **java byte kód**ot is képesek előállítani.

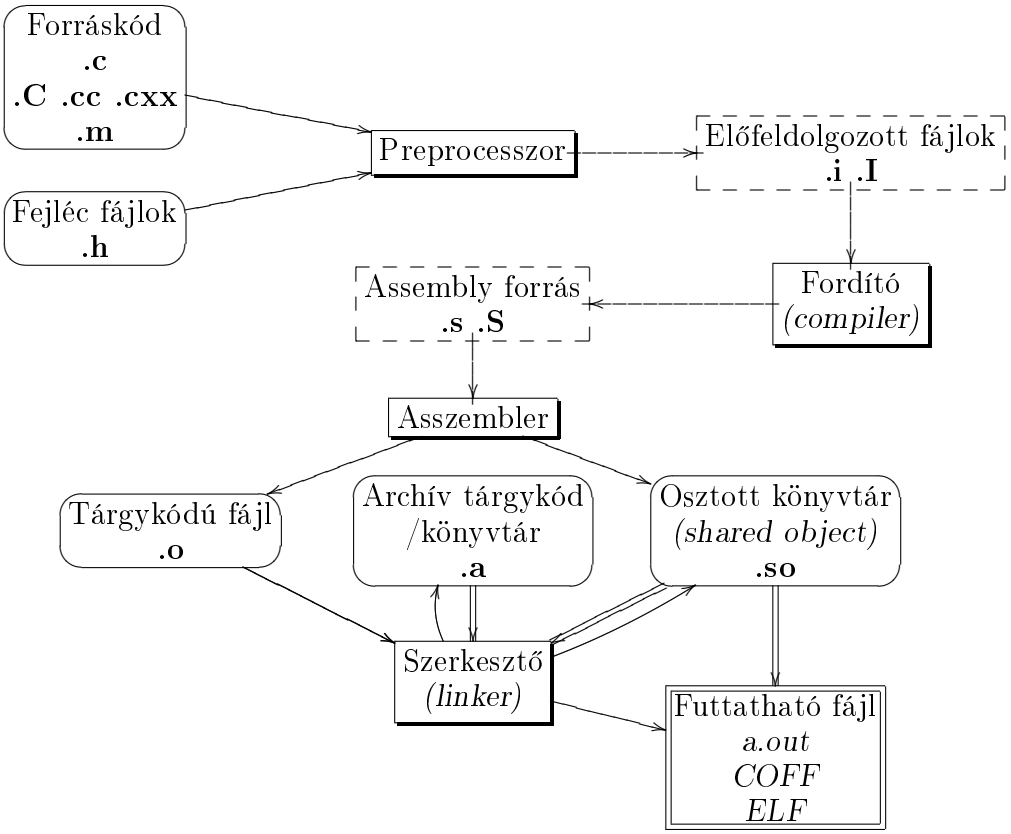
A GNU programozási eszköztára lehetőséget biztosít továbbá arra, hogy a kódot futási idő, vagy a futtatható kód mérete szempontjából optimalizáljuk. Lehetőség van más architektúrákon ill. más platformokon futó program fordítására. A célkódot ui. ez a kettő együtt határozza meg. Az ilyen előre elkészített keresztfordítót általában ennek megfelelően konvencionálisan a következő módon nevezik el: **arch-host-programnév**. *(Megj.: A* **gcc** *parancs a* **g++***-tól mindössze annyiban tér el, hogy a gcc C forráskódnak tekinti az inputját, ill C stílusú linkelést csinál rajta, míg a* **g++** *C++-nak tételezi fel a forrást, és C++ stílusú linkelést hajt rajta végre.)* Példák:

Architektúra	OS	Funkció	parancsnév
Intel 80x86 (x>3)	Linux	C Compiler	i386-linux-gcc
Intel 80x86 (x>5)	Win32	Assembler	i586-mingw32msvc-as
Intel 80x86 (x>3)	GNU/Hurd	Linker	i386-gnu-ld
Intel 80x86 (x>3)	Linux(μClibc)	Preprocesszor	i386-uclibc-cpp

Lásd még:

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-linux/2.95.4/specs
gcc version 2.95.4 20011002 (Debian prerelease)
```

A kezdetek: melyik programozási nyelvet használjuk? Noss a legjobb választás a C nyelv, mert a szinte minden architektúrán le lehet fordítani, és jól optimalizálható. Ezzel szemben, ha más nyelvet használunk, akkor azt általában azért tesszük, mert szükségünk van egy nem szabványos lehetőségére. Ebből kifolyólag esetleg kevesebben értik meg a program forráskódját, ill. a felhasználónak külön fordítót kell még telepítenie, és egyéb galibákkal járhat...Ennek ellenére előfordulhatnak olyan kivételek, amikor mégis inkább más nyelvet célszerű használni, pl. ha a program célközönségénél már várhatóan megvan az adott nyelvhez szükséges fordító. A legtöbb programot továbbfejleszthetőnek írnak meg: Tartalmaznak egy a C-nél magasabb szintű értelmezőt, és magát a programot is ezen a nyelven írják meg. Ennek egy példája az Emacs szövegszerkesző, ami élenjár ebben a technikában.
A szabványos kiterjeszthető interpreter a GNU rendszerekben a GUILE, ami a **scheme** (egy speciális, lisp változat) nyelv megvalósítása. Természetesen az egyéb scriptnyelveken is elfogadott egy GNU projekt, mint pl. a perl, vagy a python. A GUILE a teljes GNU rendszer konzisztenciájának megtartása miatt fontos.*(Ennek „ellenére" én is a perl nyelv alapjait fogom a későbbiekben ismertetni.)*



1.2. ábra. A forrástól a futásig

1.2.2. Kompatibilitás más C fordítókkal

Posix,Ansi,Standard C kompatibilitás és a GNU kiterjesztései

Néhány kivételtől eltekintve a GNU fordítója, a hozzátartozó programok, és könyvtárak*(libraryk)* felülről kompatibilisek a megfelelő Berkeley Unix-ban levővel, és felülről kompatibilis a Standard C-vel, amennyiben az definiálja az adott viselkedést, és felülről kompatibilis a POSIX-al, amennyiben az definiálja az adott viselkedést.

Amikor ezek a szabványok netán ütköznek cészerű lehet felkínálni annak a lehetőségét, hogy valamelyiket betartsa a programunk, és a gcc által adott kiterjesztéseket békén hagyni. Ilyenkor a fordításnál használhatjuk a **--ansi**, **--compatible** valamint a **--posix** kapcsolókat.
A **gcc** egy ANSI kompájler, ennek ellenére sok program lehet, ami nem ilyen. Ebben az esetben lehet használni a **-traditional** opciót, azonban ennek van mellékhatása is, ugyanis a -fwriteable-strings opciót automatikusan bekapcsolja, aminek hatására a szövegkonstansok a TEXT szegmensből a DATA szegmensbe átkerülnek, és így a program futása során felülírhatók lesznek.

Ha használjuk a GNU kiterjesztéseit, akkor esetleg átláthatóbb tisztább lesz a program, de lehet hogy nem fordul le egy nem GNU rendszeren. Bizonyos kiterjesztéseknél egyszerűen megtehető, hogy alternatívákat kínáljunk. Pl. Ha egy függvény egy inline-ként van definiálva, akkor megtehetjük, hogy az INLINE kulcsszót használjuk, amit a fordító inline-ra behelyettesít ha ismeri az inline-t, egyébként pedig semmit se tesz a helyére.
Ha sokat javít a program futásán, akkor mindenféleképp érdemes használni a GNU kiterjesz-

téseit. Ez alól kivételt képeznek a nagy és elterjedt programok (mint pl. az EMACS), amik rengeteg fajta rendszeren elérhetőek. Az ilyen kiterjesztések használata a program használóit „boldogtalanná” teheti, úgyhogy ebben az esetben ellenjavalt.

Feltételes fordítás

Ha egy programban egy ilyen részt használunk:

```
#ifdef VAN_IZÉ
        csinálok_valamit(valamivel);
#else
        mást_csinálok(mással);
#endif
```

akkor helyette javasolt a következőt használni:

```
if(VAN_IZÉ)
        csinálok_valamit(valamivel);
else
        mást_csinálok(mással);
```

Ugyanis a mai fordítók már annyira modernek, hogy mindkét esetben ugyanazt a kódot generálják. Ha olyan esetbe ütközünk, ahol az állítást nem lehetne egyszerűen egy if(...) utasításba ágyazni, mint pl. a

```
HAS_REVERSIBLE_CC_MODE
```

, akkor a következő *workaround* használatos:

```
#ifdef REVERSIBLE_CC_MODE
#define HAS_REVERSIBLE_CC_MODE 1
#else
#define HAS_REVERSIBLE_CC_MODE 0
#endif
```

Sig11

Ha egy ilyen képpel találjuk magunkat szembe:

```
Internal compiler error: cc1 got fatal signal 11
```

Akkor ennek egyik oka lehet az, hogy a gcc egy nagyon bonyolult, és sokat tudó fordító, és valamit elhibáztak benne, és a gcc megpróbál egy olyan memóriaterületre írni, ahova egyébként nem lenne neki szabad. Tehát, akár az is lehet, hogy egy gcc hibába futottunk.

De általában a valószinűbb magyarázat (főleg, ha egy jól tesztelt stabil gcc verziót használunk), hogy a hardver a hibás. Röviden szólva a gcc az egyik legjobb RAM tesztelő program. Ha a gcc egy ilyet mond, akkor lehet még hibás a CPU, az alaplap, vagy valamilyen cache is a RAM-on kívül.

sprintf()

Ha valaki már SunOS-on programozott, és használta rajta a sprintf() függvényt, tapasztalhatta, hogy a visszatérési értéke a string-re matató pointer. Linux rendszereken pedig (az ANSI-t követve) egy egész a visszatérési értéke, ami azt adja meg, hogy hány karaktert tett a string-be.

include fájlok

Általában, hogy milyen fájlokat kell `#include`-al beilleszteni a függvények manualjaiban *(2-es, 3-as szekció)* a SYNOPSYS részben le van írva. Pl.:

```
SYNOPSIS
        #include <stdio.h>

        FILE *fopen (const char *path, const char *mode);
        FILE *fdopen (int fildes, const char *mode);
```

Azonban nagyon oda kell figyelni, mert más rendszereken esetleg eltérő lehet, hogy milyen include fájlokra van szükség, ill. akár a hozzálinkelendő könyvtárakban is lehet eltérés. Pl. socket programozásnál ha egy Solaris-on dolgozunk gyakran szükség lehet az nsl könyvtár hozzászerkesztésére (**-lnsl**), míg a GNU C library tartalmazza a socketprogramozáshoz szükséges függvényeket.

select(), és busy waiting

A BSD **select(2)** manual oldala azt írja, hogy a függvényhívás mellékhatásként módosíthatj a tv (TIMEVAL) értékét, és beírja a még hátralevő időt helyette. Illetve, hogy a későbbi verziók így fogják csinálni. Bizonyos Linux verziók már így tesznek. Bizonyos verziók nem. Így nem célszerű feltételezni, hogy a függvényhívás nem változtatja meg az értéket. Tehát a javaslat, ha így nézne ki eredetileg a kód:

```
struct timeval timeout;
timeout.tv_sec = 1; timeout.tv_usec = 0;
while (some_condition)
        select(n,readfds,writefds,exceptfds,&timeout);
```

Azt cseréljük le, hogy így nézzen ki:

```
struct timeval timeout;
while (some_condition) {
        timeout.tv_sec = 1; timeout.tv_usec = 0;
        select(n,readfds,writefds,exceptfds,&timeout);
}
```

1.2.3. Optimalizáció

-Ox A gcc meghívható a -Ox paraméterrel, ahol x egy kis egész szám. A 0 semilyen optimalizációt sem jelent, míg a -O2 más sok, és a -O3 már nagyon sok optimalizációt. Belsőleg a gcc ezeket az opciókat -f és -m opciókra cseréli le. Ha nagyobb számot adunk meg, mint amekkorát a fordítónk tud, (pl. -O6) akkor a normális működés szerint a lehető legnagyobbat használja. Ennek ellenére nem javasolt nagyobb számot használni a programunk terjesztéseiben, mert lehet, hogy később a fordítóba egy nagyobb optimalizációs lehetőség is bekerül, amivel lefordítva a kódunkat az már működésképtelenné válhat. (Konkrétan a 2.7.0 tól a 2.7.2-ig a -O2 hibásan működött.)

-m486 Ha így fordítunk le egy programot, akkor az nagyobb lesz, de ennek ellenére 386-on is fut. Természetesen a 486-on pedig még gyorsabban, mert a fordításnál a 486-os processzor architektúráját figyelembe veszi. Ha még optimálisabb kódot szeretnénk, akkor érdemes esetleg speciálisan erre kihegyezett programot használni. Pl. a debian-ban adott a **pentium-builder** nevű csomag, ami megfelelő környezeti változók megfelelő értékei esetén pentium-ra optimalizált kódot fordítanak.

1.2.4. PIC

Lásd a következő óra ide vonatkozó részét :-)

1.2.5. Debug és profiling

debug infók fordítása

Ha úgy szeretnénk lefordítani a programot, hogy a debughoz szükséges infók is belekerüljenek, akkor a **-g** kapcsolót kell használnunk a fordítás során. Természetesen emiatt nem kell újrafordítanunk az egész programunkat, elég csak a debugolni kívánt részeket így fordítani.

Ha külön fázisban fordítjuk és szerkesztjük(*linkeljük*) a programot, akkor a linkernek se felejtsük el megadni a **-g** kapcsolót, ahol szükség van rá. A GNU **ld**-ben már csak kompatibilitási okokból van benne.

A debugra használható programok

gdb A GNU debuggere. Egyszerű kis command line-szerű debugger. Van grafikus változata isxxgdb néven.

ddd Ez már grafikus felületen működő debugger, jóval egyszerűbb használni mint a gdb-t. „Kezdőknek” javasolt.

strace Ez nem kifejezetten debugger program. Ezzel a programmal elindítva egy másikat, az strace figyeli, hogy milyen rendszerhívásokat hajt végre a program, és ezeket paramétereistül, gyerekprocesszestül, mindenestül képes listázni, akár pid-enként külön fájlokba. Másik nagy előnye, hogy semilyen extra fordítási opciót nem igényel a program fordítása során, így bármely programmal szemben használható általános hibakereső eszköz.

profiling

A profiling arra szolgál, hogy felderítsük, hogy a programunk, mely részére hányszor kerül a vezérlés. Ha ilyet szeretnénk tenni, akkor a programot a **-p** paraméterrel kell fordítani, majd lefuttatni, ekkor készül gmon.out néven egy „profil" a programól, és utána a **gprof** nevű programmal lehet elemezni, hogy a program mely részén mennyi időt töltött a vezérlés, valamint, hogy hányszor került rá a vezérlés.

Hasonló célú eszköz a gcov is. Példa:

```
$ cat <<END >pelda.c
#include<stdio.h>

int main () {
  int i;
  for(i=0;i<10;i++) {
    printf("%d\n",i);
  }
  printf("A program befejezte működését\n");
}
$ gcc -fprofile-arcs -ftest-coverage pelda.c
$ ./a.out
0
1
2
3
4
5
6
7
8
9
A program befejezte működését
$ gcov pelda.c
100.00% of 6 source lines executed in file pelda.c
Creating pelda.c.gcov.
$ cat pelda.c.gcov
#include<stdio.h>

int main () {
           1      int i;
          11      for(i=0;i<10;i++) {
          10        printf("%d\n",i);
          10      }
           1      printf("A program befejezte működését\n");
           1    }

$
```

1.3. A GNU fejlesztő eszközök II. rész. Linker, linkelés, osztott könyvtár készítés

1.3.1. In medias res példa

Következzen egy egyszerű kis program, ami az osztott könyvtárak készítését szemlélteti:

```
$ cat >proba.h <<'END'
#ifndef __PROBA_H
#define __PROBA_H

extern int duplaz (int x);

#endif /* __PROBA_H */
END
$ cat >proba.c <<'END'
#include<proba.h>

int duplaz (int x) {
  return x*2;
}
END
$ cat >test.c <<'END'
#include<stdio.h>
#include<proba.h>

int main () {
  int x;
  x=duplaz(5);
  printf("%d\n",x);
  exit(0);
}
END
$ cat >test2.c <<'END'
#include <dlfcn.h>
#include <stdio.h>

int main()
{
  void *proba;
  int (* duplaz)(int x);

  if(proba=dlopen("libproba.so",RTLD_LAZY))
  {
    duplaz=dlsym(proba,"duplaz");
    printf("%d\n",(*duplaz)(5));
```

```
  }
  return 0;
}
END
$ cat >Makefile <<'END'
LIBS= libproba.a
SHARED= libproba.so
OBJS= proba.o
CFLAGS= -I.
LDFLAGS= -L.
TARGETS= test tests testsp test2

all: $(TARGETS)

libproba.a: $(OBJS)
        $(AR) rvs $@ $^

libproba.so: $(OBJS)
        $(CC) -shared $(CFLAGS) $(LDFLAGS) -o $@ $^

test: test.c proba.h $(SHARED) $(LIBS)
        $(CC) $(CFLAGS) $(LDFLAGS) -lproba -o $@ $<

test2: test2.c $(SHARED) $(LIBS)
        $(CC) $(CFLAGS) $(LDFLAGS) -ldl -o $@ $<

.PHONY: runtest
runtest: test $(SHARED)
        LD_LIBRARY_PATH=$(PWD) $(PWD)/test

.PHONY: runtest2
runtest2: test2 $(SHARED)
        LD_LIBRARY_PATH=$(PWD) $(PWD)/test2

tests: test.c proba.h $(LIBS)
        $(CC) -static $(CFLAGS) $(LDFLAGS) $< -o $@ -lproba

testsp: test.c proba.h $(LIBS)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $< -Wl,-dn -lproba -Wl,-dy

.PHONY: runtests
runtests: tests
        -$(PWD)/tests

clean:
```

```
        -$(RM) $(LIBS) $(SHARED) $(OBJS) $(TARGETS)
END
$ make
cc -I.   -c -o proba.o proba.c
cc -shared -I. -L. -o libproba.so proba.o
ar rvs libproba.a proba.o
a - proba.o
cc -I. -L. -lproba -o test test.c
cc -static -I. -L. test.c -o tests -lproba
cc -I. -L. -o testsp test.c -Wl,-dn -lproba -Wl,-dy
$ make runtests
/home/pasztor/gnu/tests
10
$ make runtest
LD_LIBRARY_PATH=/home/pasztor/gnu /home/pasztor/gnu/test
10
$ ls -l test testsp
-rwxr-xr-x    1 pasztor  pasztor      5261 sze 29 15:49 test
-rwxr-xr-x    1 pasztor  pasztor      5101 sze 29 15:48 testsp
$
```

A fenti példa egyben mutatja azt is, hogy hogyan futtassuk a programot, valamint, hogy milyen eredményt kellene kapnunk. Mellesleg elég leegyszerűsített példa is. Normál esetben ui. vhogy így kell eljárnunk egy **.so** készítésénél:

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

Noss, akkor vegyük sorba a „gyakorlati” hibákat:

- A linkernek nem lett átadva soname opció. (**-Wl,-soname,libfoo.so.1**)
- Egyáltalán nincs semilyen verziókezelés.
- A „szokásos” módon nem lett létrehozva a symlink a **.so** fájlra.

1.3.2. .so-k verziószámozása

A **.so** fájlok általában a **lib**<*könyvtárnév*>**.so.**<*major/fő verziószám*>**.**<*minor/mellék verziószám*> módon vannak elnevezve. Minden osztott könyvtárhoz tartozik egy *soname*, ami egy szimbolikus név, és nem maga a fájlnév. De facto szabvánnyá vált, hogy ha egy könyvtár neve **libproba.so.1.2**, akkor a hozzátartozó *soname* **libproba.so.1**. A *soname*-ra fog hivatkozni a lefordított program, így amikor a program indul, azon a néven fogja keresni a könyvtárt, amit a fordításkor a *soname*-ből kiolvasott. Ezért is szokás az **ln -s libproba.so.1.2 libproba.so.1** parancsal szimlinket létrehozni, mert ha pl. vmi. apró módosítást, hibajavítást végzünk a

könyvtáron, akkor a *minor/mellék*verziószámot növeljük, és a szimlinket kicseréljük az újra, és a programok képesek az újat használni, újrafordítás nélkül.

Az **ln -s libproba.so.1 libproba.so** parnacsra azért van szükség, hogy amikor a GCC-t a **-lproba** paraméterrel meghívjuk, egy a proba könyvtárunkat használó program fordításakor, akkor a GCC **libproba.so** néven fogja keresni a könyvtárat (, és ebből ki fogja olvasni a *soname* szimbolikus információt).

Megjegyzések:

- A **-Wl,**-vel kezdődő gcc paraméterek a linkernek (LD) adódnak át. A fenti esetben pl. a soname-t kell így átadnunk.
- A GCC-nek a **-fPIC** paramétert át kell adni, ha könyvtárhoz fordítunk forrásokat. PIC=Pozíció Független(Indipendent) Kód(Code)
- A verziók számozásában még „mélyebbre" is lehet menni. Pl. a GNU libc-nél egy példa-verziószám: 2.2.5, **/lib/libc-2.2.5.so**, **/lib/libc.so.6**, **/usr/lib/libc.so**
- Hasonlóan a *soname*-hez az archív nevét is meg lehet adni szimbolikusan a .so fájlban. Ti. ha linkelni szeretnénk egy .so-hoz, akkor meg kell, hogy legyen a .a is.

Gyakorlati feladatok:

- A fenti Makefile megfelelő módosítása.
- Több különböző „funkció" írása, ennek megfelelően a header fájl módosítása, más forrás beépítése, a Makefile hozzáigazítása, stb. **ar**, **ranlib** tanulmányozása.

1.3.3. nm

Ha kíváncsiak vagyunk, hogy egy függvény, melyik modulban van megvalósítva, ill. hol hivatkoznak rá, akkor az **nm** nevű paranccsal ki tudjuk listázni a benne levő szimbólumokat. Pl. **nm /usr/lib/libc.a** Gyakorlati feladatok:

- Kikeresni a libc-ben, hogy lehet a verziószámot kideríteni
- Programot írni, ami a libc verziószámát kiírja
- Kikeresni néhány függvény a libc-ben, pl. fopen melyik modulban van.

1.3.4. ldd

Ha szeretnénk egy lefordított programról megtudni, hogy milyen könyvtárakhoz van hozzá-szerkesztve, akkor az **ldd** parancs jöhet a segítségünkre.
Példa:

```
$ ldd test
        libproba.so => not found
        libc.so.6 => /lib/libc.so.6 (0x40018000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ LD_LIBRARY_PATH=. ldd test
```

```
        libproba.so => ./libproba.so (0x40014000)
        libc.so.6 => /lib/libc.so.6 (0x4001a000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

1.4. A GNU fejlesztő eszközök III. rész. `make`, `Makefile`, stb.

1.4.1. Bevezető

A make parancs segít a programunk lefordításában. Egy nagyobb projekt sok modulból állhat, esetleg más modulokat, más paraméterekkel kell fordítani, stb. Ezen munkát megkönyítendő segít a make. A program elkészítésének „receptjét" a `Makefile`[1] nevű fájl tartalmazza, és a fordítás a make parancsal történik.

1.4.2. A make alapfogalma: target

A make parancsnak meg kell mondani, hogy „mit" csináljon. Ez a target. Hogy hogy csinálja az a *makefile-ban van leírva*. Az alapértelmezett target a makefile-ban definiált első target. Általában programoknál első helyre tesznek egy **all** nevű targetet, ami minden programrészletet lefordít, elkészít, stb. Gyakorlatilag végső állapotba hozza a programot. Szokás még egy **install** nevű target írása is, ami a program telepítését végzi el, illetve egy **clean** nevű target írása, ami a program fordítása közben keletkezett „melléktermeket" eltávolítja.

Íme egy egyszerű példa:

```
CFLAGS = -O2 -Wall
INSTALLPREFIX = /usr/local
INSTALLDIR = $(INSTALLPREFIX)/bin
INSTALL = install -s -p -m 0555
BINARIES = prog1 prog2
SOURCES = prog1.c prog2.c prog.c
HEADERS = prog.h
#Ez egy egyszerű Makefile 3 db forráskód,
#                         1 közös headerfájl,
#                     és 2 célprogrammal.

all: $(BINARIES)

prog1: prog1.o prog.o

prog2: prog2.o prog.o

prog1.o: prog1.c prog.h

prog2.o: prog2.c prog.h
```

[1] A pontosság kedvéért: A make először `GNUMakefile`, majd `makefile` és legvégül `Makefile` nevű fájlt keres. Ennek ellenére `Makefile`-nak javasolt elnevezni a „receptet", ti. a `GNUMakefile`-t specifikusan csak a GNU Make keresi, a `Makefile` pedig a könyvtárlistázás elején fog megjelenni a fontosabb fájlokkal, mint pl. `README`, `INSTALL`, stb. együtt. Ha a make nem talál ezek közül egyet sem, akkor csak az implicit szabályok$_{23}$ lesznek elérhetők.

```
prog.o: prog.c prog.h

.PHONY: install
install:
        $(INSTALL) $(BINARIES) $(INSTALLDIR)

.PHONY: clean
clean:
        -$(RM) -f *.o
```

Egy példa prog1.c

```
#include<stdio.h>
#include"prog.h"

int main () {
  init();
  printf("%d\n",szam);
  return 0;
}
```

Egy példa prog2.c

```
#include<stdio.h>
#include"prog.h"

int main () {
  init();
  printf("%d\n",szam*2);
  return 0;
}
```

Egy példa prog.c

```
#include"prog.h"

int szam;

void init () {
  szam=5;
}
```

Egy példa prog.h

```
#ifndef __PROG_H
#define __PROG_H

extern void init ();
extern int szam;

#endif /*__PROG_H*/
```

Egy szabály leírása a következőképp néz ki:

```
targetnév: függőségek
        parancs
        ...
```

vagy

```
targetnév: függőségek ; parancs
        parancs
        ...
```

A legegyszerűbb target, ahol a targetnél azt szeretnénk, hogy semilyen parancs ne hajtódjon végre: `semmi: ;` Itt nagyon fontos, hogy a ;-t kitegyük. Hasznos lehet egy ilyen targetet pl.

defaultnak betenni, ha azt szeretnénk, hogy egy egyszerű, a könyvtárban kiadott `make` parancs ne csináljon semmit.

Függőségeknél fájlneveket sorolhatunk fel. Nagyon fontos, hogy a rendszer a fájlok utolsó modosítási dátumát jól kezelje. Ti. a következőképp dolgozik a make:

1. Megnézi a függőségek legutolsó módosítási dátumát
2. Megnézi a célfájlt, hogy létezik-e, és hogy annak mi a módosítási dátuma
3. Ha a függőségek közt vmelyik fájl újabb, mint a target, akkor a targetet (újra) el kell készíteni.
4. Ha a targetet (újra) el kell készíteni, akkor a megadott parancsok szerint elkészíti.

1.4.3. Speciális targetek

A .PHONY target

Vannak speciális célok, amelyek nem egy fájlt hoznak létre, hanem valamilyen feladatot látnak el, pl. feltelepítik a programot, vagy a munkafájlokat törlik; mint a clean és az install targetek. Ezek `.PHONY` targetek.

A `.PHONY` targetként megjelölt targeteknél az előbb megadott lépéseknél az 1-3.-t mindíg kihagyja, és mindíg újra elkészíti a targetet a megadott parancsoknak megfelelően, még akkor is, ha pl. a clean-nek nincs függősége, és clean nevű fájl már létezik akármilyen dátummal.

Ha egy targetet `.PHONY` targetként szeretnénk definiálni, akkor azt megelőzve kezdjünk egy `.PHONY` nevű targetet és írjuk be a targetünket a `.PHONY` target függőségeként.

Az üres target jelentősége

Az üres targetek hasonlítanak picit a `.PHONY` targetekre. Itt általában nem számít a fájl tartalma, csak az utolsó hozzáférés dátuma. (Figyelem, itt már megnézi a make a fájlt) Pl. ha rendszeresen ki szeretnénk néhány megváltozott forráskódot nyomatani, akkor használhatjuk a köv. targetdefiníciót:

```
print: prog1.c prog2.c prog.c prog.h
        lpr -p $?
        touch print
```

Itt lesz egy print nevű fájlunk, aminél az utolsó módosítási dátum azt jelzi, hogy mikor nyomtattunk utoljára. A `$?` pedig arra fog behelyettesítődni, amik a függőségek közül megváltoztak a print utolsó módosítása óta. Így csak a megváltozott fájlokat nyomtatja ki a szabály.

.NOTPARALLEL target

Ha ez a target szerepel a makefájlunkban, akkor a make a futása során nem fog párhuzamosítani feladatokat, még akkor sem, ha a -j kapcsolót megadtuk neki. Ha a `.NOTPARALLEL`-nek megadunk valami függőséget, azt a make figyelmen kívül hagyja. Ha a make eleve párhuzamosan fut, mert pl. ez a makefájl egy rekurzív make hívás egyik példánya, attól még a többi make processzt nem fogja kilőni a rendszer.

Önálló feldolgozásra:

.SUFFIXES A `.SUFFIXES` target függőségeként fel kell sorolnunk azon kiterjesztéseket (a kezdő ponttal együtt), amelyeknél szeretnénk, hogy a kiterjesztésekkel kapcsolatos szabály mintákat alkalmazza.[2] Amiatt van szükség erre, mert régebben a szabály minták (Pattern Rules) megléte előtt kiterjesztésmintákat használtak, és a make tudomására kellett hozni, hogy milyen kiterjestésekre vonatkozóan van kiterjesztésminta. Ma azonban már ellenjavalt ezeknek a használata, és helyette a sokkal általánosabb szabálymintákat használjuk.

„Old fashion" példa:

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Új típusú példa:[3]

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

.DEFAULT Ami parancsokat megadunk a `.DEFAULT` targetnek, azt minden egyes target készítésekor le fogja futtatni, amikor nem talál rá szabályt, hogy hogy kell előállítani. (Ide értendő, hogy sem explicit, sem implicit mintát nem talál rá.[4])

.PRECIOUS Ha valamilyen targetet megadunk függőségként a `.PRECIOUS`-nak, akkor azt nem fogja letörölni a make, ha annak a targetnek az előállítása közben megszakad. Példa: Ha a prog1 -et készíttetjük a make-el, és a prog1.o mint függőség készítése közben megszakadna a make futása, pl. egy CTRL-c hatására, akkor nem fogja a prog1.o-t letörölni.

.SILENT Ha megadunk targeteket függőségként a `.SILENT`-nek, akkor azon targetek készítése közben nem fogja a make kiírni, hogy épp milyen parancsot futtat. Ha csek felsoroljuk a `.SILENT` targetet, függőségek nélkül, akkor a make úgy tesz, mintha minden target a `.SILENT` függősége lenne. Lásd még a *Parancsok kezelése* 26 részt.

.IGNORE Ha megadunk targeteket függéségként a `.IGNORE`-nek, akkor azon targetek készítése közben a make nem fog leállni, ha egy adott parancs hibával[5] fejezi be működését. Ha csak feltüntetjük az `.IGNORE` targetet, de nem adunk meg függőséget, akkor a make úgy tesz, mitnha minden target a `.IGNORE` target függősége lenne. Lásd még a *Parancsok kezelése* 26 részt.

[2] Jelen esetben semilyen tulajdonság kiterjesztéséről nincs szó. Mindössze a fájlok végén a pont után következő részt a magyar szakirodalomban a fájl kiterjesztésének hívja.

[3] A régi típusú kiterjesztés mintáknál nem lehetett megadni függőségeket egy mintához, ha megadtnánk pl. egy `izé.h` fájlt függőségként, akkor azt úgy értelmezné, hogy hogy kell az `izé.h` -ból elkészíteni a `.c.o-t`

[4] Lásd még: make infó oldalainál az implicit mintakeresési algoritmust

[5] Un*x rendszerekben, minden lefuttatott parancs visszaad egy hibakódot. Ha ez 0, akkor a parancs jól lefutott, ha nem, akkor a hibakódból általában lehet következtetni a hiba jellegére, de ez minden parancsnál egyéni, hogy a hibakódba hogyan kódolja bele a hiba jellegét. Fontos, hogy ne keverjük a C programozási nyelv `true`/`false` kezelésével, aholis a 0 jelenti a hamis értéket.

Többszörös szabályok

Ha a : előtt több targetet megadunk, és függőségeket felsorolunk, akkor a make az összes target függőségeihez hozzáveszi a megadott függőséget. Példa:

```
kbd.o command.o files.o: command.h
```

Ha parancsokat is megadunk, az is működik, és többszörözi a definíciót. Ha például vesszük a következő definíciót:

```
nagykimenet kiskimenet : rizsa.g
        generate rizsa.g -$(subst kimenet,,$@) > $@
```

Akkor ez a definíció ekvivalens az alábbi hosszabb definícióval:

```
nagykimenet : rizsa.g
        generate rizsa.g -nagy > nagykimenet
kiskimenet : rizsa.g
        generate rizsa.g -kis > kiskimenet
```

Implicit szabályok

Az implicit szabályok a make által már eleve ismert szabályok, amiket nem kell neki külön elmondani. A fenti példában$_{19}$ csak annyit mondtunk el a make-nek, hogy a `prog.o` függ a `prog.c`-től ill `prog.h`-tól, de hogy a `prog.o`-t, hogy kell előállítani, azt már nem közöltük vele. Ilyenkor a make implicit szabályokat használ. Pl. egy implicit szabály, hogy hogyan kell lefordítani (kompilálni) egy C programot:

```
x.o: x.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS)
```

Az implicit szabályok katalógusa megtalálható a make info oldalai közt.[6]

1.4.4. Szabály minták

Ha nem egy konkrét targetet akarok megadni, hanem egy általános dologra szeretnék egy általános szabályt adni, akkor használatosak a szabály minták. Pl. ha le akarom írni, hogy hogy készül egy .pdf fájl, akkor vmi. ilyesmit használok:

```
%.pdf: %.tex
        pdflatex $<
        makeindex $*
        pdflatex $<
```

[6] Órai feladat: keresd meg, minél gyorsabban az ide vonatkozó info oldalt!

1.4.5. Archívumtagra hivatkozás

Ha egy archív fájl egyik tagjára szeretnék hivatkozni targetként, vagy függőségként, akkor azt a következő szintaxis szerint adhatom meg: `ARCHÍVNÉV(TAGNÉV)`

Például, ha szeretném a make-el közölni, hogy a `libproba.a` -ban levő `proba.o` a `proba.o`-tól, függ, akkor a következő függőséget kell megadni a make-nek:

```
libproba(proba.o): proba.o
```

1.4.6. Változók

A `Makefile`-ban szokás változókat is használni, bizonyos dolgok egyszerűsítésére. Például előre beállítva kapunk CC változót, ami a C compiler nevét tartalmazza. De a make implicit szabályai sem közvetlenül a cc vagy gcc parancsot adják ki, hanem a CC változóból olvassák ki a parancs nevét.

Emiatt jónéhány változó értékét előre beállítva megkapjuk.[7]

Az értékadások, a targetektől függetlenül, már a make meghívásakor első lépésként lefutnak.

Az értékadás módjai

- Egyszerű értékadás: `VÁLTOZÓ = ÉRTÉK` Ilyenkor a változó felveszi az értéket.
- Egyszerű értékadás, azonnali kiértékeléssel: `VÁLTOZÓ := ÉRTÉK` Ilyenkor ha az érték egy kifejezést, pl. másik változó értékét tartalmazza, az azonnal behelyetesítődik.
- Feltételes értékadás: `VÁLTOZÓ ?= ÉRTÉK` Ha a változónak még nem volt értéke, akkor veszi csak fel a megadott értéket.
- A változó értékének bővítése: `VÁLTOZÓ += ÉRTÉK` A változó értékét megtartja, és a végéhez még hozzáírja a megadott értéket. Tipikus használat: az lefordítandó/*make*elendő alkönyvtárak meghatározására.

Automatikus változók

`$@` A szabály targetjének fájlneve. (Az esetleges bevezető könyvtárnévvel együtt.) Ha archívtagra hivatkozunk, akkor csak az archív fájl neve.

`$%` Csak archívra hivatkozásnál használatos. Az archívban levő résztvevő (member) neve.

`$<` Az első függőség neve.

`$?` Az összes olyan függőség neve, ami frissebb mint a cél. Ha archívtagra hivatkozunk, akkor az azt tartalmazó fájl neve fog a felsorolásba bekerülni.

`$^` Az összes függőség neve. Ha egy függőség a függőségek közt többször előfordul, az itt csak egyszer jelenik meg. Ha archívtagra hivatkozunk, akkor az azt tartalmazó fájl neve jelenik meg.

[7] Órai feladat: kikeresni az info oldalakból az előre beállított változókat.

`$+` Ugyanaz, mint az előző. Viszont ha egy függőség a függőségek közt többször is meg van jelölve, akkor ebben a felsorolásban is többször fog előfordulni.

`$*` Szabály mintáknál az illeszkedés után megmaradt rész. Ha pl. a szabályminta `a%.b`, és az `eperlekvár/abroncs.b` a targetünk, akkor a `$*` értékre `eperlekvár/broncs` lesz.

A define direktíva

Ha egy változónak több soron kersztül szeretnénk értéket adni (úgy, hogy a sortörés is benne legyen az értékében), akkor a define direktívát szokás használni. Szokás verbatim értékadásnak is hívni, hasonlóan a TEXverbatim környezetéhez. Ilyenkor a változónak értékadás a következőképp megy:

```
define VÁLTOZÓNÉV
érték első sora
érték második sora
s.í.t.
endef
```

1.4.7. Elágazások, stb.

A make lehetőséget ad, hogy valamilyen feltétel szerint elágazzon a `Makefileunk`. Ha pl. szeretnénk ellenőrizni, hogy a C fordítónk, biztos a GNU C fordítója, akkor a következő példa szerint tudunk eljárni:

```
izé: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o izé $(objects) $(libraryk_a_gcc_esetén)
else
        $(CC) -o izé $(objects) $(libraryk_normál_esetre)
endif
```

include

A make lehetőséget ad arra, hogy ne csak egy fájlból álljon az a komplex szabályrendszer, amivel egy nagyobb projektet elkészítünk, hanem szétdarabolhassuk több fájlra, és a végén a make egybeolvasztja őket. Erre szolgál a make include direktívája[8].

szövegek manipulálása

A make lehetőséget ad arra, hogy egy változó, vagy konstans szöveg értékét valamilyen módosítás szerint használjuk fel. Példák:

`$(dirname NEVEK)` A fájlhivatkozásból a könyvtár nevét adja vissza.
Pl a `$(dirname izé/mizé.txt brumi)` kifejezés a `izé/ ./` szöveget fogja megadni.

[8]Ez a GNU make specialitása többek közt, más make várhatóan nem fogja ismerni!

`$(suffix NEVEK)` A fájlnévhivatkozásből a fájl kiterjesztését adja vissza.
Pl a `$(dirname izé/mizé.txt brumi)` kifejezés a `.txt` szöveget fogja megadni.

`$(basename NEVEK)` A fájlnévhivatkozásből a fájl nevét adja vissza.
Pl a `$(dirname izé/mizé.txt brumi)` kifejezés a `izé/mizé brumi` szöveget fogja megadni.

`$(wildcard MINTA)` A shellben szokásos wildcardokkal megadott mintával itt is meg lehet adni mintát, és a kifejezés értéke az lesz, mintha a shell-el értékeltettük volna ki a wildcardot, vagyis a mintánkat a létező fájlok neveire fogja illeszteni, és azok közöl a létezők nevei kerülnek be az eredménybe.

`$(subst MIT,MIRE,SZÖVEG)` A megadott szövegben a megadott alrészt kicseréli a megadottra.
Pl a `$(subst irgum,burgum,irgumburgum)` kifejezés a `burgumburgum` szöveget fogja megadni.

`$(origin VÁLTOZÓNÉV)` Azt adja meg, hogy a változó hol vette fel az értékét. (Már feltéve, ha van értéke.) Néhány lehetséges érték: `undefined`, ha soha nem lett definiálva, azaz nincs értéke; `default`, ha valamilyen alapértelmezett értéke van, mint pl. a `CC` változónak, `environment` ha környezeti változó értékeként használja a make saját változójának értékéül.

`$(shell PARANCS)` Külső shell-ben lefuttatja a parancsot, és a futás eredményéül a standard kimenetre adott szöveget veszi a változó értékéül. Gyakorlatilag ugyanaz, mint amit a legtöbb shellben a backtick (‘) csinál.

1.4.8. Parancsok kezelése

A parancsoknál bizonyos esetben nem szeretnénk látni magát a parancsot, csak esetleg a parancs kimenetét. És ezt a szabályon belül nem az összes parancsra szeretnénk, mint ahogy azt a .SILENT targetek teszik. Ilyenkor a parancs elé egy `@` jelet kell tenni, mint az alábbi példában:

```
love:
        @echo "not war!"
```

A másik ami egy paranccsal történhet, hogy hibával fut le. Ilyenkor a make megszakítja működését, és a tempfájlokat letörli. De szintén lehet olyan eset, amikor hagyni akarjuk, hogy hibával fusson le egy parancs. Tipikus példa a clean target, amit ha kétszer egymás után lefuttatunk, akkor nem lesz mit letörölnie, és másodjára már hibával futna le a clean. Ezért ilyenkor egy `-` jelet szokás a parancs elé tenni.[9]

```
clean:
        -rm *.o
```

[9]Mind a `-`, mind a `@` esetén a jeleket a kezdő tab után kell tenni!

1.4.9. Egy komplex Makefile példa

```
PHONY: default
default:
        @echo "Usage: make dir.iso [VOLNAME=volumename] [TARGDIR=targetdirectory]"


PHONY: *.iso

%.iso:
ifeq ($(origin VOLNAME), undefined)
define VOLNAME
$*
endef
endif
ifeq ($(origin TARGDIR), undefined)
define TARGDIR
.
endef
endif
        echo "volname=$(VOLNAME) - $* - targetdir=$(TARGDIR)"
        find "$*" -type d | while read x; \
        do ( cd "$$x" ; echo -ne '' | md5sum -vb `find -type f -mindepth 1 \
           -maxdepth 1 \! -name md5sum.txt -printf '%f\n'` >md5sum.txt ) ; \
        done
        mkisofs -hide-joliet-trans-tbl -gui -l -J -L -r -T -V "$(VOLNAME)" \
                 -o "$(TARGDIR)/$*.iso" "$*"
```

A makefájl használatáról, működéséről néhány szó

Alapértelmezett targetjének a neve `default`, ez hajtódik végre, ha semmilyen paramétert nem adunk, a make-nek, csak egyszerűen elindítjuk a make parancsot. Ekkor kiírja, hogy hogyan is érdemes/lehet használni.

Egyébként a makefile segítségével egy .iso image generálható le. Feltételezzük, hogy a makefile-t tartalmazó könyvtár egyik alkönyvtárába összegyűjtöttük, amit a CD-re szeretnénk írni. Ekkor a make-nek a könyvtár nevét kell paraméterül megadni .iso kiterjesztéssel, ui. úgy készült a makefile, hogy a .iso kiterjesztésű targetekre van benne egy szabály minta.

Alapértelmezésben a .iso fájl, az aktuális könyvtárba kerül, illetve a lemez kötetcimkéje is az lesz, ami a könyvtár neve. Ezek felülbírálhatók a TARGDIR és VOLNAME változóknak történt értékadással.

Ha szeretnénk eltérő kötetcimkenevet akkor a következőképp kell a make-et meghívni:

```
$ make könyvtárnév.iso VOLNAME="Ez lesz a CDcimke"
```

Ekkor a make előre beállítja a VOLNAME változót, és az `$(origin VOLNAME)` már nem lesz `undefined`. Mint említettem, környezeti változóként is át lehet adni egy változó értékét. Ezt kétféleképp is szokás. Szokás úgy, hogy permanensen beállítjuk a shellünknek az adott környezeti változót, pl. bash használata mellett így nézne ki a példa:

```
$ export VOLNAME="Ez lesz a CDcimke"
$ make könyvtárnév.iso
```

Illetve lehet úgy is, hogy csak a make kapja meg az adott környezeti változót, az adott értékkel:

```
$ VOLNAME="Ez lesz a CDcimke" make könyvtárnév.iso
```

1.4.10. Összefoglalás

Röviden tekintsük át, mi lehet egy `Makefile`-ban.

- Egy explicit szabály, ahol elmondjuk, hogy egy fájlt hogyan, és mikor készítsen el, miktől függjön, és megadjuk az elkészítéshez való parancsokat is, stb.
- Egy implicit szabály, ahol elmindjuk, hogy egy fájlt hogyan, és mikor készítsen el, stb.
- Változódefiníció, ahol egy változónak értéket adunk, (vagy az értékéhez hozzáteszünk,) valamilyen módon. Általában a `Makefile` egyszerűsítése és jobb áttekinthetősége miatt használjuk.
- Egy direktíva, ami a következők közül valamelyik:
  - Beolvastat egy másik makefilet az include direktívával.
  - Döntést hoz valamilyen változó, vagy feltétel alapján.
  - Definiál egy változót verbatim módon több soron keresztül a define direktívával.
- Megjegyzés a `Makefile` írójának. A megjegyzés sorok a make-ben `#`-al kezdődnek.

1.5. A GNU fejlesztő eszközök IV. rész. `Makefile` konvenciók

1.5.1. A makefájlírás konvenciói

A `make`ben használható parancsok

A make infó oldalai közt azt találjuk, hogy ha jól hordozható `Makefile`-t szeretnénk írni, akkor a következő parancsokat használjuk csak:

`cat cmp cp diff echo egrep expr false grep install-info ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true`

A leírás kitér arra is, hogy a kompatibilitás jegyében semilyen karácsonyfa[10] jellegű shell-t ne használjunk,[11] hanem csak az alap `sh` lehetőségeire építsünk. A fejezet hátralévő részében ezen parancsok közül fogjuk részletezni a szöveg manipulálására szolgáló parancsokat, a többi ui. fájlkezeléssel kapcsolatos Un*x alapparancs.

A `make` által használt parancsok, és az ezekhez kapcsolódó előre definiált változók

A make az építkezés során a következő parancsokat használja fel:

`ar bison cc flex install ld ldconfig lex make makeinfo ranlib texi2dvi yacc`

Ennek megfelelően definiálja hozzájuk a következő változókat:

`$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX) $(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)`

Ha ezen programok helyett egy másikat szeretnénk használni, pl. gyorsan szeretnénk egy programot sun környezetben lefordítani a GNU C Compilerrel a sun saját `cc`-je helyett, akkor elég a `CC` környezeti változó értékét a `Makefile` elején átdefiniálni `gcc`-re.

További megjegyzések

- Ha szimbolikus linkeket használunk, akkor egy ún. *fallback* lehetőséget definiáljunk az olyan rendszerekhez, amelyek nem támogatják a symlinkeket.
- További programok is használhatók a `Make` változóin keresztül:
  `chgrp chmod chown mknod`
- Ezen túl csak akkor használjunk más programokat, ha tudjuk, hogy az az adott célrendszerben létezik.[12]

A fájlok telepítésével kapcsolatos konvenciók

Minden `Makefile`-ban az `INSTALL` változó definiáltnak javasolt.[13]

[10] túl sok extra tudással ellátott program, mint itt a shellek esetében a `bash`, `csh` vagy a `ksh`.

[11] Ennek hátterében az áll, hogy a `make` a programokat az `sh -c` parancsnak argumentumul adva hívja meg a célok elkészítése érdekében.

[12] Érdemes az ilyenekre felhívni az `INSTALL` vagy `README` fájlokban felhívni a figyelmet.

[13] Láthatjuk a fentiekből, hogy a `Make` már előre definiálja hozzá az `install` programot.

Ezek után javasolt még egy `INSTALL_PROGRAM` és egy `INSTALL_DATA` változó definiálása is.[14] Ez esetben a telepítés valahogy így néz ki a változók használatával:

```
install:
        $(INSTALL_PROGRAM) izé $(bindir)/izé
        $(INSTALL_DATA) libizé.a $(libdir)/libizé.a
```

Illetve ha még korrektebbül szeretnénk eljárni, akkor a telepítési célkönyvtárban még a `DESTDIR` változót is megadjuk prefixként, ezzel lehetővé tesszük a programunkat felhasználónak, hogy

- Ne egyből az éles rendszerre telepítse a programunkat, hanem csak egy alkönyvtárba
- Könnyebben készítsen `.deb` vagy `.rpm` csomagot a programunkból
- Egyszerűen tudja/tudjuk terjeszteni a programunkat „csak bináris" formában, így megkímélve a felhasználót egy esetleges hosszú programfordítástól.

Ezek után a targetünk leírása a következőre módosul:[15]

```
install:
        $(INSTALL_PROGRAM) izé $(DESTDIR)$(bindir)/izé
        $(INSTALL_DATA) libizé.a $(DESTDIR)$(libdir)/libizé.a
```

A `bindir`-hez hasonló változók felsorolva megtalálhatók a `make` infó oldalai közt a Könyvtár Változók[16] pontja alatt részletezve. Egy felsorolás ezen változókról:
`prefix exec_prefix bindir sbindir libexecdir datadir sysconfdir sharedstatedir localstatedir libdir infodir lispdir includedir oldincludedir mandir man1dir man2dir ... manext man1ext man2ext ... srcdir`
Ha a programunk sok fájlt telepít, akkor ajánlott a fájljait külön alkönyvtárba csoportosítani. Pl. `/usr/share/emacs`

konvencionális targetek a `make`-ben

all Lefordítja a teljes programot. Ennek kell az alapértelmezett targetnek lennie. Ennek a targetnek nem kell újrafordítani a dokumentációt; az `info` fájlokat normál esetben tartalmazza a terjesztés, és a `.dvi` fájlokat csak akkor kell megépíteni, ha azt külön kérjük. Alapértelmezésben a `make` a programokat a `-g`-vel fordítja le, így a futtatható fájlok tartalmazzák a debug információkat is. Ha felhasználónak nincs rá szüksége, akkor a strip-el ezeket el lehet távolítani.

install Lefordítja a teljes programot, és felmásolja a futtatható állományokat, osztott könyvtárakat, stb. a végső helyükre a használathoz. Ha adva van egy teszt ami ellenőrizni tudja, hogy a program korrektül fel van-e telepítve, akkor ezt a tesztet is le kell futtatnia

[14] pl. a programot a telepítés után futtatható jogosultságokkal telepít az `INSTALL_PROGRAM`, míg az `INSTALL_DATA` csak olvashatónak, esetleg a tulajdonosa által felülírhatónak, telepíti a programhoz tartozó egyéb adatfájlt.

[15] Javasolt konkrétan fájlnevet megadni az install parancsok második argumentumaként és nem könyvtárnevet; valamint minden egyes feltelepítendő fájlt külön parancsal telepítsünk.

[16] Directory Variables

ennek a targetnek. A futtatható fájlokat ne *strip*-eljük amikor telepítjük őket, erre az **instal-strip** targetet használjuk.

A target nem változtat meg semilyen fájlt abban a könyvtárban ahol készült. Így alkalmas rá, hogy egyik felhasználó nevében leforduljanak a programok, és egy másik (pl. a rendszergazda) nevében települjenek.

A parancsoknak létre kell hozni minden könyvtárat ahova fájlokat telepít, ha azok a könyvtárak még nem léteztek. Lásd pl. `prefix`, `exec_prefix` változókat.

Használjunk a már ismert - jelet a parancsok előt amikor man oldalt telepítünk, így a `make` figyelmen kívül hagyja a hibákat. Hasznos lehet abban az esetben ha olyan rendszerre telepítjük, amire nincs feltelepítve a Unixok man oldalrendszere.

Az infó oldalak telepítéséhez az `infodir` változót használjuk az `$(INSTALL_DATA)`-val, és utána futtassuk az `install-info` parancsot ami frissíti az Info könyvtár 'dir' fájlját és frissíti a menü bejegyzéseket az adott Info fájlról, ami része a Texinfo csomagnak. Mintapélda Info fájl telepítéséhez:

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# There may be a newer info file in . than in srcdir.
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
# Run install-info only if it exists.
# Use 'if' instead of just prepending '-' to the
# line so we notice real errors from install-info.
# We use '$(SHELL) -c' because some shells do not
# fail gracefully when there is an unknown command.
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
           install-info --dir-file=$(DESTDIR)$(infodir)/dir \
                $(DESTDIR)$(infodir)/foo.info; \
        else true; fi
```

Az **install** target írásakor a parancsokat három csoportra szét kell osztani. A telepítés előtti, és utáni, valamint normál részre. A normális parancsoknak kell a fájlokat bemásolni a megfelelő helyre, és beállítani a módot (tulajdonost, stb.). Ez a rész nem változtathat fájlokon kivéve azokat amelyek abból a csomagból jönnek, ahova tartozik. (Vagyis strip-elve lehet telepíteni pl. a futtatható fájlokat, mert a futtatható fájl ugyanahhoz a csomaghoz tartozik, mint a hozzátartozó `Makefile`.)

A telepítés előtti rész a normál parancsok előtt futnak le, és a telepítés utániak a normál rész után.

Egy tipikus példa az install utáni részre, amikor az `install-info` parancsot le kell futtatni az Info fájlok telepítése után.

A legtöbb programnak nincs szüksége semmilyen telepítés előtti parancsra.

Az installálás előtti és utáni részben csak a következő parancsokat használjuk:

`[ basename bash cat chgrp chmod chown cmp cp dd diff echo egrep expand expr false fgrep find getopt grep gunzip gzip hostname install install-info kill ldconfig ln ls md5sum mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee test touch true uname xargs yes`

uninstall Törli az összes telepített fájlt, azokat a másolatokat amelyeket az **install** target létrehoz. Ez a szabály nem módosítja azt a könyvtárat, ahol a fordítás történik, csak azokat ahova fájlok települnek. Az uninstallációs parancsokat három kategóriába szokás sorolni, az installálási parancsokhoz hasonlóan.

install-strip Hasonló az **install** targethez, de a futtatható fájlokat stripeli, amikor telepíti őket. Sok esetben ezen target definíciója az alábbi módon leegyszerűsül[17]:

```
install-strip:
        $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
                install
```

clean Töröl minden fájlt az aktuális könyvtárból amelyek normál esetben a program fordítása közben keletkeznek. Ne töröljünk fájlokat amelyek konfigurációt tartalmaznak. Szintén megőrzi azokat a fájlokat, amelyeket elő lehet állítani, de normál esetben a terjesztés nem tartalmazza.

Törli a `.dvi` fájlokat amelyek nem részei a terjesztésnek.

distclean Töröl minden fájlt az aktuális könyvtárból amelyek a program konfigurálásakor vagy fordításakor keletkeznek. A forrás kibontása és megépítése után ha semmilyen egyéb fájlt nem csinálunk, akkor a `make distclean` csak olyan fájlokat hagy meg amelyek a terjesztésben maradnának.

maintainer-clean Letöröl mindent amit a `Makefile` segítségével újra elő lehet állítani. Tipikusan tartalmazza a **distclean** által törölt fájlok törlését, továbbá: C forrás fájlokat amelyeket a *Bison* állít elő, tag táblákat, Info fájlokat, stb.

TAGS Frissíti a tag fájlokat. Lásd ctags, etags.

info Legenerálja a szükséges Info fájlokat. A javasol mód ennek megtételére az alábbi mintát követni:

```
info: izé.info

izé.info: izé.texi fejezet1.texi fejezet2.texi
        $(MAKEINFO) $(srcdir)/izé.texi
```

dvi Legenerálja a DVI fájlokat a Texinfo dokumentációhoz. Példa:

[17] Természetesen, ehhez is a konvenciók betartása szükséges

```
dvi: izé.dvi

izé.dvi: izé.texi fejezet1.texi fejezet2.texi
        $(TEXI2DVI) $(srcdir)/izé.texi
```

dist A terjesztés tar fájlát létrehozza. A tar fájl úgy jöjjön létre, hogy azon belül egy alkönyvtár legyen, amelynek a neve a programunk nevét és esetleg annak verziószámát tartalmazza.

check Leellenőrzi a programot. Ennél a résznél nem szabad a `$(bindir)` változó tartalmára hagyatkozni. A program megépítését és telepítését a felhasználó dolga a teszt előtt elvégezni.

installcheck

installdirs

1.6. A GNU szövegfeldolgozó eszközök (sed, grep, awk), shellscriptek, és regexpek

Munkánk során gyakran szükségünk van különböző - szöveges - fájlok feldolgozására (keresés egy fájlban, bizonyos sztringek törlése/cseréje, stb...) Ezen feladatok elvégzésére C nyelvű programok írása túl időigényes lenne, vannak ennél sokkal egyszerűbb eszközök is. Három ilyen szövegfeldolgozó (szűrő) program az `awk`, a `grep` és a `sed`. Mindegyik program használatához szükségünk lesz a reguláris kifejezések ismeretére és használatára.

1.6.1. Regexp alapok

Reguláris kifejezések (regular expressions, RE) az alábbi karakterekből állíthatóak össze:

. Tetszőleges karaktert helyettesít.

+ Az előtte álló kifejezés egy vagy többszöri előfordulását írja elő.

***** Az előtte álló kifejezés nulla vagy többszöri előfordulását írja elő.

? Az előtte álló kifejezés egy vagy nulla előfordulálst írja elő.

[] Tartományra, felsorolásra szolgáló kifejezést. Pl. `[a-z]` az angol abc kisbetűi, vagy pl. `[a-f,A-F,0-9]` hexa számjegyek.

^ Sor elejét jelző kifejezés.

$ Sor végét jelző kifejezés.

() Kifejezéseket egy kifejezéssé összefogó jel.

1.6.2. Egyszerű helyettesítések

A fentiek alapján néhány - egyszerű - reguláris kifejezés a következők:

a Erre illeszkedik az „a” betű.

.a Erre illeszkednek az „aa”, „ba”, „7a”, stb. sztringek.

.*a Erre illeszkedik az összes „a” betűre végződő szó.

.+a Az egyedül álló „a” betűn kívül minden illeszkedik rá az előző pontból.

1.6.3. Helyettesítéshez használt eszközök

A reguláris kifejezések önmagukban nem sokat érnek, viszont megfelelő programokat segítségül hívva hatékony eszközökké válnak.

A grep szűrőprogram

A `grep` a bemenetéről beolvasott sorok közül azokat írja ki - alapértelmezésben - a kimenetére, amelyek illeszkednek („matchel”-nek) a megadott reguláris kifejezésre.

Néhány kapcsolója:

-b Megadja a megtalált kifejezés bájt-pozícióját is.

-c Csak a sorok számát adja vissza (melyekben a kifejezést megtalálta).

-f FÁJL Az illesztési mintát (a reguláris kifejezést) a megadott fájlból veszi.

-H Kiírja annak a fájlnak a nevét, amelyben az illeszkedő sort találta.

-i Figyelmen kívül hagyja a kis- és nagybetűk közti különbséget.

-l Csak azokat a fájlneveket adja meg, amelyekben a keresett kifejezés megtalálható.

-n A szűrőn átengedett (megtalált) sorok számát adja vissza.

-r Rekurzív keresés minden alkönyvtár minden fájljában.

-s Nem jelenít meg hibaüzenetet, ha egy fájlt nem tud olvasni (nem létezik, vagy nincs rá megnyitási jogunk).

-v Azokat a sorokat keresi meg, amelyek nem tartalmazzák a megadott kifejezést

A `GREP_OPTIONS` környezeti változóban megadhatjuk az alapértelmezett kapcsolóinkat.

A sed szövegszerkesztő

A `sed` (Stream EDitor) a felhasználó közreműködése nélkül (nem interaktívan) módosításokat hajt végre az inputján, így nagyon jól használható például szkriptekben szöveg törlésére/cseréjére/beszúrására. A `sed` működésének elve:

1. Következő sor beolvasása, ha van ilyen (ha nincs, akkor kilép).
2. A beolvasott sor feldolgozása. („Van-e olyan utasításom, amely alapján ezzel a sorral kell csinálnom valamit?”)
   - ha van, akkor az utasítás végrehajtása, és az eredmény kiírása a képernyőre („pattern space” $\approx$ aktuális output);
3. GOTO 1.

Parancssori argumentumai:

-n Elkészíti az output-fájlt, és nem írja ki a képernyőre a feldolgozás eredményét.

-e SZKRIPT Az inputon végrehajtja a - SZKRIPT-ként - megadott utasításokat.

-f SZKRIPT-FÁJL A SZKRIPT-FÁJL-ban megadott utasítássorozatot hajtja végreaz inputon.

A `sed` ezeken kívül rendelkezik jópár paranccsal, közülük néhány a következő:

y/FORRÁS/CÉL Kicseréli a FORRÁS-ként megadott karaktert a CÉL-ként megadott karakterre.

a\SZÖVEG A megadott SZÖVEG-et hozzáfűzi az inputhoz.

i\SZÖVEG A megadott SZÖVEG-et beszúrja az inputba.

c\SZÖVEG Kicseréli az illeszkedő sorokat SZÖVEG-re.

= Kiírja az aktuális sor sorszámát.

r FÁJL A megadott FÁJL tartalmát beszúrja az outputba.

w FÁJL A megadott FÁJL-ba írja az aktuális outputot.

d Törli az aktuális outputot, és folytatja az input feldolgozását.

p Kiírja az aktuális outputot.

q Kilép a `sed`-ből az input további feldolgozása nélkül.

s/REG-KIF/ÚJSZÖVEG/ A megadott reguláris kifejezést keresi az inputban, és ha talál rá illeszkedőt, akkor kicseréli ÚJSZÖVEG-re.

{,} Utasításblokkot határoz meg.

Az awk programozási nyelv

Az `awk`[18] program tulajdonképpen egy interpreter (amely `awk` programozási nyelven megírt programjainkat értelmezi :-)). Használatával - például - egyszerű adatbázis(szerűsége)t tudunk üzemeltetni, statisztikákat, kimutatásokat, stb. készíthetünk.

A részletes magyarázat előtt vegyünk egy nagyon egyszerű példát! Számoljuk össze a `/dev` könyvtárban lévő blokkos eszközök közül azokat, amelynek neve tartalmaz számot!

```
$ ls -l /dev | awk '{print $1,$10}' | grep [0-9] |
  awk '{print $1}' | grep b | wc -l
   447
```

Az `awk` programok formája a következő:

```
minta { akció }
minta { akció }
...
```

[18] Az elnevezés az „eredeti” awk szerzőinek vezetéknevéből adódik: Al Aho, Peter Weinberger, Brian Kernighan

Ha a programunk rövid, vagy „egyszer használatos”, a legegyszerűbb, ha a parancssorból futtatjuk: `awk 'program' input1 input2 ...`, ahol a `program` tartalmazza a mintákat és az akciókat. Ha hosszabb programot futtatunk (vagy többször szeretnénk használni) célszerű egy fájlba elmenteni és így futtatni: `awk -f program-fájl input1 input2 ...`

Az `awk` a bemenetét rekordokra bontja, ezeket a *rekordelválasztó* (Record Separator - RS nevű beépített változó) választja el egymástól. Az alapértelmezett RS az újsor karakter (`\n`), így az input egy sora jelent egy rekordot. A rekordelválasztó megváltoztatása - például - így lehetséges: `awk -v RS="." ...` Ekkor a '.' karakter jelzi a rekordok végét. Ha nem a parancssorból szeretnénk megváltoztatni az értéket, akkor a programunkba kell beírni az `RS="."` sort. A rekordok számát a - beépített - NR (Number of Records) változóban tároló tárolja.

A rekordokat mezőkre darabolja az `awk`, a mezőket a *mezőelválasztó* (Field Separator - FS) választja el egymástól. Alapértelmezésben ez egy tabulátor vagy space karakter(ek). Megváltoztatása vagy az RS-nél leírt módon történhet, vagy pedig parancssorból a `-F` kapcsolóval. (Például `awk -F : '{print $1}' /etc/passwd` kiírja a rendszer felhasználóinak nevét.) A feldolgozás alatt álló rekord mezőire $i-vel lehet hivatkozni ($1 \leq i \leq NF$ (Number of Fields)), $0 reprezentálja az egész rekordot.

A reguláris kifejezések az `awk`-nak is szerves részét alkotják. Használatuk a következő: `awk '/REGKIF/ { akció }' input`.

Nem esett még szó a lehetséges akciókról. Nézzünk meg néhány függvényt, a vezérlési szerkezeteket, változók használatát, stb., azaz a tulajdonképpeni AWK programozási nyelvet.

print Mint neve is mutatja nyomtatást végez. A nyomtatandó elemeket vesszővel (,) elválasztva várja, és szóköz karakterekkel elválasztva írja a kimenetére. Megadhatunk neki számot, betűt, szavakat, mezőket, stb.

Például: `awk '{ print "A beolvasott rekord:",$0}' input-fájl`. Érdemes figyelni arra, hogy a kiírandó elemek közé tegyünk vesszőt, mert különben az elemeket összefűzve írja ki (ez főleg akkor lehet bosszantó, ha a kimenet egy pipe (|), és a kimeneten további szűréseket végzünk). A print kimenete az ún. kimeneti rekord, ezeket a rekordokat az ORS (Output Record Separator) változóban megadott karakter választja el egymástól, alapértelmezésben ez az újsor karakter (megváltoztatása a „szokásos” módon). Persze a FS-nek is van „párja”, az OFS (Output Field Separator) nevű változó, amely a kimeneti rekord mezőit választja el egymástól (alapértelmezésben szóköz).

printf Ha jobban bele szeretnénk szólni a kimeneti rekordok formájába, akkor a printf lesz segítségünkre. Használata nagyon hasonló a C printf() függvényéhez.

Formája: `printf form, elem1, ...`

A formázó karakterek ugyanazok, mint C-ben (`%c` egy karakter, `%d` decimális egész, ...). Amire figyelnünk kell C-ről áttérve az az, hogy a `h` (short) és `l` (long) formázókaraktereket ne használjuk, mert hibához vezet!

Mind a `print`, mind a `printf` esetén használhatjuk a shell-ben megszokott `>`, `>>`, `<` és `|` karaktereket. Ez azért jó, mert ha több `print` van a programunkban, akkor a különböző kimeneti rekordokat különböző fájlban tárolhatjuk el.

Nézzünk az eddigiekre egy példát! Két fájlt szeretnénk kapni, az egyik tartalmazza a rendszer felhasználóinak azonosítóját, és user-id-jét, a másikba pedig mentsük el a fel-

használók valódi nevét és group-id-jét (nem túl életszerű példa, de demonstrációnak talán jó).

```
$ awk -F : '{
  > print $1,$3 > "username+uid"
  > print $5,$4 > "realname+gid"
  > }' /etc/passwd
$ cat username+uid | head -3
  root 0
  daemon 1
  bin 2
```

Változók, operátorok, relációk Hosszabb programjainkban szükségünk lesz saját változókra. Az `awk` változóit nem kell külön deklarálni, a változók típustalanok. Értékadásra az `=` szolgál. Használhatjuk a C-ben megszokott `++`, `-`, `*`, `/`, `%`, `&&`, `||`, `+=`, `-=`, `*=`, `/=`, `%=`, `!` stb. operátorokat. Hatványozásra a `^` vagy a `**` szolgál. Az „igaz" és „hamis" logikai értékeket a C-hez hasonlóan használhatjuk (0 vagy üres szó a „hamis", a „nem hamis" pedig igaz). Az relációs operátorok - szokás szerint - C-szerűek: `<`, `>`, `==`, `!=` stb.

Vezérlési szerkezetek Itt is „kísért" a C, nézzük!

- Szelekciós: `if (feltétel) utasítások else utasítások`
- Kezdőfeltételes ismétléses: `while (feltétel) utasítások`
- Végfeltételes ismétléses: `do utasítások while (feltétel)`
- Számlálásos: `for (inicializálás; végfeltétel; növelés)`

Használhatjuk továbbá a `break` és `continue` kifejezéseket ciklusból kiugrásra, illetve az iteráció folytatására.

Tömbök Az `awk`-ban is használhatunk tömböket. Alapvető különbség - például a C-hez képest -, hogy nem kell előre megadnunk a tömbünk méretét, valamint indexeléshez nem csak egymás utáni egész számokat használhatunk, hanem tetszőleges számokat, továbbá szavakat (betűket) is (asszociatív tömbök).

Egy tömb elemének elérésére a `[ ]` szolgálnak (például T nevű tömbünk „izé" indexű elemére `T[izé]`-vel hivatkozhatunk).

A fenti tulajdonság miatt szükségünk lehet arra, hogy leellenőrizzük, hogy egy tömbnek adott indexű eleme létezik-e. Ennek módja: `tömbindex in tömb`. A kifejezés 0 (hamis) értéket ad vissza, ha nincs ilyen indexű elem, igazat egyébként.

A tömbfeltöltésre is az `=` szolgál: `T[izé] = "lalala"`.

Többdimenziós tömbök kezelését is megengedi az `awk`. Egy n-dimenziós tömb indexeléséhez n indexre van szükségünk, így például `T[3,5]` egy $3x5$-ös mátrixot jelent. Itt kell megemlíteni a SUPSEP nevű beépített változót, amely elválasztja egymástól egy többdimenziós tömb indexeit (alapértéke \034). Ez azért fontos, mert az `awk` a fenti `T[3,5]` hivatkozást átalakítja `T[3 SUPSEP 5]`-re, és ezt a kifejezést használja indexelésre.

Függvények Beszélhetünk beépített-, és felhasználó által definiált függvényekről. A „szokásos” matematikai függvények (`sin()`, `cos()`, `sqrt()` ...) az awk-ban is megtalálhatóak, érdekesek lehetnek még a `length([sztring])`, `match(sztring, regexp)`, `sprintf()`, `tolower(string)`, `toupper(string)`, `system(command)`, `systime()`, melyek működésére a nevük utal.

Ugyanakkor mi is definiálhatunk függvényeket, ezzel „teljes értékű” programozási nyelvet kapva. A függvénydefiníció alakja: `function fv_név(paraméterlista) { utasítások }`, és használhatjuk a `return` kifejezést függvényből való visszatérésre (értékkel is).

1.6.4. Egy komplex shellscript példa

Íme egy összetett shellscript példa:[19]

```
#!/bin/bash
x=‘echo "$1" | sed ’s/\/$//’ ‘
echo "$x"
[ "‘basename "$0"‘" = "allarch_ln"  ] && j="‘dirname "$x"‘/binary-$2"
[ "‘basename "$0"‘" = "allarch_mov" ] && j="‘dirname "$x"‘/binary-all"
echo "$j"
p=‘dirname "$0" ‘
echo "$p"
subst="s/^‘echo "$x/" | sed -f "$p/minta" ‘//"
#[ "‘basename "$0"‘" = "allarch_ln" ] && find "$j" -type l -print0 |\
#                                       xargs -0 -n 1 rm -f
for i in ‘find "$1" -name "*.deb" -type f‘
do
  echo "subst = $subst "
  k=‘echo "$i" | sed "$subst" ‘
  l=‘dirname "$k"‘
  m=‘basename "$k"‘
  [ "$l" = "." ] || dp=‘echo "$l" | sed ’s/[^/]\+/../g’ ‘
  if [ "‘basename "$0"‘" = "allarch_ln" ]; then
    echo "dp = $dp"
    echo "ktcsin $j / $l"
    [ "$l" = "." ] || echo "linkel $i ( $dp / ../binary-all / $k ) -> $j / $l "
    [ "$l" = "." ] && echo "linkel $i ( ../binary-all / $k ) -> $j / $l "
    echo " $k == $l / $m "
    mkdir -p "$j/$l"
#   ln -s "$i" "$j/$l"
    [ "$l" = "." ] || ln -fsn $dp/../binary-all/$k $j/$l
    [ "$l" = "." ] && ln -fsn ../binary-all/$k $j/$l
    echo "ok"
  elif [ "‘basename "$0"‘" = "allarch_mov" ]; then
    if dpkg-deb -f "$i" Architecture |grep -q all ; then
      echo "ktcsin $j / $l"
      echo "mozgat $i ( $x / $k ) -> $j / $l "
      echo " $k == $l / $m "
      mkdir -p "$j/$l"
      mv "$i" "$j/$l"
    fi
  fi
done
```

[19] A script megtalálható `allarch_ln` és `allarch_mov` néven is (symlinkek). Ill. a scriptel egy könyvtárban található egy `minta` nevű fájl is, aminek a tartalma a következő: `s/\//\\\//g`

1.7. Bevezetés a Perlbe

A Perl (Practical Extraction and Report Language[20]) egy platformfüggetlen, interpretált, elsősorban szövegfeldolgozásra szánt nyelv. Tervezésekor a hatékonyságot tartották szem előtt a kód szépségével szemben. A Perl-t szokták a „rendszergazdák nyelve”-ként is emlegetni, nagyon hatékonyan használható például log-fájlokfeldolgozására. Megismeréséhez hasznunkra válhat a C-, awk-, shell-szkript ismeret.

1.7.1. Változótípusok

A nyelv ismertetését kezdjük a változótípusokkal! A Perl nem szab határt a változók méretére, egy tömbbe akár egy több megabájtos fájlt is betölthetünk (felső korlátot persze azért a hardver szab). Három változótípus áll rendelkezésünkre: változó, tömb, asszociatív tömb. A változónevek típusát nevüknek első karaktere határozza meg:

- változó: `$valtnev`
- tömb: `@tombnev`, tömbelem: `$tombnev[index]`
- asszociatív tömb: `%atombnev`, tömbelem: `$atombnev['index']`

Mivel a változónevek a típusuk szerint külön szimbólumtáblába kerülnek, így használhatunk akár ugyanolyan nevű „sima” változót, tömböt és asszociatív tömböt is egyszerre.

A Perl rendelkezik jónéhány beépített változóval is, nézzünk ezekből párat!

$1,$2,... Reguláris kifejezésekből kapott betűk.

$& vagy $MATCH A legutolsó mintaillesztésnél az illesztett rész.

$‘ vagy $PREMATCH A legutolsó mintaillesztésnél a $MATCH előtti rész.

$’ vagy $POSTMATCH A legutolsó mintaillesztésnél a $MATCH utáni rész.

$. vagy $NR Az utolsó beolvasott sor sorszáma.

$/ vagy $RS Az input rekordokat elválasztó karakter. Alapértelmezésben ez az újsor karakter.

$\ vagy $ORS Az output rekordokat elválasztó karakter (amit a `print` kiír minden sor végére). Alapértelmezésben ez a változó üres.

$$ vagy $PID A processz azonosítója.

$< vagy $UID A programot indító felhasználó valódi user id-je.

$> vagy $EUID A futás közbeni jogok tulajdonosa.

$0 A programot indító parancs neve.

@ARGV A parancssori argumentumok listája.

%ENV A környezeti változók.

[20] vagy Pathologically Eclectic Rubbish Lister, ahogy alkotója, Larry Wall nevezte

1.7.2. Vezérlési szerkezetek

A Perl utasításait ; karakter választja el egymástól, továbbá az egyszerű utasítások végrehajtásához is köthetünk feltételeket: `print "Hello World!" if $nem_koszontem_meg;` A lehetséges módosítók: if, unless, while, until.

Most nézzük meg a vezérlési szerkezeteket.

- Szelekciós: `if (feltétel) { blokk } [[elsif (feltétel) { blokk} ...] else { blokk }`
- Szelekciós 2: `unless (feltétel) { blokk } [else { blokk }]`
- Számlálásos: `for (kezdőérték;feltétel;iteráció) { blokk }`
- Számlálásos 2: `foreach változó (tömb) { blokk }`
- Kezdőfeltételes: `while (feltétel) { blokk }`
- Végfeltételes: `do { blokk } while (feltétel)`
- Végfeltételes 2: `do { blokk } until (feltétel)`

Használhatjuk a „szokásos” kontroll szavakat is:

- next: átlép a következő iterációra.
- last: az előző iterációs lépés.
- redo: az aktuális iteráció ismétlése.
- return: visszatérés.

1.7.3. Operandusok

Használhatjuk a C-ből és a shell programozásból megismert operátorok (`++`, `--`, `**`, `!`, `<<`, `>>`, stb), de van pár „új” operátor is:

x Ismétlés. Például `"la"x3` eredménye `"lalala"`.

. Konkatenáció. Például `"Hello "."world!"` eredménye `"Hello world!"`.

lt, gt, le, qe, eq, ne, cmp Használhatóak a szövegek összehasonlítására.

.. Tartomány meghatározása.

1.7.4. Beépített függvények

Az standard C függvények mind „megvannak” a Perl-ben is, nyugodtan használhatjuk őket. Ha egyes függvények „gyanúsan” viselkednek, használjuk a POSIX modult: `use POSIX;`

A Perl-hez ezen kívül rengeteg modul létezik (például SQL, hálózati programok, stb.), nézzünk néhány érdekesebb beépített függvényt:

caller Megmondja, hogy mi hívta az aktuális programrészt.

chop Levágja az utolsó karaktert a sztring végéről. Hasznos lehet például sor vége (\n) karakterek „eltüntetésére”.

defined KIFEJEZÉS Megmondja, hogy a megadott kifejezésnek van-e értéke.

delete KIFEJEZÉS Egy elem törlése egy asszociatív tömbből.

each ASSZOC_TÖMB Egy asszociatív tömb elemeit iterálja. Ha ki szeretnénk írni az összes környezeti változót: `while (($nev,$ertek) = each %ENV) { print "$nev = $ertek"; }`

exists KIFEJEZÉS Megmondja, hogy létezik-e az asszociatív tömb megadott eleme.

my KIFEJEZÉS A megadott változók csak az adott blokkban lesznek láthatóak.

open FÁJLLEÍRÓ, KIFEJEZÉS A kifejezésben leírt fájl megnyitása a megadott fájlleíróba.

- Megnyitás olvasásra: `open(R,"<inputfile");`
- Megnyitás írásra: `open(R,">outfile");`
- Megnyitás írásra és olvasásra: `open(R,"file");`
- Olvasás pipe-ból: `open(R,"finger|");`
- Írás pipe-ba: `open(R,"|head");`

close FÁJLLEÍRÓ Bezárja a megadott fájl(leíró)t.

print FÁJLLEÍRÓ LISTA Lista tartalmát beleírja a FÁJLLEÍRÓ által meghatározott fájlba. Ha nincs fájlleíró megadva, akkor a stdout-ra ír.

sort LISTA Lexikografikus rendezést hajt végre a listán.

undef KIFEJEZÉS Változó megszüntetése.

1.7.5. Mintaillesztések

Mintaillesztésre két operátor szolgál: =~ az illeszkedés, !~ pedig a nem illeszkedés. Nézzünk pár példát:

- `$sor =~ /izé/;` hasonló, mint a grep.
- `$sor =~ s/izé/mizé/;` hasonló, mint a sed-nél a csere.

Persze reguláris kifejezéseket[21] is használhatunk.

[21] Az előző fejezet szerintieket, tehát: ^, ., $, |, (), [], *, +, ?, {}

1.8. A GNU M4 makróprocesszor, és az M4 makrónyelv

1.8.1. Bevezető

A GNU „M4”[22] egy implementációja a hagyományos Unix makrófeldolgozónak. A makróprocesszor dolga hogy a bemenetét a kimenetére másolja át, és terjessze ki a benne található makrókat. (Lásd pl. `#include<inklúdfájl.h>` a C előfeldolgozóban.) Ezek a makrók lehetnek beépítettek vagy a „felhasználó” által definiáltak, és akárhány argumentumuk lehet.

Az egyszerű makrókifejtés mellett az M4 képes beilleszteni egy megnevezett fájlt, parancsot lefuttatni, egész aritmetikai műveleteket végezni, a szöveget számos módon manipulálni, rekurzióra, stb. Akár egy előtét is lehet egy programfordító elé. (Végül-is ennek speciális esetének tekinthető, amikor az autoconf elkészíti nekünk a configure scriptet.)

1.8.2. A használat alapjai

Természetesen az M4 használható lebutítva, az eredeti SYSTEM V beli makróprocesszorral kompatibilisen is, ha a `-G` vagy `--traditional` kapcsolót megadjuk neki.

A C-hez hasonlóan itt is lehet a `-Ikönyvtár` vagy `--include=könyvtár` paraméterrel további include könyvtárakat adni.

Elérhető az is, hogy az M4 eredeti beépített makrói megjelölsére kerüljenek oly-módon, hogy mindegyik neve elé odakerüljön az „`m4_`” prefixum, ha a `-P` vagy `--prefix-builtins` paramétert is megadjuk. Például az eredeti `define` helyett így `m4_define`-t kell írni.

A GCC ill. CPP-hez hasonlóan itt is használhatóak a `-DNÉV` , `-DNÉV=ÉRTÉK` , `-UNÉV` és `-UNÉV=ÉRTÉK` paraméterek.

1.8.3. Lexikai és szintaktikus konvenciók

Makró nevek Szokás szerint, egy név állhat betűkből, számjegyekből, és az _ karakterből, ahol a bevezető karakter nem számjegy. Ha egy névhez tartozik makródefiníció, akkor az a makró kifejtésre fog kerülni.
Példák helyes nevekre: `ize`, `_tmp`, `nev01`.

Idézett szövegek Az idézet szöveg egy karaktersorozat, amit az ‘ kezdő és az ’ záró idézőjelek vesznek körül, ahol-is a kezdő és a záró idézőjelek a szövegben egyensúlyban vannak. A „sztring token” értéke az lesz, mintha az idézőjelekből egy szintet elhagynánk. Így pl. a ‘’ egy üres sztring lesz, és az ‘‘`idézet`’’ értéke pedig ‘`idézet`’ lesz.[23]

Egyéb tokenek Minden karakter ami nem egy név része, vagy egy idézett sztringé, önmagában is egy token.

[22] Néhány ember úgy gondolja hogy az M4 megfelel a szenvedélyeinek. Ők először csak egyszerű problémák megoldására használják az m4-et, később egyre és egyre nagyobb kihívást látnak benne, és megtanulják, hogyan írjanak komplex M4 makróhalmazokat. Egyszer-csak függővé válnak tőle, és a felhasználó szokásává válik mesterkélt M4 alkalmazások írása a problémák megoldása, még akkor is, ha az M4 szkriptek hibakeresésnek több időt szentel mint a valódi munkának. Én figyelmeztettelek! Az M4 káros lehet a megszállott programozók egészségére!

[23] Az idézőjeleket jelentő karakter bármikor megváltoztatható. Lásd a `changequote` makrót.

Megjegyzések Az M4 alapesetben a `#` és újsor karakterek közé tett szöveget megjegyzésként tekinti. Minden karaktert ami megjegyzést határoló jelek közt van, figyelmen kívül hagy, de az egész megjegyzés (,beleértve a határolójeleit is) keresztül megy az M4-en és átkerül annak a kimenetére, vagyis az M4 nem dobja el a megjegyzéseket.
A megjegyzéseket nem lehet beágyazni, olyan-értelemben, hogy az első újsor karakter egy `#` után lezárja a megjegyzést. A „megjegyzés effektus" elkerülhető ha a bevezető megjegyzés karaktert idézőjelbe tesszük.[24]

1.8.4. Makrók

Makró meghívása

Egy makró meghívása egyszerűen annyi, hogy leírjuk a makró nevét `név` feltéve, hogy nincsenek argumentumai. Egyébként a `név(arg1, arg2, ..., argN)` forma használatos.

A makróknak tetszőleges számú argumentumuk lehet. Minden egyes argumentum egy sztring, de különböző makrók, különböző módon értelmezhetik az argumentumokat.
A nyitó zárójelnek közvetlenül a makró neve után kell álljon, és nem lehet közte szóköz. Ha ez nem így van, akkor a makró teljesen argumentumok nélkül hívódik meg.

Ha egy olyan makrót hívunk meg, amelynek nincsenek argumentumai, akkor a zárójeleket kötelező kihagyni, ugyanis a `makrónév()` hívás nem úgy értelmeződik, mintha argumentum nélkül hívtuk volna meg a makrót, hanem úgy mintha egy argumentummal hívtuk volna meg, és az az egy argumentum pedig egy üres sztring.

Makró meghívásának elkerülése

Az M4 makrónyelv nagy fejlődése az őt megelőző makróprocesszorokhoz képest, hogy képes felismerni a makróhívásokat anélkül, hogy azt valami speciális bevezetőkarakterrel jeleznénk számára. Bár ez általában hasznos dolog, időnként nemszándékos makróhívásokat eredményezhet. Így a GNU M4 számos mechanizmust vagy technikát kínál ara, hogy elkerüljük, hogy neveket makróhívásokként ismerjen fel.

Először-is tudni kell, hogy számos beépített makrót nem lehet értelmesen meghívni argumentumok nélkül. Ezen makrók bármelyikére igaz, hogyha nem követi egy nyitó zárójel, ott nem váltódik ki („triggerelődik") makróhívás. Ez a legtöbb ilyen szokványos esetet megoldja, mint például az `include` és `eval` esetén.

Használható még a bevezetőben említett `-P` kapcsoló, ami előírja hogy a beépített makrók meghívásakor a `m4_` prefixet használjuk a makrónevekben. Ellenben ez az opció nincs hatással a felhasználó által definiált makrók neveire.

A legegyszerűbb módja, hogy megelőzzük egy létező makró kifejtését, ha idézőjelbe tesszük. Habár a beidézést alkalmazhatjuk az egész makrónévre, lehetőség van arra is, hogy egy üres sztringet tegyünk idézőjelbe, de ez csak a makrón belül működik. Tehát az alábbi esetekben nem kerüli el a makrókifejtést:

```
‘’eltérít
eltérít‘’
```

[24] A megjegyzéseket határoló jelek is megváltoztathatók tetszőleges sztringre, bármikor, ha a beépített `changecom` makrót használjuk.

Míg ezen példákban a makrókifejtés elkerülésre kerül:

```
‘eltérít’
‘e’ltérít
el‘tér’ít
elté‘’rít
```

A makrókifejtések értékei mindig újraolvasódnak. Az alábbi példa a `de` sztringet adja, pontosan mintha az M4-nek a `subsr(abcde, 3, 2)`-t adtuk volna bemenetül:

```
define(‘x’, ‘substr(ab’)
define(‘y’, ‘cde, 3, 2)’)
x‘’y
```

Az olyan idézőjelbe nem tett sztringek, amelyek valamelyik oldalán álnak egy idézőjelbe tett sztringnek ki vannak téve annak a lehetőségnek, hogy makrónevekként legyenek felismerve. Az alábbi példában az üressztring lehetővé teszi a dnl makrónak, hogy akként ismerődjön fel ami (vagyis makróként):

```
define(‘macro’, ‘di$1’)
macro(v)‘’dnl
```

Ha nem lennének ott az idézőjelek, akkor megengedné a `divdnl` sztringet egy sorvégjellel lezárva azt.

Az idézés megelőzheti azt is, hogy egy makrónevet ismerjen fel, ott ahol egybevon egy kifejtett makrót az őt körülvevő makrókkal. Vagyis az alábbi példa:

```
define(‘macro’, ‘di$1’)
macro(v)‘ert’
```

a bemenet a `divert` sztringet állítja elő. Ha az idézőjelet eltávolítanánk, akkor a `divert` makró meghívásra kerülne.

Makrók argumentumai

Amikor egy nevet lát az M4 és ahhoz létezik makródefiníció, azt makróként kifejti. Ha a nevet egy nyitó zárójel követi, akkor előbb az argumentumokat begyűjti, mielőtt a makrót kifejtené. Ha túl kevés argumentum van megadva, akkor a hiányzó argumentumokat üressztringnek tekinti. Ha több argumentum van, akkor pedig a további argumentumokat figyelmen kívül hagyja.[25]

Alapesetben az M4 figyelmeztet ha a beépített makrókat nem megfelelő számú argumentummal hívjuk meg, de ezt el lehet folytatni a `-Q` parancssori opcióval. A felhasználó által definiált makrók esetén nincs az argumentumok számára vonatkozó ellenőrzés.

A makrókat normális esetben az argumentumgyűjtés során is kifejtésre kerülnek, legyen bennük vessző, idézőjel, vagy zárójel, így a kifejtett szöveg szintén argumentum célját látja el. Így ha feltesszük, hogy az `izé` makrónév a `, b, c` szövegre fejtődik ki, akkor a `mizé(a izé, d)`

[25] Vajon gondoltak arra a fejlesztők, hogy vmi egyszerű módon ezt a tulajdonságot fel lehetne használni *megjegyzés* készítésére?

egy olyan makróhívás, ahol négy argumentum van, ezek rendre: `a`, `b`, `c` és `d`. Hogy megértsük, hogy az első argumentum itt miért tartalmaz (könnyű)szóközt, emlékezzünk rá, hogy a bevezető idézőjellel körbe nem vett (könnyű)szóköz soha nem része az argumentumnak, ellenben a záró szóköz viszont igen.

Az argumentumok idézése

Minden argumentumból a bevezető (könnyű)szóközök eltávolításra kerülnek. Minden egyes argumentumban, az összes idézőjelbe nem tett zárójelnek kell legyen párja. Például ha az `izé` egy makró, akkor az `izé(() (‘(’) ‘(’)` egy makróhívás, aminek egy argumentuma van, aminek az értéke `() (() (`.

Általános gyakorlat, hogy a makrók minden argumentumát idézőjelek közé tegyük, kivéve ha pont azt szeretnénk, hogy az argumentumok kifejtésre kerüljenek. Ennek megfelelően a fenti példát ha áttekinthetően szeretnénk kivitelezni a zárójelekkel, akkor a nagykönyvbeli alak így néz ki: `izé(‘() (() (’)`

Makró kifejtésének menete

Amikor argumentumok, ha vannak, mind begyűjtésre kerültek, a makró kifejtésre kerül, és a kifejtett szöveg visszakerül a bemenetre (nem bezárva idézőjelek közé), és újraolvasódik. A kifejtett szöveg egy makróhívás során további makróhívásokat eredményezhet.

Vegyünk egy egyszerű példát: Az `izé` a `mizé` szövegre fejtődik ki, és a `mizé` pedig a `Hello world` szövegre. Így ha a bemeneten találunk egy `izé`-t, akkor az először kifejtődik `mizé`-re, majd mikor újraolvasódik, akkor kifejtődik `Hello world`-re.

1.8.5. Definíciók

Új makró definiálása

A makró definiálás a beépített `define` makróval megy. Melynek módja a következő:

```
define(NÉV [, KIFEJTÉS])
```

Ez a mód definiálja a `NÉV` nevű makrót, amit az M4 `KIFEJTÉS`-re fog kifejteni. Ha a `KIFEJTÉS` nincs megadva, akkor az üresnek számít. A define makró üres szövegre fog kifejtődni. A makródefiníció helyén egy új sor meg fog jelenni, ha a makródefiníció után jön egy új sor, amit az M4 konzekvensen bemásol a kimenetre, de ez elkerülhető a dnl makróval.

A makrók argumentumai

A makróknak lehetnek argumentumai. Az N-edik argumentumot a `$n`-el jelöljük a kifejtendő szövegben. Vagyis például a `$1` az első argumentumra fog kifejtődni.

A `define(‘exch’, ‘$2, $1’)` makródefiníció egy olyan makrót definiál, ami a neki átadott két argumentumot felcseréli. Így például a `define(exch(‘‘kifejtendő szöveg’’, ‘‘macro’’))` makródefiníció egy `macro` nevű makrót definiál, amely kifejtve `kifejtendő szöveg`-re fejtődik ki.

A GNU M4 megengedi, hogy a `$` után több számjegy is álljon, lehetővé téve, hogy bármilyen számú argumentuma legyen. Ez nem teljesül az M4 egyéb UNIX implementációira, amelyek csak egy számjegyet ismernek fel a `$` után.

Létezik még egy speciális eset, a `$0` argumentum, ami az éppen kifejtett makró nevére fejtődik ki. Ha azt szeretnénk hogy idézőjelbe tett szöveg megjelenjen a beágyazott szöveg részeként, akkor ne feledjük, hogy idézőjeleket beágyazhatunk az „idézett" szövegbe. Így a `define(‘izé’, ‘Ez az ‘izé’ makró.’)` makródefiníció az `izé`-t `Ez az izé makró` szövegre[26] fejti ki.

Pszeudó argumentumok

Van egy speciális jelzés a makrónak átadott argumentumok számának jelzésére. A makró hívásakor átadott argumentumok számát a `$#`-al jelölhetjük a kifejtett szövegben. Így a makró megjelenítheti, hogy hány argumentuma van.
Példa: `define(‘nargs’, ‘$#’)` definiálja az `nargs` nevű makrót, amely arra fejtődik ki, hogy hány argumentumot adtunk neki át. Például a `nargs` `0`-ra fejtődik ki, a `nargs()` 1-re fejtődik ki (, ugye mindenki emlékszik, hogy a `()`-el meghívott makrónak egy üressztring átadódik paraméterül), és az `nargs(arg1, arg2, arg3)` makróhívás `3`-ra fejtődik ki.

A `$*` szimbólum a kifejtődés során, a makrónak átadott összes paramétert visszaadja a kifejtés során, vesszővel elválasztva azokat.[27] Így ha definiálunk egy makrót a következőképp `define(‘visszhang’, ‘$*’)`, akkor a `visszhang(arg1, arg2, arg3 , arg4)` makróhívás a `arg1,arg2,arg3 ,arg4`-re fejtődik ki.[28]

A `$@` szimbólum hasonló az előzőhöz, avval a különbséggel, hogy minden egyes argumentum idézve jelenik meg. Így ha a makródefiníciónk a következő: `define(‘visszhang’, ‘$@’)`, akkor a `visszhang(arg1, arg2, arg3 , arg4)` az `arg1,arg2,arg3 ,arg4` szövegre fejtődik ki. Nyilván jogos a kérdés, hogy akkor hova lettek az idézőjelek. Noss természetesen, amikor az M4 kifejtette a makrót, visszatolta a bemenetére, és az idézőjeleket megette. Hogy lássuk a különbséget hajtsuk végre az alábbi kísérletet:

```
$ cat <<’END’ >proba.m4
define(‘vissz1’, ‘$*’)dnl
define(‘vissz2’, ‘$@’)dnl
define(‘ize’, ‘Ez az ‘ize’ makro’)dnl
vissz1(ize)
vissz2(ize)
END
$ m4 proba.m4
Ez az Ez az ize makro makro
Ez az ize makro
```

[26] Ne feledjük, hogy a makrónevek csak számokat, az aláhúzásjelet, és az angol abc betűit tartalmazhatják, így a fenti példákban érvénytelenek a makrónevek a magyar ékezetes betűk miatt. Azok csak az egyszerűbb szemléltetés miatt tartalmaznak ékezetet!

[27] Hasonlít a shell `$0`jához.

[28] Ne feledjük, hogy a vessző után álló bevezető szóközök nem számítanak be az argumentumba, de az argumentum végén a következő vesszőig/lezáró zárójelig tartó szóközök viszont igen.

Ha a $ után nem áll semmi olyan, amit az M4 értelmezni tudna, akkor a $ egyszerűen átmásolódik a kimenetre. Ha szeretnénk egy olyan makrót, amelyben valami ilyesmi szöveg van mint az $12, akkor a $ után írjunk egy üres idézőjelet, így: $‘’12.

Makró törlése

Egy definiált makró egyszerűen eltávolítható az `undefine` makróhívással. Viszont figyeljünk arra, hogy a makró nevét szükségszerűen zárójelbe kell tegyük, különben az M4 kifejti a makrót, még mielőtt definiálatlanná tenné!

Ha az `undefine`-nak olyan makrónevet adunk meg, amire vonatkozóan nincs makródefiníció, akkor az nem számít hibának. Ilyenkor az `undefine` nem csinál semmit.[29]

Makró átnevezése

Lehetőség van már definiált makró átnevezésére is. Hogy ezt megtehessük, szükségünk lesz a `defn` nevű beépített makróra, ami az argumentumul kapott makrónév idézőjeles definícióját szolgáltatja. Ha az argumentuma nem egy definiált makró, akkor a kifejtése üres. Ha az argumentumul megadott makrónév egy felhasználó által definiált makró, akkor egyszerűen az idézőjeles definícióját adja vissza. Egyébként, ha egy beépített makró, akkor a kifejtés egy speciális tokent ad eredményül, ami a beépített makró belső definíciójára mutat. Ennek a tokennek csak akkor van jelentősége, ha az egy `define` makró második argumentuma (vagy egy `pushdef` makróé), egyéb környezetekben figyelmen kívül lesz hagyva.

Íme egy példa, hogyan nevezzük át az `undefine` makrót `zap`-ra:
`define(‘zap’, defn(‘undefine’))`. Ilymódon a `defn` felhasználható makródefiníciók másolására, még akkor is, ha beépített makróról van szó. Még akkor is, ha az eredeti makródefiníciót eltávolítottuk, a másolat makrónév még mindig használható a definíció elérésére.[30]

Makrók ideiglenes felüldefiniálása

Lehetőség van makrók definíciójának ideglenes felüldefiniálásra, és visszatérni az eredeti makródefinícióra egy későbbi időpontban. Ezek a műveletek a beépített
`pushdef(NÉV [, KIFEJTÉS])` és a `popdef(NÉV)` makrókkal tehetők meg, amelyek a `define` és `undefine` makrók analogonjai. Ebben a modellben a makrók úgy működnek mint egy verem szerű szerkezet. Egy makrót átmenetileg újradefiniálhatunk a `pushdef` makróval, ami lecseréli az épp aktuális definícióját a `NÉV` nevű makrónak, miközben a korábbi definíciót is elmenti, még mielőtt az újjal felülírná. Ha nincs korábbi definíciója a makrónak, akkor a `pushdef` pontosan úgy működik, mint a `define`.

Ha egy makrónak már számos definíciója van (amelyből természetesen csak egy érhető el egyszerre), akkor a legfelső definíciót eltávolíthatjuk a `popdef` makróhívással. Ha nincs további definíciója a makrónak, akkor a `popdef` pontosan úgy viselkedik mint az `undefine`.

Ha egy makrónak már van néhány definíciója, akkor a `define` pontosan csak a legfelsőt cseréli ki. Ellenben ha egy makrónak a definícióját eltávolítjuk az `undefine`-al, akkor az az összes definíciót eltávolítja, nem csak a legfelsőt.

[29] Az `undefine` makrót csak paraméterrel ismeri fel a makróprocesszor.

[30] A `defn` makrót csak paraméterekkel hajlandó felismerni a makróprocesszor.

Makró közvetett meghívása

Bármilyen makró meghívható közvetett módon is az `indir` makróval, melynek szintaktikája a következő:

```
indir(NÉV, ...)
```

Ami azt eredményezi, hogy a `NÉV` nevű makró meghívódik, és a további argumentumok átadódnak neki. Ez használható arra, hogy meghívjunk olyan makrókat, melyeknek „érvénytelen”[31] neve van.[32]

Ennek az lehet az értelme, hogy nagyobb makrócsomagoknak lehessenek belső privát makrói, amelyeket véletlenül nem lehet meghívni, csak az `indir` makróhíváson keresztül.

Beépített makrók közvetett hívása

Beépített makrókat közvetetten is meg lehet hívni a `builtin` makróval, amelynek a szintaktikája a következő:

```
builtin(NÉV, ...)
```

Ami azt eredményezi, hogy a `NÉV` nevű makró meghívódik, és a további argumentumok, pedig átadódnak neki. Ez használható akkor is, ha a `NÉV` nevű makrónak adtunk egy másik definíciót, ami elrejti az eredetit.

A `builtin` makrót csak paraméterekkel ismeri fel a makróprocesszor.

1.8.6. Feltételek, hurkok, vezérlési szerkezetek

Az egyszerű makrók, amelyeket egyszerű szövegre kifejtünk, esetleg argumentumai vannak, nem feltétlenül nyújtanak nekünk elég funkcionalitást. Ezért hasznos, ha futásidőben eldőlő döntéseket is meg lehet hozni. Például szükségünk van feltételekre, valamilyen ciklus szerű vezérlési szerkezetre. Így valamit egy adott számszor vagy amíg egy feltétel igaz, végre tudunk hajtani.

ifdef

Az M4-ben két különböző típusú feltételes elágazás van. Az egyik ilyen az `ifdef`, melynek szintaktikája a következő:

```
ifdef(NÉV, STRING1, opt STRING-2)
```

Ez lehetővé teszi, hogy ellenőrizzük, hogy egy makró definiálva van-e már, vagy nincs. Ha a `NÉV` egy már definiált makró, akkor az `ifdef` a `STRING1`-re fejtődik ki, egyébként a `STRING2`-re. Ha a `STRING2` hiányzik, akkor üressztringnek veszi a makróprocesszor (hivatkozva a normális szabályokra).

Íme egy egyszerű példa:

```
ifdef(‘izé’, ‘az ‘izé’ definiált’, ‘az ‘izé’ nem definiált’)
```

Az `ifdef` makrót csak paraméterekkel ismeri fel a makróprocesszor.

[31] Az eredeti szöveg szerint „illegális”

[32] A `define` megengedi az ilyenek létrehozását is.

ifelse

A másik feltételes elágazási lehetőség az `ifelse`, ami sokkal többet tud. Lehet használni arra is, hogy csináljunk egy hosszú megjegyzést, vagy egy if-else szerkezetre, vagy egy többágú elágazási szerkezetre, függően a megadott argumentumok számától:

```
ifelse(MEGJEGYZÉS)
ifelse(STRING1, STRING2, EGYENLŐ, opt NEMEGYENLŐ)
ifelse(STRING1, STRING2, EGYENLŐ, ...)
```

Ha csak egy argumentummal hívjuk meg, akkor az `ifelse` egyszerűen eldobja azt, és nem képződik kimenet. Ez az általános M4 stílus, hogy bevezessünk egy blokk-megjegyzést, a `dnl` alternatívájaként. Ezt a speciális felhasználást felismeri a GNU M4, így amikor csak egy argumentummal hívjuk meg nem váltódik ki figyelmeztető üzenet.

Ha három, vagy négy paraméterrel hívjuk meg, akkor az `ifelse` az `EGYENLŐ` szöveg kerül kifejtésre, amennyiben a `STRING1` és a `STRING2` szövegek megegyeznek (karakterről karakterre), egyébként pedig a `NEMEGYENLŐ` szöveg.

Egy egyszerű példa: `ifelse(izé, mizé, ‘igaz’, ‘hamis’)`

Egyébként az `ifelse` meghívható, több mint négy argumentummal is. Ha több mint négy argumentuma van, akkor az `ifelse` úgy működik, mint egy hagyományos `case` vagy `switch` szerkezet az egyéb hagyományos programozási nyelvekben. A működés módja ilyen esetben a következő: Ha a `STRING1` és a `STRING2` egyenlők, akkor az `EGYENLŐ` szöveg helyettesítődik, egyébként az eljárás az első három argumentum eldobásával megismétlődik. Egy példa: `ifelse(izé, mizé, ‘harmadik’, répa, retek, ‘hatodik’, ‘hetedik’)`

Az `ifelse` makrót csak paraméterekkel ismeri fel a makróprocesszor.

ciklusok

Nincs közvetlen támogatás a ciklusok létrehozására az M4ben, de a makrók lehetnek rekurzívak. Nincs határ a rekurziók szintjére vonatkozólag, egyéb határokat csak a hardver és az operációs rendszer szabhat.

A fentiek alapján a ciklusokat le lehet programozni csak a feltételes elágazás és a rekurzió segítségével. Ehhez segítségünkre lesz még a `shift` makró is, mely többek közt képes arra, hogy a makrónak átadott argumentumain iterációt hajtsunk végre. A `shift` tetszőleges számú argumentumot kaphat, és kifejtve az összes argumentumát adja vissza vesszővel elválasztva, minden egyes argumentumot idézve, és kihagyva a legelső argumentumot.

Így pl. a `shift(izé)` egy üressztringre fejtődik ki, az `(izé, mizé, bizé)` pedig `mizé,bizé` lesz.

Íme egy példa arra, hogy hogyan csináljunk meg egy olyan makrót, amely kifejtve, az argumentumul kapott paramétereket adja, fordított sorrendben:

```
define(‘forditott’, ‘ifelse($#, 0, , $#, 1, ‘‘$1’’, ‘forditott(shift($@)), ‘$1’’)’)
```

Vagyis itt a `forditott(a, b, c, d)` makróhívás a `d, c, b, a`-ra fejtődik ki.

Egy másik példa azt szemlélteti, hogyan készíthetünk, egy egyszerű számlálásos for ciklust az M4 eszközeivel:

```
define(‘forloop’,
       ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)dnl
define(‘_forloop’, ‘$4‘’ifelse($1, ‘$3’, ,
              ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)dnl
```

És néhány példa az imént definiált `forloop` makró használatára:

```
forloop(‘i’, 1, 6, ‘i ’)
forloop(‘i’, 1, 6, ‘forloop(‘j’, 1, 7, ‘(i, j) ’)
’)
```

Mely kifejtve így néz ki:

```
1 2 3 4 5 6
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7)
(2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7)
(3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7)
(4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7)
(5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7)
(6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7)
```

1.8.7. Bemenet vezérlése

dnl

A beépített `dnl` makró beolvas minden karaktert az utána következő első újsorkarakterig, beleértve az adott újsor karaktert is, és ezeket kitörli/eldobja. Általában arra szokás használni, hogy egy sorba leírjunk egy makródefiníciót, és a makró definiálásakor ne kerüljön fölöslegesen újsor a kimenetre.

Általában a `dnl` makrót egy újsor karakter követi közvetlenül. De ha véletlenül, egy nyitózárójelet találna utána a makróprocesszor, akkor figyelmeztető üzenetet küld, de begyűjti a dnl argumentumait (és persze kifejti őket), és végrehajtódnak az evvel járó mellékhatások. Ellenben minden kifejtett karakter, egész az újsorkarakterig ekkor is el lesz dobva.

changequote

A `changequote` makró segítségével megváltoztathatjuk az idézőjelet jelző sztringet. Így például, ha a `changequote([, ])` utasítást kiadjuk, akkor az `izé` makrónkat a következőképp tudjuk definiálni: `define([izé], [[izé] makró.])`

Ha nem csak egy egyszerű karaktert adunk meg, akkor a kezdő és a záró zárójel akármilyen hosszú is lehet.

Ha az idézőjeleket üressztringre változtatjuk, akkor evvel átmenetileg effektíve minden idézőjelezési mechanizmust kiiktatunk, elzárva minden lehetőséget arra, hogy szöveget idézőjelbe tegyünk.

Az idézőjel nem kezdődhet sem betűvel, sem aláhúzásjellel, mert az ütközne a bemenetre kerülő nevekkel. Ha mégis így tennénk, akkor avval letiltjuk az idézési mechanizmust a makróprocesszorban.

changecom

A `changecom` makró segítségével beállíthatjuk, hogy mi legyen a megjegyzések kezdését és végét jelző sztring. Ha valamelyik argumentumát elhagynánk a makrónak, akkor az eredeti alapértelmezett megjegyzést határoló jelek lépnek életbe.

Ha pl. C stílusú megjegyzéseket szeretnénk, akkor használhatjuk a `changecom(‘/*’, ‘*/’)` parancsot.[33]

1.8.8. Fájl beillesztése

Fájlok beillesztésére két makró is szolgál az M4ben:

```
include(FÁJLNÉV)
sinclude(FÁJLNÉV)
```

Az include makró kifejtve, gyakorlatilag az argumentumul megadott fájl tartalma. Az hibához vezet és hibaüzenetet kapunk, ha az argumentumul adott fájl nem létezik. Ilyenkor a `sinclude` makrót használjuk, mert akkor ha nem létezik a fájl, egyszerűen továbbmegy a makróprocesszor.

A `include`-al behozott fájlokban levő makrók is kiértékelődnek, stb.

A beépített `include` és `sinclude` makrókat a makróprocesszor csak akkor ismeri fel, ha adunk neki argumentumot.

Az M4 lehetővé teszi, hogy más (nem csak az aktuális) könyvtárból származó fájlokat is beszúrhassunk. Ha egy fájlt nem talál az aktuális munkakönyvtárban, és a megadott fájl neve nem abszolút fájlnév, akkor a fájlt a megadott keresési útban végignézi. Először a `-I` kapcsolóval megadott könyvtárakban, majd pedig a `M4PATH` környezeti változó által jelzett könyvtárakban.

1.8.9. A kimenet eltérítése, visszatérítése

Az eltérítés egy módja annak, hogy átmenetileg elmentsük a kimenetet. Az M4 kimenetét bármikor el lehet téríteni egy átmeneti fájlba, és újrabeszúrni a kimeneti folyamba (visszatéríteni) egy későbbi időpontban.

A számozott eltérítések 0-tól felfele számozódnak, és a 0 jelenti a normális kimeneti folyamot (*stream*). A párhuzamos eltérítések számát csak a leírásukhoz szükséges memória korlátozza.Egyébként létezik egy korlát ami az összesen felhasználható memóriát együttvéve korlátozza (, és ez aktuálisan 512K). Amikor a maximumot elérné, akkor egy átmeneti fájlt megnyit a makróprocesszor, és odateszi a legnagyobb eltérítés tartalmát, evvel memóriát felszabadítva a többi eltérítés számára. Így tehát elméletileg lehetséges, hogy az eltérítések számát a maximálisan elérhető fájlleírók (*file descriptor*ok) száma korlátozza.

A kimenet eltérítése

A `divert` makró szintaxisa:

```
divert(opt SZÁM)
```

[33] Arra figyeljünk, hogy a megjegyzés egyszerűen átmásolódik a kimenetre, mintha idézőjelbe tett szöveg lenne. Mindössze annyi a jelentősége, hogy a megjegyzésen belül nem hajtódik végre makrókiértékelés.

Ahol a `SZÁM` a használandó eltérítés száma. Ha a számot nem adjuk meg, akkor alapértelmezett értékének a 0-t kell feltételezni. A `divert` makró kifejtése üres sztring lesz.

Amikor az M4 minden bemenetét feldolgozta, akkor minden létező eltérítést visszatérít, számsorrendben. Íme egy példa:

```
divert(1)
Ez a szöveg el van térítve
divert
Ez a szöveg nincs eltérítve
```

Amelyet ha az m4-el értelmezetünk, akkor a következő kimenetet kapjuk:

```
Ez a szöveg nincs eltérítve

Ez a szöveg el van térítve
```

Ha többször ugyanavval az argumentummal meghívjuk a `divert`-et, akkor az nem írja felül a benne levő korábbi szöveget, hanem hozzáfűzi azt.

Ha a kimenetet egy nemlétező eltérítésbe küldjük, akkor az egyszerűen eldobásra kerül. Ezt használhatjuk arra, hogy elkerüljük a nemkívánatos kimenetet. Egy általános példa arra, hogy a makródefiníciók utáni újsor karaktereket elnyeljük, ha a következő módon cselekszünk:[34]

```
divert(-1)
define(‘izé’, ‘‘izé’ makró.’)
define(‘mizé’, ‘‘mizé’ makró.’)
divert
```

A kimenet visszatérítése

Az eltérített szöveg bármikor visszatéríthető explicit módon is az `undivert` beépített makró használatával, melynek a szintaxisa a következő:

```
undivert(opt SZÁM, ...)
```

Ez az argumentumul adott eltérítéseket visszatéríti, abban a sorrendben, ahogy argumentumul megadtuk. Ha nem adunk meg argumentumot, akkor az összes eltérítés visszatérítődik, számsorrendben.

Az `undivert` makró kifejtve szintén egy üressztring lesz.

Amikor egy eltérített szöveget visszatérítünk, azt nem olvassa újra az M4[35], de ahelyett, hogy egyszerűen csak átmásolná azt a kimenetére, az is lehetséges, hogy egy eltérítést egy másik eltérítésbe térítsünk vissza:

```
$ cat <<’END’ |m4
> divert(1)
> Először ezt térítjük el
> divert‘’dnl
```

[34] Természetesen a makródefiníció után írt dnl is jó, de amikor már sok definíciós sor van, akkor áttekinthetetlenné teszi a *forrás*szöveget.

[35] értsd: nem fogja újra átnézni makrókifejtések után

```
> Ez nincs eltérítve
> divert(-1)undivert(1)divert(1)dnl
> Ez is 1-esbe van eltérítve, de vissza fog jönni
> END
Ez nincs eltérítve
Ez is 1-esbe van eltérítve, de vissza fog jönni
```

A GNU M4 lehetővé teszi, hogy nevén nevezhető fájlokat térítsünk vissza. Ha egy nemnumerikus argumentumot adunk meg, akkor az argumentumul megadott fájlnév tartalma lesz bemásolva, értelmezés nélkül az aktuális kimenetre, ilymódon kiegészítve a beépített `include` makrót. Hogy lássuk a pontos különbséget íme egy példa:

```
$ cat <<'END' >foo
> bar
> END
$ cat <<'END' |m4
> define(`bar', `BAR')
> undivert(`foo')
> include(`foo')
> END

bar

BAR
```

Eltérítési számok

Az aktuális eltérítés számát a beépített `divnum` makróval lehet megtudni. A dolog szemléltetésére egy példa:

```
$ cat <<'END' |m4
> Initial divnum
> divert(1)
> Első eltérítés: divnum
> divert(2)
> Második eltérítés: divnum
> END
Initial 0

Első eltérítés: 1

Második eltérítés: 2
```

Eltérített szöveg eldobása

Ha egy eltérített szöveget szeretnénk kitörölni, akkor azt tudjuk tenni, amit abban a példában láttunk, ahol az eltérített szöveget egy másik eltérítésbe térítettünk vissza. Ha egy olyan

eltérítésbe térítünk vissza, ami eldobásra kerül, akkor el tudunk dobatni eltérítéseket. Pl. ha az input feldolgozás végén szeretnénk eldobni minden eltérítés tartalmát, ahelyett, hogy implicit módon visszatérítődjenek, akkor a következő írjuk az inputfájl végére:

```
divert(-1)
undivert
```

1.8.10. A legfontosabb makrók rövid leírása

len(STRING) szöveg hossza. Pl `len(‘abcdef’)` => `6`

index(STRING, SUBSTRING) Szövegrész keresése a szövegben. Pl. `index(‘ecc, pecc, kimehetsz, holnapután bejöhetsz’, ‘pecc’)` => `5`. Az első karakter indexe `0`, ha nem találta a szövegben, akkor `-1` lesz a kifejtés eredménye.

regexp(STRING, REGEXP, opcionális HELYETTESÍTŐ) Reguláris kifejezés keresése a szövegben. Ha helyettesítő szöveg nincs megadva, akkor a REGEXP első előfordulása a STRING-ben, ill. ha semelyik részére nem illeszkedik, akkor `-1`. Ha van HELYETTESÍTŐ megadva, akkor végrehajtódik a szövegen a helyettesítés.

substr(STRING, FROM, opt LENGTH) A szöveg egy részének kivágása.

translit(STRING, KARAKTEREK, HELYETTESÍTŐ) A shell tr parancsához hasonló karakterfordítás.

pathsubst(STRING, REGEXP, HELYETTESÍTŐ) reguláris kifejezéssel való helyettesítés.

format(FORMÁTUMSZTRING, ...) printf szerű formázások.

incr(SZÁM) szám növelése.

decr(SZÁM) szám csökkentése.

ecal(KIFEJEZÉS, opt RADIX, opt SZÉLESSÉG) egész kifejezés kiértékelése.

syscmd(SHELLPARANCS) Shell parancs végrehajtása.

esyscmd(SHELLPARANCS) Shell parancs végrehajtása, a parancs kimenete lesz a kifejtés értéke.

sysval A legutoljára végrehajtott shellparancs exitkódja.

1.8.11. M4-et használó rendszerek

A teljesség igénye nélkül néhány olyan program ami az M4-et használja:

- autoconf
- automake

- sendmail
- pcb
- fvwm
- debian linux külső kernel modulok

2. fejezet

II. félév

2.1. A GNU autoconfba bevezető

Az autoconf egy eszköz arra, hogy automatikusan shellszkripteket állítsunk elő, amely konfigurálja egy program forráskódját, hogy az számos UNIX-szerű rendszerhez tudjon illeszkedni. Az autoconf által előállított szkriptek függetlenek az autoconf-tól amikor futnak, így az már nem kell hogy a programforrást felhasználó oldalán telepítve legyen.

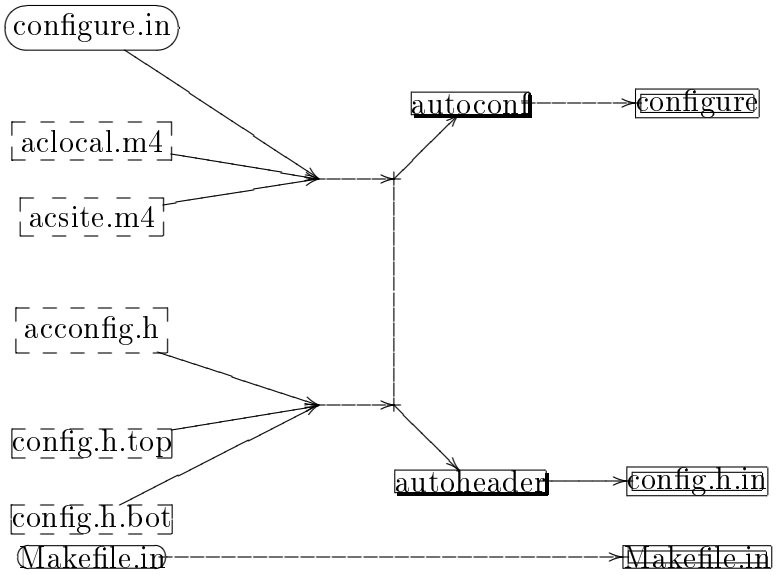
Az autoconf által előállított konfigurációs szkriptek nem igényelnek kézi beavatkozást amikor futnak, normális esetben még csak egy argumentumot se kell neki adni, ami az adott rendszer típusára vonatkozik. Ehelyett az teszteli minden egyes szükséglet jelenlétét, amelyeket a szoftvercsomag igényelhet. (Minden egyes ellenőrzés előtt kiír egy egysoros üzenetet amely utal arra, hogy épp mit ellenőriz, így a felhasználó nem unatkozik míg a szkript végez.) Az eredmény jól leírja az akár hibrid, vagy személyreszabott UNIX-variánsunk minden tulajdonságát, és nem kell fájlokat karbantartanunk, ami minden egyes UNIX-variáns tulajdonságait leírja.

Minden egyes autoconf-ot használó szoftvercsomag, létrehoz egy konfigurációs szkriptet egy mintafájlból (*template*) ami felsorolja, hogy milyen rendszertulajdonságokat igényel, vagy tud használni. Mivel a shell kód feladata, hogy felismerjen és reagáljon egy rendszertulajdonságra, az autoconf lehetővé teszi, hogy ezt a kódot több szoftvercsomag közt megosszuk, amely tudja használni vagy igényli ezt a rendszertulajdonságot. Ha később kiderülne, hogy ez a shellkód némi igazítást igényelne valamilyen okból kifolyólag, és azt egy helyen megváltoztatjuk, akkor az összes konfigurációs szkript(,mármint minden olyan szoftvercsomag konfigurációs szkriptje, amely az adott shellkódot használta az autoconfból) újragenerálható automatikusan, hogy kihasználhassa a javított kód előnyeit.

2.1. ábra. Az autoscan használata

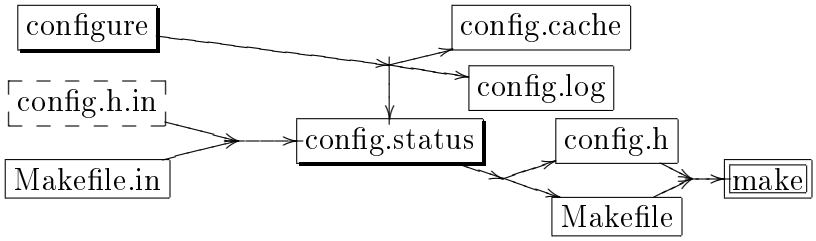
Számos olyan feladat van, amely szoftvercsomagok portolhatóságához kapcsolódik, amelyeket jelenleg az autoconf nem tud. Ezek közt van, hogy automatikusan létrehozzon `Makefile`okat az összes konvencionális targettel, vagy hogy helyettesítést biztosítson a nem implementált sztandard könyvtári funkciók helyett, vagy hiányzó headerfájlokat olyan rendszeren amelyen az nincs meg.

Az autoconf megkövetel néhány megszorítást a makrónevekkel kapcsolatban amelyek C programok `#ifdef`-jénél használhatók.



2.2. ábra. Az autoconf használata

Az autoconf megköveteli a GNU M4-et is a scriptek generálásához, ugyanis használ néhány olyan GNU kiterjesztést is, amelyet más UNIX M4 verziók nem támogatnak. Továbbá túlcsordítja néhány M4-verzió belső határait is, beleértve a GNU M4 1.0-t is. Hogy minden rendben legyen legalább az 1.1 verziójú GNU M4-et kell használni, de az 1.3 verzió sokkal gyorsabb mint az 1.1 vagy 1.2 verzió.[1] Az autoconf által készített konfigurációs szkriptet konvecionálisan



2.3. ábra. A fenti fájlok felhasználása egy programcsomag fordításakor

`configure`-nak hívjuk. Amikor lefuttatjuk a `configure` számos fájlt létrehoz, belehelyettesítve konfigurációs paramétereket a megfelelő értékekkel. A configure által létrehozott konfigurációs fájlok:

Makefile Egy vagy több `Makefile`-t létrehoz a `configure` a csomag minden egyes alkönyvtárába.

config.h Opcionálisan egy C fejlécfájlt létrehoz a `configure` amely `#define` direktívákat tartalmaz.

config.status Létrehoz egy shell-szkriptet, amelynek feladata, hogy a fenti fájlokat újralétrehozza.

[1] Egy nem túl újnak mondható verzió, ami a Debian GNU/Linux Woody (3.0) verziójában van, és már az is 1.4-es, tehát nem egy nehezen teljesíthető feltétel.

config.cache Létrehoz egy shell-szkriptet, amelynek feladata, hogy a tesztek eredményeit elmentse.

config.log Létrehoz egy logfájlt, amely a futtatás alatt a fordítóprogram által adott üzeneteket tartlmazza, hogy adott esetben könnyebb legyen a hibát megkeresni, ha a `configure` hibát vét.

Hogy létrehozzuk a `configure` szkriptet, meg kell írnunk egy `configure.in` nevű bemeneti fájlt, és ezen lefuttatnunk az `autoconf` parancsot. Ha saját *feature*-teszteket írunk, az `autoconf`éit kiegészítendő, akkor azokat az `aclocal.m4` és `acsite.m4` fájlokba írhatjuk. Ha szeretnénk egy C fejlécfájlt, amely `#define` direktívákat tartalmaz, akkor írhatunk egy `acconfig.h` nevű fájlt is, és terjeszthetjük a szoftvercsomagban az autoconf által generált `config.h.in` fájlt is.

A `configure.in` fájl áttekintése

A `configure.in` fájl az autoconf csomag M4 makróra vonatkozó hívásokat tartalmaz. Az autoconf már jónáhány makrót tartalmaz arra, hogy a rendszert és annak adottságait teszteljük vele. Amihez nem tartalmaz támogatást az autoconf, ott használhatjuk a mintáit (*template*jeit), hogy saját ellenőrzőrutinokat írjunk.

A `configure.in` fájl megírását nem a nulláról kell kezdeni, lásd az idevonatkozó `autoscan` használatát szemléltető ábrát. Az `autoscan` parancsot lefuttatva egy jó kiindulópontunk van a `configure.in` megírásához.

Annak a sorrendje, hogy a `configure.in` milyen sorrendben hívja meg a makrókat nincs különösebb jelentősége, néhány kivételtől eltekintve:

- Mindig az `AC_INIT` makróval kell kezdenünk
- Mindig az `AC_OUTPUT` makróval kell befejeznünk
- Továbbá néhány makró helyes működése azon múlik, hogy előtte bizonyos makrókat már használnunk kell, mert felhasználnak olyan értékeket, amelyek más makrók állítanak be előtte. Az ilyen esetek szerepelnek a makrók leírásaiban.

Ezen indokok miatt egy `configure.in` váza valahogy így néz ki:

```
‘AC_INIT(FÁJL)’
programok meglétének ellenőrzése
könyvtárak (library) ellenőrzése
fejlécállományok (header) ellenőrzése
típusdefiníciók ellenőrzése
struktúrák ellenőrzése
programfordító (compiler) tulajdonságainak ellenőrzése
könyvtárfunkciók ellenőrzése
rendszerszolgáltatások ellenőrzése
‘AC_OUTPUT([FÁJL...])’
```

Néhány általános jótanács az `configure.in` megírására

Lehetőleg minden makróhívást tegyünk külön sorba, mert a legtöbb makró nem tesz be extra újsor karaktereket, így a makróhívás utáni újsorkaraktertől erősen függenek. Ez a megközelítés a generált `configure` scriptet olvashatóbbá teszi, hogy nem teszünk bele egy csomó üres sort.

Amikor olyan makrókat hívunk meg, amelyek argumentumokat vesznek át, az M4 szabályai szerint nem lehet szóköz a makrónév és a nyitó zárójel közt. Az argumentumok több mint egysorosak is lehetnek, ha bezárjuk őket az M4 zárójeleivel, amely ebben a környezetben a `[` és a `]` karakter lesz. Ha egy hosszú sort írunk, amely például fájlneveket tartalmaz, akkor nyugodtan tehetünk egy visszapert a sor végére, és logikailag folytathatjuk a következő sorban. Ez nem az `autoconf` vagy az M4 adta tulajdonság, hanem a shell szkriptek adta lehetőség.

Néhány makró két esetet kezel: Mi történjen, ha egy adott feltétel igaz, illetve ha hamis. Bizonyos helyeken csak akkor szeretnénk tenni valamit, ha a feltétel igaz, illetve fordítva: bizonyos helyeken csak akkor szeretnénk tenni valamit, ha a feltétel hamis. Hogy elkerüljük az igaz esetet, adjunk meg üres értéket az `TEENDŐ-HA-IGAZ` makróargumentumnak. Ha pedig a hamis ágat szeretnénk kihagyni, akkor pedig hagyjuk ki a `TEENDŐ-HA-HAMIS` argumenumot, és a hozzá tartozó, azt megelőző vesszőt.

Hogy áttekinthetőbbé tegyük a kódot, tegyünk a `configure.in` fájlba megjegyzéseket, amelyet az M4 beépített `dnl` makrójával kezdünk, amely lenyeli az utánaálló szöveget egész az azt követő újsor karakterig. Így ezek a megjegyzések már nem fognak megjelenni a `configure` szkriptben.

2.2. A GNU autoconf

2.2.1. A `configure` hogyan találja meg a bemenetét

Minden `configure` scriptnek meg kell hívnia az `AC_INIT`-et mielőtt bármi mást csinálna. A másik olyan makró amelynek használata kötelező az a `AC_OUTPUT`, amelyet a `configure` scriptek végén kell meghívni. Az `AC_INIT` használatának szintaxisa:

AC_INIT (Egyedi-fájl-a-forrás-könyvtárban)

Feldolgoz minden paranccsori argumentumot, és megtalálja a forráskód könyvtárát. Az *Egyedi-fájl-a-forrás-könyvtárban* tetszőleges fájl lehet, ami a csomag forráskönyvtárában van. A `configure` ellenőrzi eme fájl létezését, hogy a könyvtár amelyben van, valóban tartalmazza a forráskódot.

Az olyan csomagoknak, amelyek kézi konfigurációt igényelnek, vagy az `install` programot használják, valahogyan közölniük kell a `configure`al, hogy hol találhatja meg az egyéb shell-szkripteket. Ez az `AC_CONFIG_AUX_DIR` makró segítségével történik, habár az alapértelmezett helyek ahol a fájlokat a szkript keresi megfelelő a legtöbb esetben.

AC_CONFIG_AUX_DIR(DIR)

A *DIR* könyvtárból használja az `install-sh`, `config.sub`, `config.guess` szkripteket. Ezek segédfájlok a konfigurációhoz. A *DIR* lehet abszolút vagy relatív is a *SRCDIR*-re nézve. Az alapértelmezett az *SRCDIR* vagy az *SRCDIR/..* vagy az *SRCDIR/../..*, attól függően, hogy melyik tartalmazza legelőször az `install-sh`-t. A többi fájlt nem ellenőrzi, így az `AC_PROG_INSTALL` nem igényli automatikusan a többi segédfájl közzétevését. Ez a rész az `install.sh`-t is ellenőrzi, de ez már idejétmúlt, lévén a `make` programoknak van egy szabálya arra vonatkozóan, hogy csináljon `install`-t belőle, ha nincs `Makefile`.

2.2.2. Kimeneti fájlok előállítása

Minden autoconf által generált `configure` szkriptnek az `AC_OUTPUT` makró meghívásával kell végződnie. Ez az a makró ami `Makefile`-okat és más opcionális fájlokat generáltat a konfigurációtól függően.

AC_OUTPUT([FÁJL... [, EXTRA-PARANCSOK [, INIT-PARANCSOK]]])

Létrehozza a kimeneti fájlokat. Csak egyszer hívjuk meg ezt a makrót, és ez szerepeljen a `configure.in` fájlunk végén. A *FÁJL...* argumentum egy szóközökkel elválasztott listája a kimeneti fájloknak, ami akár üres is lehet. Ez a makró létrehoz minden olyan fájlt, amit itt felsoroltunk. Olymódon, hogy ha pl. az eredetileg felsorolt név *FÁJL* volt, akkor azt a *FÁJL.in* nevű fájlból hozza létre belehelyettesítve a kimeneti változók értékeit. A makró létre is hozza a *FÁJL*hoz tartozó könyvtárat, ha az előtte nem létezett. (De az adott könyvtár szüleit már nem!) Általában `Makefile`okat generálunk így, de más fájlokat is létrehozhatunk ilymódon.

Ha korábban használtuk az `AC_CONFIG_HEADER`, `AC_LINK_FILES` vagy `AC_CONFIG_SUBDIRS` makróhívásokat, akkor az ezek argumentumaiból következő fájlok is itt kerülnek létrehozásra.

Egy `AC_OUTPUT` makróhívás tipikusan valahogy így szokott kinézni:
`AC_OUTPUT(Makefile src/Makefile man/Makefile X/Imakefile)`

A bemeneti fájlok neveit felül lehet bírálni, olymódon, hogy a fájl nevéhez kettőspontal elválasztva hozzáírjuk, hogy mi legyen a bemeneti fájl vagy fájlok (ekkor is kettőspont a határolójel) neve. Példák:

```
AC_OUTPUT(Makefile:templates/top.mk lib/Makefile:templates/lib.mk)
AC_OUTPUT(Makefile:templates/var.mk:Makefile.in:templates/rules.mk)
```

Ha adunk meg *EXTRA-PARANCSOK*at, akkor azok beillesztődnek a `config.status`-ba, hogy leffuttatódjanak az egyéb feldolgozások után. Ha *INIT-PARANCSOK*at is megadunk, akkor azok a *EXTRA-PARANCSOK* elé illesztődnek, és a shell változók értékei, a parancs-, és a zárójel- helyettesítések végrehajtódnak rajtuk a `configure`-ban. Így például felhasználható arra, hogy a `configure`-ból változók értékeit átadjuk az *EXTRA-PARANCSOK*nak. Ha az `AC_OUTPUT_COMMANDS` makrót meghívjuk, akkor a akkor a parancsok amelyeket annak adunk pont azelőtt futnak le, hogy az ezen makrónak adott parancsokra kerülne a sor.

AC_OUTPUT_COMMANDS (EXTRA-PARANCSOK [, INIT-PARANCSOK])

Olyan további shell parancsok adhatók meg itt, amelyek a `config.status` szkript végén fognak lefutni, illetve olyan parancsok amelyek a `configure`-ból inicializálnak változóértéket. Ezt a makrót több alkalommal is meg lehet hívni. Ime egy a valóságtól elrugaszkodott példa:

```
ize=27
AC_OUTPUT_COMMANDS([echo ez egy extra $ize, sít.], ize=$ize)
AC_OUTPUT_COMMANDS([echo ez egy másik, extra, apróság], [echo init apróság])
```

Ha a `make`t futtatjuk alkönyvtárakon, akkor azt úgy tegyük, hogy a `MAKE` változó tartalmát beállítjuk `make`-re. A legtöbb `make` verzió beállítja a `MAKE` értékét a `make` program nevére plussz a neki átadott opciókra. De számos verzió pedig nem veszi bele a parancssorban átadott változók értékeit, így azok nem kerülnek automatikusan átadásra. Néhány régi `make` verzió ilyen. Az alábbi makró lehetővé teszi, hogy akár ezekkel a régi verziókkal is használható legyen művünk.

AC_PROG_MAKE_SET

Ha a `make` előre beállítja a `MAKE` változót, akkor a `SET_MAKE` kimeneti változót üresre állítja. Egyébként a `SET_MAKE` változó a `MAKE=make` sztringet fogja tartalmazni. Ez a makró az `AC_SUBST` makrót használja a `SET_MAKE` beállítására.

Hogy eme makró lehetőségét kihasználjuk, tegyünk egy ilyen sort a `Makefile.in`be, ahol az lefuttatja a `MAKE`et a többi könyvtárra:

```
@SET_MAKE@
```

2.2.3. `Makefile`ebeli helyettesítések

Minden egyes könyvtárnak, amelyben van valami lefordítani való, vagy telepíteni való, tartalmaznia kell egy `Makefile.in`t, amelyből a `configure` `Makefile`-t csinál abba a könyvtárba. Hogy létrejöjjön a `Makefile` a `configure` egy egyszerű változóhelyettesítést végrehajt a fájlokon, olymódon, hogy minden egyes `@VÁLTOZÓ@` előfordulást a `Makefile.in`ben lecserél arra az értékre amelyet a `configure` megállapított. Ezen változókat, melyek helyettesítésre kerülnek a kimeneti fájlokban, kimeneti változóknak hívjuk. Általában ezek olyan shell változók, amelyet a `configure` szkript állít be. Ahhoz, hogy a `configure` behelyettesítse ezeket a változókat a kimeneti fájlokban az `AC_SUBST` makrót kell meghívni a helyettesítendő változó nevét argumentumul adva.

Egy szoftvercsomagot amely egy `configure` szkriptet használ egy `Makefile.in`el kell terjeszteni és nem `Makefile`-al, mert a `configure` fogja majd a megfelelő értékeket behelyettesíteni, így a felhasználónak csak annyi a dolga, hogy a helyi rendszernek megfelelően konfigurálja a csomagot mielőtt lefordítja, és telepíti.

2.2.4. Kimeneti változók

Bizonyos kimeneti változók értéke mindig előre be van állítva az autoconf makróinak köszönhetően. Bizonyos autoconf makrók pedig további kimeneti változókat állítanak be amelyek az adott makrók leírásában szerepelnek. Ime azon kimeneti változók amelyek értéke előre be van állítva, magyarázattal együtt:

bindir Annak a könyvtárnak a neve, ahova a felhasználó által futtatható állományokat telepíteni kell.

configure_input Egy megjegyzés, amely jelzi, hogy a fájl automatikusan lett generálva a `configure` szkript által, és megadja a bemeneti fájl nevét. A `AC_OUTPUT` hozzávesz egy megjegyzés sort minden egyes általa létrehozott `Makefile` elejéhez, amely ennek a változónak a tartalma lesz. Más fájlok esetén külön hivatkozni kell erre a változóra, hogy a megjegyzés megjelenjen a bemeneti fájl elején. Például, ha van egy (bemeneti) shell szkriptünk, akkor annak az eleje valahogy így kell kinézzen a cél érdekében:

```
#!/bin/sh
# @configure_input@
```

Ezen sor megléte emlékezteti a fájlt szerkesztő embert, hogy azt a `configure` szkripttel fel kell dolgoztatnia.

datadir Annak a könyvtárnak a neve, ahova a csak olvasható, architektúra-független adatokat kell elhelyezni.

exec-prefix Az architektúra-függő fájlok telepítési prefixuma.

includedir Annak a könyvtárnak a neve, ahova a C fejlécállományokat kell telepíteni.

infodir Annak a könyvtárnak a neve, ahova az info formátumú dokumentációt kell telepíteni. (Lásd Makefile konvenciók rész)

libdir Annak a könyvtárnak a neve, ahova a tárgykódú könyvtárakat kell telepíteni.

libexecdir Annak a könyvtárnak a neve, ahova a más programok által futtatott futtatható programokat kell telepíteni.

localstatedir Annak a könyvtárnak a neve, ahova a gépre nézve egyedi, módosítható adatokat kell telepíteni.

mandir A felsőszintű könyvtárnév ahova a man formátumú dokumentációt kell telepíteni.[2]

oldincludedir Annak a könyvtárnak a neve, ahova a C fejlécfájlokat kell telepíteni a nem-gcc fordítók számára.

prefix Az architektúra-független fájlok telepítési prefixuma.

sbindir Annak a könyvtárnak a neve, ahova az adminisztrátor által használt futtatható programokat kell telepíteni.

sharedstatedir Annak a könyvtárnak a neve, ahova a módosítható architektúra-független adatokat kell telepíteni.

srcdir Annak a könyvtárnak a neve, amely az adott `Makefile`hoz tartalmazza a forráskódot.

sysconfdir Annak a könyvtárnak a neve, ahova a csak olvasható gépre nézve egyedi adatokat kell tenni.

top_srcdir A csomag forráskódjának legfelső szintű könyvtárának a neve. Ha a legfelső szintű könyvtárban van, akkor ugyanaz lesz mint az `srcdir`

CFLAGS A debuggolási és optimalizációs opciók, amelyeket a C fordítónak át kell adni. Ha ez nincs beállítva a `configure` futtatásánál, akkor az alapértelmezett értéke az `AC_PROG_CC` makró meghívásánál állítódik be (, vagy nem ha nem hívjuk azt meg). A `configure` használja ezt az értéket, amikor programokat fordít, hogy ellenőrizze a C (fordító) lehetőségeit.

CPPFLAGS A fejléc állományok keresési könyvtára és egyéb különféle opciók a C preprocesszor és fordító számára. Ha nincs beállítva környezeti változóként, amikor a `configure` fut, akkor az alapértelmezett értéke üres lesz. A `configure` használja ezt az értéket, amikor programokat fordít vagy preprocesszál, hogy ellenőrizze a C lehetőségeit.

CXXFLAGS Ugyanaz lényegében mint a `CFLAGS`, csak ez a C++ fordítóra vonatkozik. Itt az alapértelmezett érték az `AC_PROG_CXX` makróhívásnál állítódik be. A `configure` használja ezt ez értéket a C++ fordító tulajdonságainak tesztelésére.

FFLAGS Mint a fenti `CFLAGS` csak a Fortran 77 fordítóra vonatkozik, és az alapértelmezett érték a `AC_PROG_F77` makró meghívásánál állítódik be. A `configure` használja ezt az értéket a Fortran 77 tulajdonságainak tesztelésére.

[2]Ezen belül a dokumentáció szekciója szerinti alkönyvtárba kell a kézikünyv-oldalt betenni.

DEFS `-D` opciók, amelyeket a C fordítónak át kell adni. Ha az `AC_CONFIG_HEADER` makrót meghívjuk, akkor a configure inkább a `-DHAVE_CONFIG_H`ra seréli a `@DEFS@` sztringet. Ez a változó nincs definiálva, amikor a `configure` a saját tesztjeit futtatja, csak amikor a kimeneti fájljait hozza létre.

LDFLAGS A *strip*elésre és egyéb vegyes a szerkesztőnek (*linker*) átadandó opciókat tartalmazza. Ha nincs környezeti változóban beállítva, amikor a `configure` fut, akkor az értéke üres lesz. A `configure` használja ennek az értékét, amikor programokat szerkeszt össze a C tulajdonságainak tesztelésekor.

LIBS A szerkesztőnek átadandó `-l` és `-L` opciók.

2.2.5. Konfigurációs fejlécállományok

Amikor egy szoftvercsomag több mint egy néhány C preprocesszor szimbólumot tesztel, akkor a parancssor, amelyben `-D` opciókat adunk át meglehetősen hosszúra nyúlhat. Ez kétféle problémát okoz: Egyrészt, vizuálisan nem lesz áttekinthető a `make` kimenete, ha hibát keresünk benne. És a komolyabbik hiba, pedig az, hogy a paranccsor hosszabbra nyúlhat, mint az operációs rendszer által engedélyezett maximális hossz. Hogy ezt a nehézséget leküzdjük, alternatívaként szóba jöhet, hogy a `configure` script létrehozzon egy C fejlécfájlt, amely ezen `#define` direktívákat fogják tartalmazni. Ha ilyen jellegű kimenetet szeretnénk, akkor azt az `AC_CONFIG_HEADER` makró meghívásával jelezhetjük, amelyet közvetlenül az `AC_INIT` után kell használnunk.

A szoftvercsomagnak be kell illesztenie egy `#include` direktívával ezt a fejlécfájlt, mielőtt bármilyen egyéb fejlécfájllal foglalkozna, hogy a deklarációk inkonzisztenciáját megelőzzük. (Például, ha megpróbálná a `const` szót újradefiniálni.)

A fejlécállomány beillesztésére a `#include <config.h>`-t javasolt használni a `#include "config.h"` helyett, és még így a fordítónak (csak az előfeldolgozó résznek) át kell adni a `-I.` ill. `-I..` opciókat, attól függően, hogy melyik könyvtár tartalmazza a `config.h`-t. Ilymódon ha a forráskönyvtár konfigurálta magát, más build-könyvtárak szintén konfiguráltak lehetnek, anélkül, hogy külön meg kellene keresniük a `config.h`-t a forráskönyvtárból.[3]

AC_CONFIG_HEADER (LÉTREHOZANDÓ-FEJLÉC)

Beállítja az `AC_OUTPUT` makrót, hogy amikor a fájlokat létrehozza, akkor a szóközökkel elválasztva felsorolt *LÉTREHOZANDÓ-FEJLÉC* fájlokat is hozza létre, amelyek tartalmazzák a `#define` állításokat, és a generált fájlokban a `@DEFS@`-t a `-DHAVE_CONFIG_H`ra cserélje a `DEFS` értéke helyett. A szokásos név, ami a *LÉTREHOZANDÓ-FEJLÉC* szokott lenni, a `config.h`

Ha a *LÉTERHOZANDÓ-FEJLÉC* már létezik és tartalma azonos avval, amit az `AC_OUTPUT` beletenne, akkor nem foglalkozik vele. Így elkerülhetővé válik az, hogyha a konfigurációban bekövetkezik néhány apró változás, (ami nem változtatná meg a `config.h`-t) szükségtelenül újrafordítani tárgykódokat, amelyek függenek ettől a fejlécfájltól.

[3]Példa: tfh. van egy x szoftverem, amely részként használja az y-t, ami szintén autoconf-os. Ha x `configure`-ja tud mindent, amit y-é, és a `config.h`-t megfelelően elkészíti, akkor nem kell az y részben külön `./configure`-ni.

Általában a bemeneti fájl neve *LÉTREHOZANDÓ-FEJLÉC.in*, habár ez felülbírálható hogyha kettőspontal elválasztva felsoroljuk a bemeneti fájlok nevét. Példák:

```
AC_CONFIG_HEADER(defines.h:defines.hin)
AC_CONFIG_HEADER(defines.h:defs.pre:defines.h.in:defs.post)
```

2.2.6. Konfigurációs fejléc minták

Ha a programnak tartalmaznia kell egy minta fejlécfájlt, amely úgy néz ki, mint a végső fejlécfájl, beleértve a kommenteket, és a `#define`-ok alapértelmezett értékeit. Például, ha a `configure.in`-ben ezek a makróhívások vannak:

```
AC_CONFIG_HEADER(conf.h)
AC_CHECK_HEADERS(unistd.h)
```

Akkor a `config.h.in`-be valami ilyesminek kellene lennie:

```
/* Értéke legyen 1 ha van unistd.h. */
#define HAVE_UNISTD_H 0
```

Egyéb megoldásként szóba jöhet, hogyha a kód `#ifdef` direktívákkal dönt `#if` helyett, hogy az alapértelmezett értéket `#undef`-eljük ahelyett, hogy a változónak egy értéket definiálnánk. Tehát, egy olyan rendszeren ahol van `unistd.h`, akkor a `configure` meg fogja változtatni a második sort `#define HAVE_UNISTD_H 1`-re. Az egyéb rendszereken pedig megjegyzésbe kerül a sor:

```
/* Definiálódik ha van unistd.h */
#undef HAVE_UNISTD_H
```

2.2.7. Az `autoheader` használata a konfigurációs fejléc minták létrehozására

Az autoheader program létre tud hozni egy ilyen mintafájlt amely C `#define` állításokat tartalmaz, a `configure`nak. Ha a `configure.in` meghívja az `AC_CONFIG_HEADER(FÁJL)` makrót, akkor az `autoheader` létrehozza a `FÁJL.in` nevű fájlt, ha pedig több argumentum is meg volt adva, akkor az elsőt használja. Egyéb esetben pedig az autoheader `config.h.in`-t hoz létre.

Ha az `autoheader`-nek adunk egy argumentumot, akkor azt a fájlt fogja nézni a `configure.in` helyett, és a fejlécfájlt a sztandard kimenetre fogja kiírni a `config.h.in` helyett. Ha az `autoheader`-nek argumentumul egy `-` -t adunk meg, akkor a sztandrd bemenetét fogja olvasni a `configure.in` helyett, és a fejlécfájlt pedig a sztandard kimenetére fogja írni.

Az `autoheader` elolvassa a `configure.in`-t (, vagy amit adunk neki), és az alapján kitalálja, hogy milyen C előfeldolgozó szimbólumokat definiálhat. Az `acconfig.h`ban található megjegyzéseket, `#define` és `#undef` állításokat átmásolja amely az `autoconf` telepítésével jön. Ha az aktuális könyvtárban is van `acconfig.h`, akkor azt is használja. Ha az `AC_DEFINE`-al további szimbólumokat is létrehozunk, akkor kötelezően létre kell hoznunk ezt a fájlt, az ehhez tartozó bejegyzésekkel. Azokhoz a szimbólumokhoz, amelyeket az `AC_CHECK_HEADERS`,

AC_CHECK_FUNCS, AC_CHECK_SIZEOF, AC_CHECK_LIB makróhívások definiálnak, az autoheader generál megjegyzéseket és #undef állításokat saját maga, ahelyett, hogy ezeket egy fájlból másolná, lévén a lehetséges szimbólumok jól körülírhatók.

A fájl, amit az autoheader létrehoz jórészt #define és #undef állításokat tartalmaznak, és a hozzájuk tartozó megjegyzéseket. Ha a ./acconfig.h tartalmazza a @TOP@ szöveget, akkor az autoheader a @TOP@ szöveget tartalmazó sor előtti sorokat átmásolja a generált fájl elejére. Hasonló módon, ha az ./acconfig.h tartalmazza a @BOTTOM@ szöveget, akkor az autoheader átmásolja a @BOTTOM@ szöveget tartalmazó sor utáni sorokat a generált fájl végére. Egyik, vagy akár mindkét szöveg elhagyható.[4]

Egy alternatív módszer hogy hasonlót érjünk el, hogy ha csinálunk FILE.top nevű fájlt (vagyis általában ennek a neve config.h.top lesz) és/vagy FILE.bot nevű fájlt az aktuális könyvtárba. Ha ezek léternek akkor az autoheader átmásolja őket a kimenetének az elejére, illetve a végére.[5]

2.2.8. Az alapértelmezett prefix

A configure beállít egy prefixet, hogy hova települjenek a fájlok. Ennek az alapértelmezése a /usr/local könyvtárra mutat. A configure scriptet használó személy ezt beállíthatja más helyre is, ha a --prefix illetve --exec-prefix opciókat használja. Ezt a beállítást két módon is lehet módosítani, mint láttuk az egyik lehetőség, a configure script futtatásakor. A másik lehetőség, hogy a configure script alapértelmezését változtassuk meg, már a létrehozásakor. (Vagyis amikor kigeneráljuk a configure.in fájlból.) Ez az alábbi makróhívással tehető meg:

AC_PREFIX_DEFAULT(PREFIX)

Beállítja az alapértelmezett telepítési prefixet *PREFIX*re a /usr/local helyett.

Hasznos lehet az is, ha a configure kitalálja a telepítési prefixet egy hozzákapcsolódó, már telepített program nevéből. Ha így szeretnénk cselekedni, akkor erre az AC_PREFIX_PROGRAM makrót használhatjuk az alábbi módon:

AC_PREFIX_PROGRAM(PROGRAM)

Ha a felhasználó nem adott meg telepítési prefixet (, amit ettől függetlenül bármikor felülbírálhat a --prefix opcióval a configure futtatásakor), akkor megpróbálja kitalálni az értékét az alapján, hogy megnézi a PATH-ban hogy a *PROGRAM* hol található. Ha a *PROGRAM* megtalálható, akkor a prefixet a *PROGRAM*-ot tartalmazó könyvtár szülőkönyvtárára állítja, egyébként nem változtatja meg a Makefile.in-ben a prefix értékét.

Például, ha a *PROGRAM* a gcc és a PATH alapján megtalálja a configure script a /usr/local/gnu/bin/gcc nevű fájlt, akkor beállítja a prefixet a /usr/local/gnu könyvtárra.

Az egyszerűbb érthetőség kedvéért íme, egy szemléltető példa:

[4] Természetesen ilyenkor a hozzá tartozó rész nem lesz átmásolva.

[5] Ezek használata ellenjavalt, mert az M$-DOS rendszereken nem lehet két pont a fájlok nevében, valamint két további fájl, hogy telezsúfolja a könyvtárat.

```
$$ cat <<'END' >Makefile.in
> # Ez egy Makefile lesz, és a köv sorban, látható lesz az előző sorbeli megjegyzés
> # @configure_input@
> bindir=@bindir@
> datadir=@datadir@
> exec_prefix=@exec_prefix@
> includedir=@includedir@
> infodir=@infodir@
> libdir=@libdir@
> libexecdir=@libexecdir@
> localstatedir=@localstatedir@
> mandir=@mandir@
> oldincludedir=@oldincludedir@
> prefix=@prefix@
> sbindir=@sbindir@
> sharedstatedir=@sharedstatedir@
> srcdir=@srcdir@
> sysconfdir=@sysconfdir@
> top_srcdir=@top_srcdir@
> CFLAGS=@CFLAGS@
> CPPFLAGS=@CPPFLAGS@
>
> .PHONY: test
> test:
>         @echo "bindir=$(bindir)"
>         @echo "datadir=$(datadir)"
>         @echo "exec_prefix=$(exec_prefix)"
>         @echo "includedir=$(includedir)"
>         @echo "infodir=$(infodir)"
>         @echo "libdir=$(libdir)"
>         @echo "libexecdir=$(libexecdir)"
>         @echo "localstatedir=$(localstatedir)"
>         @echo "mandir=$(mandir)"
>         @echo "oldincludedir=$(oldincludedir)"
>         @echo "prefix=$(prefix)"
>         @echo "sbindir=$(sbindir)"
>         @echo "sharedstatedir=$(sharedstatedir)"
>         @echo "srcdir=$(srcdir)"
>         @echo "sysconfdir=$(sysconfdir)"
>         @echo "top_srcdir=$(top_srcdir)"
>         @echo "CFLAGS=$(CFLAGS)"
>         @echo "CPPFLAGS=$(CPPFLAGS)"
> END
$ cat <<'END' >configure.in
> AC_INIT(Makefile.in)
```

```
> AC_OUTPUT(Makefile)
> END
$ autoconf
$ ./configure
creating cache ./config.cache
updating cache ./config.cache
creating ./config.status
creating Makefile
$ make
bindir=/usr/local/bin
datadir=/usr/local/share
exec_prefix=/usr/local
includedir=/usr/local/include
infodir=/usr/local/info
libdir=/usr/local/lib
libexecdir=/usr/local/libexec
localstatedir=/usr/local/var
mandir=/usr/local/man
oldincludedir=/usr/include
prefix=/usr/local
sbindir=/usr/local/sbin
sharedstatedir=/usr/local/com
srcdir=.
sysconfdir=/usr/local/etc
top_srcdir=.
CFLAGS=
CPPFLAGS=
$ head Makefile
# Generated automatically from Makefile.in by configure.
# Ez egy Makefile lesz, és a köv sorban, látható lesz az előző sorbeli megjegyzés
# Generated automatically from Makefile.in by configure.
bindir=${exec_prefix}/bin
datadir=${prefix}/share
exec_prefix=${prefix}
includedir=${prefix}/include
infodir=${prefix}/info
libdir=${exec_prefix}/lib
libexecdir=${exec_prefix}/libexec
$ CFLAGS=-O2 ./configure --prefix=/opt
loading cache ./config.cache
creating ./config.status
creating Makefile
$ make
bindir=/opt/bin
datadir=/opt/share
```

```
exec_prefix=/opt
includedir=/opt/include
infodir=/opt/info
libdir=/opt/lib
libexecdir=/opt/libexec
localstatedir=/opt/var
mandir=/opt/man
oldincludedir=/usr/include
prefix=/opt
sbindir=/opt/sbin
sharedstatedir=/opt/com
srcdir=.
sysconfdir=/opt/etc
top_srcdir=.
CFLAGS=-O2
CPPFLAGS=
$ cat <<'END' >configure.in
> AC_INIT(Makefile.in)
> AC_PREFIX_DEFAULT(/usr/opt)
> AC_OUTPUT(Makefile)
> END
$ autoconf
$ ./configure
loading cache ./config.cache
creating ./config.status
creating Makefile
$ make
bindir=/usr/opt/bin
datadir=/usr/opt/share
exec_prefix=/usr/opt
includedir=/usr/opt/include
infodir=/usr/opt/info
libdir=/usr/opt/lib
libexecdir=/usr/opt/libexec
localstatedir=/usr/opt/var
mandir=/usr/opt/man
oldincludedir=/usr/include
prefix=/usr/opt
sbindir=/usr/opt/sbin
sharedstatedir=/usr/opt/com
srcdir=.
sysconfdir=/usr/opt/etc
top_srcdir=.
CFLAGS=
CPPFLAGS=
```

2.2.9. Verziószámok

Az alábbi makrókkal lehet verziószámmal kapcsolatos műveleteket végezni a `configure` szkriptekben.

AC_PREREQ(VERZIÓ)

Evvel a makróval meg lehet győződni arról, hogy megfelelően új verziójú autoconfot használunk. Ha az éppen használt autoconf verziója régebbi, mint a *VERZIÓ*, akkor kiír egy hibaüzenetet, és nem készíti el a `configure` szkriptet az `autoconf`. Például:

```
AC_PREREQ(1.8)
```

Ez a makró hasznos lehet, ha a `configure.in` egy az autoconf kiadások közti nem magától értetődő viselkedésbeli különbségre épít. Ha frissiben hozzáadott makrók miatt szeretnénk használni, akkor az `AC_PREREQ` kevésbé hasznos, mert az autoconf enélkül is megmondja a felhasználónak, ha egy makrót nem talál. Ugyanez történik akkor is, ha a `configure.in`t egy olyan régebbi `autoconf`al dolgoztatjuk fel, amelyik még az `AC_PREREQ`et sem ismerte.

AC_REVISION(KIADÁS-INFÓ)

A `configure` szkriptbe bemásolja a *KIADÁS-INFÓ* kiadási bélyeget, a dollár jelek vagy a dupla-idézőjel eltávolításával. Ez a makró lehetővé teszi, hogy egy kiadási bélyeget tegyünk a `configure.in`ből a `configure`ba, anélkül, hogy az RCS vagy a CVS megváltoztatná azt, amikor *check in*eljük[6]. Ilymódon megállíptható, hogy milyen kiadású `configure.in`ből lett generálva a `configure` szkript.

Jó ötlet lehet, hogy ezt a makrót az `AC_INIT` előtt hívjuk meg, így a kiadási szám, közelebb van a `configure.in` és a `configure` fájl tetejéhez. Hogy ezt megtehessük, az `AC_REVISION` kimenete egy `#!/bin/sh` résszel kezdődik, mint ahogy egy normális `configure` szkriptnek kell. Például, ha ez a sor van a `configure.in`ben:

```
AC_REVISION($Revision: 1.30 $)dnl
```

Akkor az egy ilyen sort eredményez a `configure`ban:

```
#!/bin/sh
# From configure.in Revision: 1.30
```

2.2.10. Kész tesztek

Figyelem, ebben a fejezetben nem abban a sorrendben jön a tesztek leírása, mint ahogyan azt a `configure.in`ben kellene írnunk az ajánlás$_{60}$ szerint.

[6] A commit „szinonímája”

Programok létének ellenőrzése

Ezek a makrók arra szolgálnak, hogy ellenőrizzük, hogy létezik-e egy bizonyos program. Akár arra is felhasználhatjuk, hogy több különböző alternatív lehetőség közül válasszunk egyet, és hogy a döntésünk mellett milyen egyéb teendőnk van. Ha a keresett programra nincs külön előre gyártott makró, és a keresett prograram semilyen különleges tulajdonságát nem kell ellenőrizni, akkor használhatjuk az általános programellenörző makrókat.

Az alábbi makrók egyedi programokat ellenőriznek, és bizonyos esetekben azok megfelelő tulajdonságait:

AC_DECL_YYTEXT

Definiálja az `YYTEXT_POINTER`-t, ha az `yytext` `char *`-nak van definiálva `char []` helyett. A `LEX_OUTPUT_ROOT` kimeneti változót is beállítja a lexer által generált alapfájlnévre, ami általában `lex.yy` szokott lenni, de időnként más is lehet. Ezen eredmények erősen függnek attól, hogy `lex`et vagy `flex`et használunk.

AC_PROG_AWK

Ellenőrzi a `mawk`, `gawk`, `nawk` és az `awk` programok meglétét, ebben a sorrendben, és beállítja az `AWK` kimeneti változót arra, amelyet elsőnek találja meg. Azért a `mawk`al kezdi a sort, mert állítólag ez a leggyorsabb implementáció.

AC_PROG_CC

Meghatározza a használandó C fordítót. Ha a `CC` környezeti változó nincs eleve beállítva, akkor ellenőrzi a `gcc`t és sikertelen esetben a `cc`vel is megpróbálkozik. A `CC` kimeneti változó értékét a talált fordító nevére beállítja.

Ha a GNU C fordítót használjuk, akkor a `GCC` környezeti változót is beállítja `yes`re, különben üresen hagyja. Ha a `CFLAGS` nem volt eredetileg beállítva, akkor beállítja azt `-g -O2`re GNU C esetén (illetve `-O2`re az olyan rendszereken, ahol a `-g`t nem fogadja el a gcc) vagy `-g`-re az egyéb fordítók esetén.

Ha a C fordító olyan végrehajtható állományt generál, ami nem képes futni azon a rendszeren, ahol a `configure` szkript éppen fut, akkor a `cross_compiling` környezeti változót `yes`re állítja, egyébként `no`ra. Másszóval, ellenőrzi, hogy az a rendszer ahol a programot fordítjuk, mint ahol futtatni fogjuk.

Lássunk egy példát, hogy is működik ez a makró:

```
$ cat <<'END' >Makefile.in
> CFLAGS= @CFLAGS@
> CC= @CC@
> cross_compiling= @cross_compiling@
>
> .PHONY: test
> test:
>         @echo "CFLAGS=$(CFLAGS)"
>         @echo "CC=$(CC)"
>         @echo "cross_compiling=$(cross_compiling)"
```

```
> END
$ cat <<'END' >configure.in
> AC_INIT(Makefile.in)
> AC_PROG_CC
> AC_SUBST(cross_compiling)
> AC_OUTPUT(Makefile)
> END
$ autoconf
$ ./configure
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
$ make
CFLAGS=-g -O2
CC=gcc
cross_compiling=no
```

AC_PROG_CC_C_O

Ellenőrzi, hogy a fordító elfogadja-e egyszerre a `-c` és `-o` paramétereket, és ha nem, akkor a `NO_MINUS_C_MINUS_O`t definiálja.[7]

AC_PROG_CPP

Beállítja a `CPP` kimeneti változót arra a parancsra, amely képes futtatni a C előfeldolgozót. Ha a `$CC -E` nem működne, akkor a `/lib/cpp`re állatja. Csak akkor általánosan hordozható a `CPP` futtatása, ha a fájlnak `.c` kiterjesztése van.

Ha az aktuálisan használt nyelv a C, akkor számos más makró használja közvetve a `CPP` értékét, például a `AC_TRY_CPP`, `AC_CHECK_HEADER`, `AC_EGREP_HEADER` vagy az `AC_EGREP_CPP`.

AC_PROG_CXX

Meghatározza a használandó C++ fordítót. Ellenőrzi a `CXX` illetve a `CCC` környezeti változókat (ebben a sorrendben), hogy be vannak-e állítva, és ha igen, akkor a `CXX`-et erre az értékre állítja. ...Lásd alábbi táblázat.

AC_PROG_CXXCPP

Beállítja a `CXXCPP` kimeneti változó értékét arra a parancsra, ami képes a C++ előfeldolgozót futtatni. ...Szintén hasonlóképp viselkedik mint az `AC_PROG_CPP`: Ha a `$CXX -E`

[7] Ha valaki tudja, hogy mi értelme van egy C makró definiáltságában ellenőrizni a C fordító egy tulajdonságát, az ossza meg velem!

nem megy, akkor `/lib/cpp`, illetve ez is csak akkor hordozható, ha a fájlnak, amin használjuk `.c`, `.C` vagy `.cc` kiterjesztése van.

AC_PROG_F77

A használandó Fortran 77 fordítót határozza meg. ...

Eltérések a `AC_PROG_CC`hez képest:

	`AC_PROG_CC`	`AC_PROG_CXX`	`AC_PROG_F77`
ellenőrzött környezeti változó	`CC`	`CXX`, `CCC`	`F77`
kipróbálandó fordító	`gcc`, `cc`	`c++`, `g++`, `gcc`, `CC`, `cxx`, `cc++`	`g77`, `f77`, `f2c`
beállított környezeti változó	`GCC`	`GXX`	`G77`
fordítóhoz paraméterei	`CFLAGS`	`CXXFLAGS`	`FFLAGS`

A `cross_compiling` ellenőrzést mindhárom makró elvégzi.

AC_PROG_F77_C_O

Ellenőrzi, hogy a fordító elfogadja-e egyszerre a `-c` és `-o` paramétereket, és ha nem, akkor az `F77_NO_MINUS_C_MINUS_O`t definiálja.

AC_PROG_GCC_TRADITIONAL

A `CC` kimeneti változóhoz hozzáteszi a `-traditional` kapcsolót is, ha a GNU C fordítót használjuk, és az *ioctl*ek nem működnek enélkül megfelelően. Ez általában akkor történik meg, ha a fix fejlécállományokat nem telepítettünk fel egy régi rendszerre. Lévén az újabb GNU C verziók már kijavítják a fejlécállományokat automatikusan, ez egyre kevésbé jelentős probléma.

AC_PROG_INSTALL

Beállítja a `INSTALL` kimeneti változót egy a BSD `install` programjával kompatibilis program nevére, ha talál egy ilyet a `PATH`ban. Egyébként a `DIR/install-sh -c` lesz az értéke, aholis a *DIR* a már említett `AC_CONFIG_AUX_DIR` makró által meghatározott érték. Ez a makró beállítja az `INSTALL_PROGRAM` és `INSTALL_SCRIPT` változókat az `${INSTALL}` értékre és az `INSTALL_DATA` környezeti változó értékét `${INSTALL} -m 644` re.

...

Ha saját plussz tulajdonságokkal felruházott `install` programot szeretnénk használni, akkor ne ezt a makrót használjuk, hanem egyszerűen írjuk be a nevét a `Makefile.in`ünkbe.

AC_PROG_LEX

Ha talál `flex`-et akkor a `LEX` kimeneti változó értékét `flex`re állítja és a `LEXLIB`et pedig `-lfl`re, ha ez a könyvtár (*library*) a rendes helyén megtalálható. Egyébként beállítja a `LEX`et `lex`re és a `LEXLIB`et `-ll`re.

AC_PROG_LN_S

Ha az `ln -s` működik az aktuális fájlrendszeren (mind az operációs rendszer, mind a használt fájlrendszer támogatja a szimbolikus linkeket), akkor az `LN_S` környezeti változót `ln -s`re állítja, egyébként `ln`re.

Ha ez aktuális könyvtártól eltérő könyvtárba teszünk linket, akkor a működése eltér attól függően, hogy `ln` vagy `ln -st` használunk-e. Hogy biztonságos módon hozzunk létre linkeket az `$(LN_S)` használatával, vagy mindig próbáljuk kitalálni, hogy épp melyik változat van érvényben, vagy mindíg abból a könyvtárból hívjuk meg a ahol a linket létre kell hozni. Vagyis ehelyett:

```
$(LN_S) izé /x/mizé
```

használjuk ezt:

```
(cd /x && $(LN_S) izé mizé)
```

AC_PROG_RANLIB

A `RANLIB` kimeneti változót beállítja `ranlib`re ha talál `ranlib`et, egyébként nem csinál semmit.

AC_PROG_YACC

Ha talál `bison`t, akkor a `YACC` kimeneti változó értékét `bison -y`ra állítja. Egyébként ha `byacc`ot talál, akkor `byacc`ra állítja, és végső elkeseredésében pedig `yacc` lesz az értéke.

Az alább következő makrók arra valók, hogy olyan programokat ellenőrizzünk, amelyeket a fenti makrókkal nem tudtunk ellenőrizni. Ha a program egyéb viselkedési módjait is ellenőrizni szeretnénk (mint pl. az `AC_PROC_CC_C_O` makró csinálja), akkor saját makrókat kell írnunk. Alapértelmezésben az alábbi makrók a `PATH` környezeti változót ellenőrzik. Ha szeretnénk a keresett programot olyan helyen is keresni, ami a `PATH`ban nincs feltüntetve, akkor egy módosított `PATH`t is átadhatunk, valahogy így:

```
AC_PATH_PROG(INETD, inetd, /usr/libexec/inetd,
  $PATH:/usr/libexec:/usr/sbin:/usr/etc:etc)
```

AC_CHECK_FILE(FÁJL, [, TEENDŐ-HA-VAN [,TEENDŐ-HA-NINCS]])

Ellenőrzi, hogy a *FÁJL* létezik-e a rendszerben. Ha megtalálta, akkor végrehajtja a *TEENDŐ-HA-VAN* résznek megfelelő teendőket, egyébként pedig a *TEENDŐ-HA-NINCS* részben előírt teendőket, ha azok adottak.

AC_CHECK_FILES(FÁJLOK, [, TEENDŐ-HA-VAN [,TEENDŐ-HA-NINCS]])

Végrehajtja egyszer az `AC_CHECK_FILE`-t minden egyes felsorolt *FÁJLOK* fájlra. Továbbá definiál `HAVEFILE`-t minden egyes megtalált fájlhoz, és beállítja 1-re.

AC_CHECK_PROG(VÁLTOZÓ, ELLENŐRZENDŐ-PROGRAM, ÉRTÉK-HA-VAN [,ÉRTÉK-HA-NICS [, PATH, [REJ]]])

Ellenőrzi az *ELLENŐRZENDŐ-PROGRAM* létezését a `PATH`ban. Ha megtalálta, akkor beállítja a *VÁLTOZÓ*t az *ÉRTÉK-HA-VAN* értékre, egyébként az *ÉRTÉK-HA-NINCS* értékre, ha az adott. Mindig kihagyja a keresésből a *REJ* fájlt (aminek egy abszolút fájlnévnek kell lennie), még akkor is, hogy ha ezt találja meg elsőnek a keresési `PATH`ban, és beállítja a *VÁLTOZÓ* értékét az *ELLENŐRZENDŐ-PROGRAM*-ra, ami nem a *REJ*. Ha a *VÁLTOZÓ* értéke már be van állítva, akkor nem csinál semmit. Ez a makró meghívja az `AC_SUBST` makrót is a *VÁLTOZÓ*-ra.

AC_CHECK_PROGS(VÁLTOZÓ, ELLENŐRZENDŐ-PROGRAMOK [, ÉRTÉK-HA-NINCS [, PATH]])

Ellenőrzi, minden egyes az *ELLENŐRZENDŐ-PROGRAMOK*ban listázott (white-spaceszel elválaszott lista) program létezését a `PATH`ban. Ha megtalálta, akkor beállítja a *VÁLTOZÓ* értékét a talált program nevére. Egyébként folytatja az ellenőrzést a listában szereplő következő program után. Ha egyik programot se találta meg, akkor a *VÁLTOZÓ* értékét *ÉRTÉK-HA-NINCS*re állítja. Illetve, ha ez nincs megadva, akkor a *VÁLTOZÓ* értéke nem változik. Ez a makró is meghívja az `AC_SUBST` makrót a *VÁLTOZÓ*ra.

AC_CHECK_TOOL(VÁLTOZÓ, ELLENŐRZENDŐ-PROGRAM [, ÉRTÉK-HA-NINCS [, PATH]])

Hasonló az `AC_CHECK_PROG`hoz, de először az *ELLENŐRZENDŐ-PROGRAM*ot egy az `AC_CANONICAL_HOST` által megállapított prefixel (és egy azt követő minusz jellel) ellenőrzi. Így például, ha a `configure` parancsot így adtuk ki: `./configure --host=i386-gnu`, akkor az `AC_CHECK_TOOL(RANLIB, ranlib, :)` makróhívás beállítja a `RANLIB`et `i386-gnu-ranlib`re ha létezik ilyen a `PATH`ban, vagy `ranlib`re, ha legalább az létezik a `PATH`ban, vagy `:`ra, ha az sem.

AC_PATH_PROG(VÁLTOZÓ, ELLENŐRZENDŐ-PROGRAM [, ÉRTÉK-HA-NINCS [, PATH]])

Hasonló az `AC_CHECK_PROG`hoz, de a *VÁLTOZÓ* a program teljes útvonalát veszi fel értékül, ha az *ELLENŐRZENDŐ-PROGRAM*ot megtalálta.

AC_PATH_PROGS(VÁLTOZÓ, ELLENŐRZENDŐ-PROGRAMOK [, ÉRTÉK-HA-NINCS [, PATH]])

Hasonló az `AC_CHECK_PROGS`hoz, de a *VÁLTOZÓ* a program teljes útvonalát veszi fel értékül, ha az *ELLENÖRZENDŐ-PROGRAMOK* egyikét megtalálta.

Könyvtárak (*library*k) ellenőrzése

AC_CHECK_LIB(LIBRARY, FÜGGVÉNY [,TEENDŐ-HA-VAN [, TEENDŐ-HA-NINCS [, EGYÉB-LIBRARYK]]]

Az aktuális nyelvtől függően, megpróbál meggyőződni arról, hogy a C, C++, FORTRAN 77ben elérhető a *FÜGGVÉNY* a *LIBRARY*ban, egy tesztprogramon keresztül, amit ehhez szerkeszt. A *LIBRARY* az alapneve a könyvtárnak, ha pl. a `-lmp` -vel szeretnénk hozzászerkeszteni az `mp` könyvtárat, akkor legyen `mp` az argumentum.

A *TEENDŐ-HA-VAN* shell parancsok listája, amelyet akkor futtat le, ha a könyvtárhoz való szerkesztés sikerült. Ha ez nem definiált, akkor az alapértelmezett teendő hozzáveszi a `-lLIBRARY`-t a `LIBS`hez és definiálja a `HAVE_LIBLIBRARY` szimbólumot, ahol minden csupa nagybetűs.

Ha a *LIBRARY* hozzászerkesztése feloldatlan szimbólumokat eredményezne, mert további könyvtárakat is meg kell adni a szerkesztéshez, akkor azokat az *EGYÉB-LIBRARYK* argumentumként adjuk meg szóközökkel szeparálva, pl. így `-lXt -lX11`. Egyébként a makró hibásan detektálhatja egy könyvtár létezését, mert a tesztprogram összeszerkesztése mindig hibára vezet a feloldatlan szimbólumok miatt.

AC_HAVE_LIBRARY(LIBRARY [, TEENDŐ-HA-VAN [,

TEENDŐ-HA-NINCS [, EGYÉB-LIBRARYK]]]

Ez a makró ekvivalens avval mintha az `AC_CHECK_LIB`-et hívtuk volna meg a *FÜGGVÉNY* paraméterként `main`-t megadva. Továbbá a *LIBRARY* írható `izé`, `-lizé`, vagy `libizé.a` alakban. Mindegyik esetben a fordító a `-lizé` paramétert fogja megkapni. A *LIBRARY* nem lehet shellbeli változó, hanem azt betű szerint érti. Ez a makró már idejétmúltnak tekintendő.

AC_SEARCH_LIBS(FÜGGVÉNY, PRÓBA-LIBEK [, TEENDŐ-HA-VAN [,

TEENDŐ-HA-NINCS [, EGYÉB-LIBRARYK]]]

Hasonló a fentiekhez. A teszt megpróbálja kitalálni, hogy a *FÜGGVÉNY* melyik libraryban van. Először további könyvtár nélkül, majd az *PRÓBA-LIBEK*ben felsoroltakat próbálja az `AC_TRY_LINK_FUNC` makró hozzászerkeszteni a példaprogramhoz. A többi argumentum kezelése, és alapértelmezései, hasonlóak a fentiekhez.[8]

Könyvtári függvények ellenőrzése

Speciális könyvtári függvények tulajdonságainak ellenőrzése:

AC_FUNC_ALLOCA

AC_FUNC_CLOSEDIR_VOID

AC_FUNC_FNMATCH

AC_FUNC_GETLOADAVG

AC_FUNC_GETMNTENT

AC_FUNC_GETPGRP

AC_FUNC_MEMCMP

AC_FUNC_MMAP

AC_FUNC_SELECT_ARGTYPES

AC_FUNC_SETPGRP

AC_FUNC_SETVBUF_RESERVED

[8]Például használhatjuk arra, hogy ha hálózatos programot írunk, ki tudjuk találni, hogy az `nsl` library-t hozzá kell-e szerkeszteni a socket() hívás miatt a programhoz.

AC_FUNC_STRCOLL

AC_FUNC_STRFTIME

AC_FUNC_UTIME_NULL

AC_FUNC_VFORK

AC_FUNC_VPRINTF

AC_FUNC_WAIT3

Általános függvényekkel kapcsolatos tesztek:

AC_CHECK_FUNC(FÜGGVÉNY [, TEENDŐ-HA-VAN [, TEENDŐ-HA-NINCS]]

AC_CHECK_FUNCS(FÜGGVÉNY ... [, TEENDŐ-HA-VAN [, TEENDŐ-HA-NINCS]]

AC_REPLACE_FUNCS(FÜGGVÉNY ...)

Fejléc állományok meglétének ellenőrzése

Speciális fejlécek ellenőrzése:

AC_DECL_SYS_SIGLIST

AC_DIR_HEADER

AC_HEADER_DIRENT

AC_HEADER_MAJOR

AC_HEADER_STDC

AC_HEADER_SYS_WAIT

AC_MEMORY_H

AC_UNISTD_H

AC_USG

Általános fejlécellenőrző makrók:

AC_CHECK_HEADER(FEJLÉCFÁJL [, TEENDŐ-HA-VAN [, TEENDŐ-HA-NINCS]])

AC_CHECK_HEADERS(FEJLÉCFÁJL ... [, TEENDŐ-HA-VAN [, TEENDŐ-HA-NINCS

Struktúrák, illetve a tagjainak ellenőrzése

AC_HEADER_STAT

AC_HEADER_TIME

AC_STRUCT_ST_BLKSIZE

AC_STRUCT_ST_BLOCKS

AC_STRUCT_ST_RDEV

AC_STRUCT_TM

AC_STRUCT_TIMEZONE

Típusdefiníciók ellenőrzése

Speciális típusdefiníciók ellenőrzése:

AC_TYPE_GETGROUPS

AC_TYPE_MODE_T

AC_TYPE_OFF_T

AC_TYPE_PID_T

AC_TYPE_SIGNAL

AC_TYPE_SIZE_T

AC_TYPE_UID_T

Általános típusdefiníció ellenörző makrók:

AC_CHECK_TYPE(TÍPUS, ALAPÉRTELMEZÉS)

Programfordító fordítási tulajdonságainak (*karakterisztikájának*) tesztelése

AC_C_BIGENDIAN

AC_C_CONST

AC_C_INLINE

AC_C_CHAR_UNSIGNED

AC_C_LONG_DOUBLE

AC_C_STRINGIZE

AC_CHECK_SIZEOF(TÍPUS [, KERESZT-MÉRET])

AC_INT_16_BITS

AC_LONG_64_BITS

AC_F77_LIBRARY_LDFLAGS

Rendszerszolgáltatások ellenőrzése

AC_CYGWIN

AC_EXEEXT

AC_OBJEXT

AC_MINGW32

AC_PATH_X

AC_PATH_XTRA

AC_SYS_INTERPRETER

AC_SYS_LONG_FILE_NAMES

AC_SYS_RESTARTABLE_SYSCALLS

Unix változatokra vonatkozó ellenőrzések

AC_AIX

AC_DYNIX_SEQ

AC_IRIX_SUN

AC_ISC_POSIX

AC_MINIX

AC_SCO_INTL

AC_XENIX_DIR

2.2.11. Hogyan írjunk saját teszteket

2.3. A GNU automake

2.3.1. Bevezető

Az `automake` egy olyan eszköz amely elkészíti helyettünk a `Makefile.in` fájlt a `Makefile.am` alapján. Minden egyes `Makefile.am` alapvetően egy sor makródefiníció a MAKE számára. A generált `Makefile.in` megfelel a GNU MAKEFILE szabványainak.[29]

A *GNU Makefile Szabványok Dokumentuma* nagy, komplikált és hajlamos a változásra. Az `automake` célja, hogy ne a Makefile karbantartásával menjen el a programozó ideje (hanem az legyen az az `automake` karbantartójának gondja).

Egy tipikus `automake` bemeneti fájl egyszerűen néhány makró definicióból áll. Minden egyes ilyen fájlból készül egy `Makefile.in`. Így a projekt minden egyes könyvtárának tartalmaznia kell egy `Makefile.am` fájlt.

Az `automake` a projektre vonatkozóan számos kötöttséggel jár; például avval, hogy feltételezi, hogy `autoconf`ot is használunk, valamint ránkkényszerít néhány megszorítást a configure.in tartalmával kapcsolatban.

Az `automake`nek szüksége van a PERLre is hogy legenerálja a `Makefile.in`-t, de ez csak a projtekből disztributálandó fájlt elkészítő gépen szükséges (vagyis ahol, az `automake` parancsal a `Makefile.in`-t elkészítjük).

2.3.2. Általános működés

Az `automake` elolvssa a `Makefile.am` fájlt és generál belőle egy `Makefile.in`t. Számos makró és target amit a `Makefile.am`ben definiálunk arra utasíthatja az `automake`et, hogy specializáltabb kódot készítsen. Például a `bin_PROGRAMS` makró definíciója olyan hatással van a targetekre, hogy lefordítsák és összeszerkesszék a generálandó programot.

A `Makefile.am`ben szereplő makródefiníciók és targetek pontosan (betűről betűre) átmásolódnak a generált `Makefile.in`be.

Figyelem a GNU MAKE kiterjesztéseit nem ismeri fel az automake. Ha ezeket a kiterjesztéseket használjuk a `Makefile.am`-ben akkor hibához és nemvárt viselkedéshez vezethet!

Ha egy targetet definiálunk a `Makefile.am`ben és létezik olyan target amit ugyanilyen néven az automake is létrehozna alapértelmezésben, akkor a „miénk" fog érvényre jutni. Habár ez egy támogatott tulajdonság, általában jobb kerülni (mert időnként a generált szabályok nagyon finnyásak lehetnek).

Hasonlóképpen ha a `Makefile.am`ben definiálunk egy makrót, akkor az az `automake` által alapértelmezetten generáltat felülbírálja. Vedd figyelembe, hogy az `automake` által generált makrók a saját belső használatára készülnek, és a nevük a későbbi `automake` kiadásokban változhat.

Amikor makró definíciókat vizsgálunk, akkor tudnunk kell, hogy az automake rekurzívan megvizsgálja a hivatkozott makró definícióját. Például, ha az `automake` egy ilyen részlettel találkozik:

```
xs = a.c b.c
izé_SOURCES = c.c $(xs)
```

Akkor az `a.c`, `b.c` és `c.c` fájlokat veszi a `izé_SOURCES` tartalmául.

Az `automake` lehetővé teszi olyan megjegyzések írását is ami **nem** másolódik át a kimenetére. Minden ##el kezdődő sort teljesen figyelmenkívül hagy az `automake`. Például érdemes lehet a `Makefile.am` első sorát így kezdeni:

```
## Dolgoztasd fel ezt a fájlt az automake-el, hogy készítsen belőle egy Makefile.in-t.
```

2.3.3. Könyvtár mélységek

Az `automake` három különböző könyvtárhierarchiát támogat: `flat`, `shallow` és `deep`.

A „flat" csomag minden fájlját egyszerűen egy könyvtárban tartja. A `Makefile.am` ebben az esetben nem definiálja a `SUBDIRS` makrót. Például ilyen a `termutils` csomag, vagy a múlt órán bemutatott `fdist`.

A „deep" csomag esetén a forráskódok alkönyvtárakban helyezkednek el, a legfelső szintű könyvtár leginkább csak konfigurációs információt tartalmaz. A `cpio` jó példa egy ilyen csomagra, csakúgy mint a GNU `tar`. A felső szintű `Makefile.am` egy „deep" csomagnál tartalmaz egy `SUBDIRS` makrót (illetve annak definícióját), de nem tartalmaz más makrókat amelyek a készítendő objektumokra vonatkoznának.

A „shallow" csomag olyan, ahol a fő forrás a legfelső szintű könyvtárban található, még számos más része (tipikusan könyvtárak/*libraryk*) pedig alkönyvtárakban.

2.3.4. Szigor

Habár az `automake`t arra tervezték hogy a GNU csomagok karbantartói használják, tesznek arra vonatkozóan is erőfeszítéseket, hogy azoknak is hasznos legyen akik nem szeretnék az összes GNU konvenciót követni.

Emiatt az `automake` három szintet támogat a „szigorúságot" illetően, attól függően hogy mennyire szeretnénk, hogy az automake a szabványoknak való megfelelést ellenőrizze. Az érvényes szigorúsági szintek:

foreign Az `automake` csak azokat a funkciókat ellenőrzi, amelyek abszolút szükségesek a megfelelő működéshez. Például a GNU Szabványok előírják a `NEWS` fájl létezését, ez nem szükséges ebben a módban. A név abból a tényből származik, hogy az `automake`t arra szánták, hogy a GNU programokhoz használják; így ezek a görcsös szabályok nem szükségesek a működéshez.

gnu Az `automake` annyi ellenőrzést végez, amennyit csak lehet, hogy betartassa a GNU előírásait. Ez az alapértelmezett.

gnits Az `automake` a még iratlan „Gnits szabványok"nak való megfelelőséget ellenőrzi. Ezek a GNU szabványokon alapulnak, de sokkal részletesebbek. Ameddig el nem készül a Gnits és nem publikálják, addig javallott nem ezt használni.

2.3.5. Az egyésges nevezéktani séma

Az `automake` makrók (innentől *változó*ként hivatkozunk rájuk) általában egy általános nevezéktani sémát követnek, ami egyszerűvé teszi, hogy eldöntsük, hogy a programok hogyan

készülnek, és hogyan települnek. Ez a séma támogatja például azt, hogy a `configure` parancs futásakor dőljön el, hogy minek kell lefordulnia.

A `make` futásakor a megfelelő változókat használjuk arra, hogy meghatározzuk mely objektumokat kell megépíteni. Ezeket a változókat hívjuk „elsődleges változóknak". Például a `PROGRAMS` nevű elsődleges változó tartalmaz egy listát azokról a programokról amelyeket le kell fordítani, és össze kell szerkeszteni.

A változók egy másik részét arra használjuk, hogy meghatározzuk, hogy az elkészült objektumokat hova telepítsük. Ezeket a változókat az elsődleges változók után nevezzük el, de megmarad prefixként a szabványos könyvtár, amit telepítő könyvtárnak használnánk. A szabványos könyvtárnevek meg vannak határozva a *GNU* szabványokban. Az automake kiterjeszti ezt a `pkglibdir`, `pkgincludedir` és a `pkgdatadir`el. Ezek hasonlók a nem`pkg`s változathoz, csak még a `@PACKAGE@` hozzáadódik a végéhez. Például a `pkglibdir`t `$(libdir)/@PACKAGE@`ként definiálja.

Minden egyes elsődleges változóhoz létezik egy további változó, ahol a nevét az `EXTRA_` prefixummal jelöljük meg az elsődleges változathoz képest. Ezt a változót arra használjuk, hogy felsoroljunk objektumokat amelyeket talán megépítünk, talán nem, függően attól, hogy a `configure` hogyan döntött. Ezt a változó azért kell, mert az `automake`nek statikusan tudnia kell a teljes listáját azon objektumoknak amelyet lehet hogy meg kell építenie, és olyan `Makefile.in`t kell generálnia hozzá, ami minden esetben működik. Például a `cpio` eldönti a `configure` futtatásakor, hogy mely programokat kell megépítenie. Ezek közül bizonyos programokat a `bindir`be és bizonyosakat a `sbindir`be kell telepítenie:

```
EXTRA_PROGRAMS = mt rmt
bin_PROGRAMS = cpio pax
sbin_PROGRAMS = @PROGRAMS@
```

Ha egy elsődleges változót prefix nélkül definiálunk (például `PROGRAMS`) az hibát jelent.

Az általános `dir` utótag lemarad amikor változóneveket konstruálunk, így pl. `bin_PROGRAMS`-t kell írni és nem `bindir_PROGRAMS`-t.

Nem minden fajta objektumot lehet bármelyik könyvtárba telepíteni. Az `automake` figyeli az ilyen próbálkozásokat és hibát jelez. Az `automake` diagnosztizálja azt is, ha könyvtárneveket elgépelünk.

Néha a szabványos könyvtára kevésnek bizonyulnak, még akkor is, ha azokat némileg feljavítja az `automake`. Különösen hasznos lehet, ha objektumokat valamilyen előre definiált alkönyvtárba szeretnénk telepíteni. Evégett az `automake` lehetővé teszi, hogy kiterjesszük a lehetséges telepítési könyvtárakat. Legyen adott egy prefix (például `zar`) amit érvényesnek tekint, ha a hasonló nevű változóhoz a `dir`t hozzáadjuk, és az definiált (vagyis `zardir`).

Például, ameddig a html támogatás nem része az `automake`nek addig megtehetjük, hogy ezt használjuk a nyers html dokumentáció telepítésére:

```
htmldir = $(prefix)/html
html_DATA = automake.html
```

A speciális `noinst` prefix azt jelzi, hogy a kérdéses objektumokat egyáltalán nem kell telepíteni.

A speciális `check` prefix azt jelzi, hogy a kérdéses objektumokat nem lehet elkezdeni építeni ameddig a `make check` parancs le nem futott.

A lehetséges elsődleges nevek: `PROGRAMS`, `LIBRARIES`, `LISP`, `SCRIPTS`, `DATA`, `HEADERS`, `MANS` és `TEXINFOS`.

Hogyan származtatjuk a változók neveit

Megesik, hogy egy `Makefile` változónév valamilyen a felhasználó által megadott szövegből származik. Például programneveket újraírunk `Makefile` makrónevekké. Az `automake` kanonizálja ezt a szöveget így annak nem kell követnie a `Makefile` makróelnevezési szabályokat. Minden karaktert a névben kivéve a betűket, számokat és az aláhúzásjelet aláhúzásjellé változtat amikor makróhivatkozásokat készít. Például ha a programot `izé-mizének` hívják, akkor a származtatott változónév `izé_mizé_SOURCES` lesz és nem `izé-mizé_SOURCES`.

2.3.6. Példák

Teljes

Tegyük fel, épp most fejeztem be az `fdist` megírását. Szeretném az `autoconf`ot használni a hordozhatósági keretmunkák végett, de a `Makefile.in` csak ad-hoc módon lett összedobálva, és bolondbiztosat szeretnék helyette.

Az első lépés hogy a `configure.in`t frissítsem ennek megfelelően, hogy az `automake` által igényelt részek is belekerüljenek. A legegyszerűbb módja, hogy ezt megtegyük, az az `AM_INIT_AUTOMAKE` makróhívás hozzáadása az `AC_INIT` után:

```
AM_INIT_AUTOMAKE(fdist, 0.1)
```

Mivel a programba nem tettem semmit, ami megnehezítené a dolgom (például, nem használja a gettextet, helyette vegyesen angolul és magyarul káromkodik, és nem is akar osztott könyvtárat készíteni), evvel a résszel kész vagyunk. (Természetesen még az `AC_CONFIG_HEADER`t le kell cserélni `AM_CONFIG_HEADER`re.)

Most újragenerálhatom a `configure`omat, de mielőtt ezt megtenném, az `autoconf`al közölnöm kell hogyan talája meg az általam használt új makrót. A legegyszerűbb módja ennek az `aclocal` program meghívása, ami generál egy `aclocal.m4` fájlt nekem. De ha netán már lett volna egy ilyen fájlom, mert némi csicsázó makrót itt írtam volna meg magamnak, akkor az `aclocal` lehetővé teszi számomra, hogy a saját makróimat inkább az `acinclude.m4`be tegyem, így csak át kell neveznem a „saját” `aclocal.m4`emet `acinclude.m4`re, majd a többi:

```
$ mv aclocal.m4 acinclude.m4
$ aclocal
$ autoconf
```

Itt az ideje, hogy megírjam a `Makefile.am`-et az `fdist`hez. Lévén az `fdist` egy felhasználói program, oda szeretném telepíteni, ahova a felhasználói programok mennek. Továbbá feltételezhetjük, hogy lesz némi Texinfo dokumentációja. Ha a `configure.in` scriptem használná az `AC_REPLACE_FUNCS`ot is, akkor a `@LIBOBJS@`hoz hozzá kellene szerkesztenem. Így az egész így nézne ki, amit bele kell írjak a `Makefile.am`be:

```
bin_PROGRAMS = fdist
fdist_SOURCES = fdist.c fdist.h md5.c md5.h
fdist_LDADD = @LIBOBJS@

info_TEXINFOS = fdist.texi
```

Most lefuttathatom az `automake --add-missing` parancsot, ami legenerálja a Makefile.in-t és begyűjti a szükséges segéd fájlokat, amire szükségem lehet, és kész is vagyok.

Hello

A GNU HELLO[9] híres az egyszerűségéről és a sokoldalúságáról. Ez a rész bemutatja, hogyan lehet a GNU HELLO csomagot automakessé tenni. Az alábbi példák a béta verzióból származnak.

Természetesen a GNU HELLO valamivel többet tud mint a tradícionális kétsoros változat. A GNU HELLO nemzetköziesítve van, opciófeldolgozást csinál, és van kézikönyve és egy tesztkörnyezete. A GNU HELLO egy „deep" csomag.

Ime a `configure.in` hozzá:

```
dnl Dolgoztasd fel az autoconf-al, hogy kapj egy configure scriptet.
AC_INIT(src/hello.c)
AM_INIT_AUTOMAKE(hello, 1.3.11)
AM_CONFIG_HEADER(config.h)
dnl Az elérhető nyelvek
ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.
AC_PROG_CC
AC_ISC_POSIX

dnl Checks for libraries.

dnl Checks for header files.
AC_STDC_HEADERS
AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.
AC_FUNC_ALLOCA

dnl Check for st_blksize in struct stat
AC_ST_BLKSIZE

dnl internationalization macros
AM_GNU_GETTEXT
AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \
           src/Makefile tests/Makefile tests/hello],
   [chmod +x tests/hello])
```

Az `AM_` makrókat az `automake` biztosítja, a nagyja pedig `autoconf` makró.

Íme a felső szintű `Makefile.am`:

```
EXTRA_DIST = BUGS ChangeLog.0
SUBDIRS = doc intl po src tests
```

Mint látható az igazi munka itt az alkönyvtárakban folyik. A `po` és az `intl` könyvtárakat automatikusan hozza létre a `gettextize` használata, amelyre most nem térek ki.

[9] url:`ftp://prep.ai.mit.edu/pub/gnu/hello-1.3.tar.gz`

Íme a doc/Makefile.am:

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

Ez a rész a GNU HELLO kézikönyvének építéséhez, telepítéséhez és terjesztéséhez szükséges.

A test/Makefile.am:

```
TESTS = hello
EXTRA_DIST = hello.in testdata
```

A hello egy a configure által generált script, és ez az egyetlen teszt eset. A make check ezt a tesztet fogja futtatni.

És végül a src/Makefile.am, ahol az igazi munka folyik:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
hello_LDADD = @INTLLIBS@ @ALLOCA@
localedir = $(datadir)/locale
INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"
```

etags

Íme egy másik trükkös példa. Bemutatja, hogy generáljunk két programot (az ctagsot és az etagsot), ugyanabból a közös forrás fájlból (etags.c). A nehéz része az etags.c minden fordításakor más opciókat kell a preprocesszornak átadni.

```
bin_PROGRAMS = etags ctags
ctags_SOURCES =
ctags_LDADD = ctags.o

etags.o: etags.c
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
        $(COMPILE) -DCTAGS -o ctags.o -c etags.c
```

Figyeljük meg, hogy a ctags_SOURCESt üresnek definiáltuk, ilymódon nem lesz implicit érték behelyettesítve. Az implicit érték egyébként amit az etags generálásához használ az etags.o.

A ctags_LDADDot arra használjuk, hogy a ctags.o belekerüljön a szerkesztéskor a parancssorba. A ctags_DEPENDENCIESt az automake generálja.

A fenti szabályok nem fognak működni, ha a fordító nem fogadja el egyszerre a -c és a -o kapcsolókat. A legegyszerűbb javítás ha a evvel együtt koholt függőségeket is bemutatunk (ami amiatt kell, hogy ne okozzon problémát két párhuzamosan futó make):

```
etags.o: etags.c ctags.o
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
        $(COMPILE) -DCTAGS -c etags.c && mv etags.o ctags.o
```

Ezek az explicit szabályok nem működnek, ha a deANSIfikációt is bekapcsoljuk. Ha ezt is támogatni szeretnénk, akkor a következő kell:

```
etags._o: etags._c ctags.o
        $(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags._o: etags._c
        $(COMPILE) -DCTAGS -c etags.c && mv etags._o ctags.o
```

2.4. A digitális hitelesítés alapjai

2.4.1. Kódolási típusok

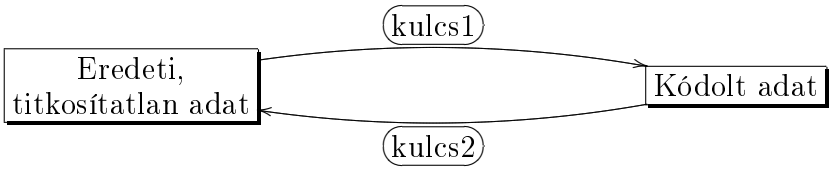
Már elég korán az emberiség történetében felmerült, hogy információkat olyan formában juttassanak el valahova, hogy aki közben esetleg elolvassa, az ne tudja kinyerni belőle a „titkos" információt. Sokféle módszert kitaláltak, de mi csak a „digitálisan" tárolt információk kódolásának egy részét tekintjük át.

A titkosítási algoritmusok alapvetően két csoportra oszthatók:

szimmetrikus kulcsú titkosítások Ugyanazt a közös „titkot" kell ismerni a kódolt információ visszanyeréséhez, és eltitkosításához is (például egy jelszót). Nagy adatmennyiség esetén is viszonylag nehezen fejthető vissza. Ilyen pl. a DES, 3DES, BLOWFISH. A DES-t pl. az ssh 1-es protokollban használták az adatfolyam elkódolására, használata a kriptografikus gyengesége miatt ellenjavalt. A 3DES ennek olyan módosítása, ahol a kódolandó adatfolyam egy kódolás-dekódolás-kódolás hármason megy át, 3 különböző kulccsal. A BLOWFISH egy blokkos jellegű, gyors titkosítási algoritmus.

2.4. ábra. szimmetrikus kulcsú titkosítások sémája

asszimetrikus kulcsú titkosítások A titkosítotáshoz itt is szükség van egy kulcsra, de a visszakódoláshoz már egy másik kulcs kell. Ez a két kulcs egymásból nem számítható ki. Az ilyen jellegű algoritmusok arra a tényre támaszkodnak, hogy a nagy számok prímfaktorizációja nehezen megoldható. A két kulcs szerepe teljesen „szimmetrikus", olyan értelemben, hogy ha az egyikkel kódoljuk el az adatot akkor a másikkal tudjuk dekódolni. Nem érdemes nagy mennyiségű, kis entrópiájú adatot ilymódon elkódolni, mert avval csak azok dolgát könnyítjük meg, aki megpróbálja a nem ismert kulcsot kitalálni. (Általában a kulcspár egyik felét közzé szokták tenni.) Ilyen titkosítási algoritmusok pl. RSA, DSA, El Gammal.



2.5. ábra. asszimmetrikus kulcsú titkosítások sémája

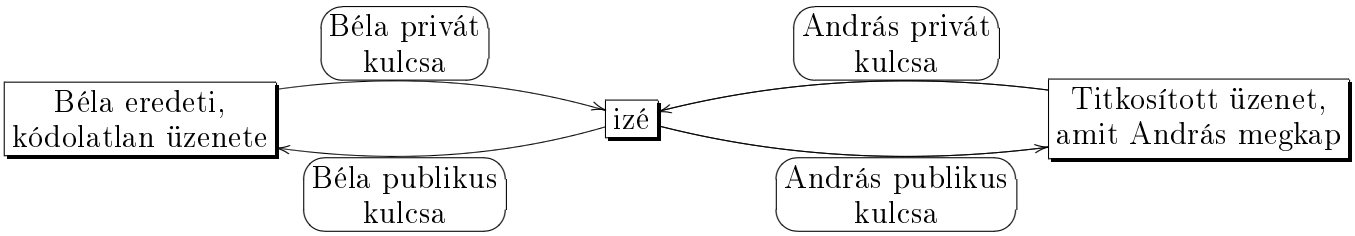
Ezt a típusú kódolást többféleképpen is lehet alkalmazni. Az egyik cél, hogy úgy küldjünk valakinek információt, hogy azt senki más ne tudja elolvasni, csak a címzett. A másik amire még lehet használni, hogy valaki úgy adjon közre információt, hogy avval „hitelesen" bizonyítja, hogy azt az információt ő tette közzé. De ezt a két lehetőséget akár kombinálhatjuk is. Nézzünk konkrét példákat:

András generál magának egy kulcspárt. A kulcspár egyik felét megtartja magának, ez lesz a privát kulcsa; a kulcspár másik felét közzéteszi, ez lesz a publikus kulcsa. Ha

szeretnék Andrásnak üzenni, úgy hogy csak ő tudja elolvasni, akkor az András által közzétett kulccsal eltitkosítom az üzenetem, így azt csak András tudja visszafejteni az ő privát kulcsával.

Nézzünk egy másik példát, ahol Béla szeretne közzetenni információt úgy, hogy biztosak lehessünk benne, hogy azt Béla tette közzé. Ekkor Béla a privát kulcsával elkódolja az információt, és az ilymódon elkódolt információt teszi közzé. Ezt az elkódolt információt viszont bárki visszanyerheti, hiszen, Béla a publikus kulcsát közzétette, és mi pedig biztosak lehetünk benne, hogy az információt Béla tette közzé, hiszen Béla privát kulcsa egyedül Bélának van meg, és azt tudjuk, hogy csak Béla privát kulcsával kódolhatták el az infót, hiszen a dekódoláshoz Béla publikus kulcsa kellett.

Végül nézzünk egy olyan példát, ahol Béla szeretne információt eljuttatni Andrásnak, úgy hogy csak András tudja elolvasni ezt, és András biztos lehessen benne, hogy Bélától származik a kódolt információ: Ekkor Béla az üzenetét elkódolja a saját privát kulcsával (ezt az átmeneti „információt", most hívjuk IZÉnek), és az így kapott eredményt elkódolja András publikus kulcsával. Ezt a legvégső fázisban kapott információt elküldi Andrásnak. Andrásnak most az a dolga, hogy a (feltehetően) Bélától kapott információt a saját privát kulcsával elkódolja (ami ebben az esetben épp „dekódolás"). Ekkor András megkapja IZÉt, amiről feltételezi, hogy Béla küldte neki. András most IZÉt elkódolja Béla publikus kulcsával, aminek eredményéül megkapja Béla eredeti üzenetét titkosítatlanül, és biztos lehet benne, hogy az IZÉ Bélától származik, hiszen azt Béla publikus kulcsával tudta dekódolni.



2.6. ábra. titkos üzenet küldésének sémája

Mint azt említettem, nagy mennyiségű (és kis entrópiájú) adatra nem érdemes asszimetrikus kulcsú titkosítást használni, mert evvel támadási felületet nyújtunk a kulcspár ismeretlen felének visszafejtéséhez. A gyakorlatban, pl. az ssh program az asszimetrikus kulcsú titkosítást arra használja, hogy a szerver és a kliens egy (nem túl) rövid, random, nagy entrópiájú titkosítási kulcsban megeggyezzen, és a tényleges adatfolyam pedig 3DES vagy BLOWFISHel lesz elkódolva. (Lásd ssh kézikönyv -c opció: cipher kiválasztása.)

Az ssh az asszimetrikus kulcsú kódolásokat arra is használja, hogy a felhasználó azonosítást ezen keresztül oldja meg: A kliens oldalon nem egy jelszót kell begépelnünk, hanem egy kulcs privát felével kell rendelkeznünk, valamint a túloldalra el kell juttatnunk a kulcs publikus felét. Ekkor a szerver eljuttat a kliensünknek egy random adatot, amit a privát kulcsal el kell kódolnia, és visszaküldenie (Challange-Response). A szervernek az ellenőrzéshez annyi feladata van, hogy a választ a publikus kulcsunkal elkódolva, összehasonlítsa, hogy megegyezik-e az eredetileg küldött „random" adattal. (megj: szintén ajánlatos viszonylag kevés, nagy entrópiájú adatot küldenie a szervernek.)

2.4.2. Kivonatolási algoritmusok

Ha valaki közzé szeretne tenni valamilyen információt (amelyre nem áll fenn, az, hogy kevés, és nagy entrópiájú), akkor nem egy asszimetrikus kulcsú titkosítást fog az adat elkódolására használni. Helyette nem az eredeti üzenetét kódolja el, hanem készít az üzenetéről egy „kivonatot", és a kivonatot az üzenet mellé teszi a privát kulcsával elkódolva. Mivel a kulcspár publikus felét ismerjük, így össze tudjuk hasonlítani, hogy az üzenetre saját magunk által számított kivonat egyezik-e avval, amit a kulcspár publikus felével visszakapunk.

Egy ilyen kivonattól általában a következőket várjuk el: legyen rövid, „jól reagáljon" a változásokra (vagyis, ha az eredeti üzeneten, akár egy bitet is megváltoztatunk, akkor már kapjunk eltérő kivonatot), és lehetőleg legyen nagy az entrópiája. Ilyen kivonatolási algoritmusok:

md5 1992-ben fejlesztette ki az RSA megalkotásában is részt vevő Ron Rivest. 128 bit hosszúságú kivonatot készít. (RFC 1321)

sha1 Amerikai szövetségi információ feldolgozási szabvány. 160 bit hosszúsági kivonatot készít.

2.4.3. gpg/pgp

Mivel az SMTP protokoll (a telnethez hasonlóan) nem biztosította annak lehetőségét, hogy úgy juttasson el egy levelet a célhoz, hogy közben ne lehessen azt elolvasni, ezért egy felsőbb réteg beli megoldást hoztak létre. Ez volt a PGP (Pretty Good Privacy), amely egy nyilt forráskódó, (jobbára) „szabadon használható" megoldás volt. Azonban használt szabadalommal védett megoldásokat (pl. az idea és rsa kódolásokat. megj.: az rsa-ra vonatkozó korlátozás már lejárt, az idea-ra vonatkozó 2007-ben fog), emiatt szükség volt egy teljesen szabad megoldásra. Ez a teljesen szabad megoldás az FSF által készített GnuPG volt. Az OpenPGP-ről az RFC 2440 szól.

Eddig ugyan nem esett szó arról, hogy hogyan lehet megbizonyosodni arról, hogy egy kulcs publikus fele valóban a tulajdonosához tartozik. Ha valakivel szeretnénk privát levelezést folytatni, vagy a levelei hitelességéről meggyőződni, akkor a legbiztosabb mód, ha személyesen találkozunk az illetővel, elkérjük a személyi igazolványát, és a publikus kulcsának „újjlenyomatát" (fingerprint[10]). Pl. ha valaki a Debian projektbe szeretne bekerülni, akkor rendelkeznie kell, egy olyan kulccsal, amit egy már aktív Debian fejlesztő hitelesített, pl. ilyen személyes találkozás útján. A pgp-nél az X.509-el ellentétben ez a bizalmi rendszer nem egy hierarchikus, fába rendezhető módon van megoldva, hanem egy ún. bizalmi hálón alapul: Mindenki egyenrangú, a bizalom „szintje" pedig azon múlik, hogy hanyadik lépésben találunk valaki kulcsához egy azt hitelesítő „ismerőst"[11].

A gpg a Debian projektben is fontos szerepet tölt be. A debian csomagok elkészültekor a debian csomaghoz tartozó `ChangeLog`ot (valamint az elkészült bináris csomag kivonatát) ill. forráscsomagról szóló fájlt gpg-vel alá kell írni a csomag készítőjének, majd a kapott fájlokat az incoming.debian.org anonymous ftp sitejára feltölteni. Így azon országok tehetséges fejlesztői elől sincs elzárva a lehetőség a Debian projekthez való hozzájáruláshoz, ahol az adott ország

[10] Minden kulcshoz tartozik egy kivonatszerű újjlenyomat, amely a kulcsot azonosítja.

[11] Akik már ilyen személyes találkozás módján meggyőződtek arról, hogy a személyhez feltételezett kulcs valóban az adott személyhez tartozik, azok „hitelesítik" egymás kulcsát

törvényei korlátozzák az erős titkosítást használó programok használatát[12] polgárai számára. De pl. a Debian (és más kereskedelmi disztrubúciók) által kiadott hibajegyeket is egy arra jogosult ember hitelesíti, és úgy küldi ki a hibajegyekkel foglalkozó levelezési listára, így ellenőrizhető, hogy nem egy átverésről van szó, amelyben egy trójai program feltelepítését kéri tőlünk, egy idegen rosszindulatú ... ember.

2.4.4. Gyakorlati feladatok órára

- ssh kulcspár generáltatása
- ssh kulcsos azonosítás beállítása
- gpg kulcs(pár) generáltatása (akinek még nincs)
- gpg kulcs publikus felének exportálása, importálása
- más gpg kulcsának aláírása, az aláírt kulcs exportálása, visszaimportálása

[12] Vagyis nem ssh/scp-vel kell feltölteni a fájlokat, és mégis hellytálló módon meg lehet győződni a fájlok hitelességéről.

2.5. Bevezető a program telepíthető csomagba összeállításába

2.5.1. A debian csomagolással kapcsolatos dokumentumok

Ha debian csomagot szeretnénk készíteni, akkor a **maint-guide**, **debian-policy**, **doc-base**, **developers-reference** csomagokat érdemes feltenni, mert ezekből a Debiannal kapcsolatos minden infrastruktúrális kérdésünkre választ kaphatunk.

maint-guide Viszonylag rövid alapozó összeállítás arról, hogy hogyan kezdjünk el debian csomagot készíteni. Mi is ezen fogunk véigmenni (csak magyarul).

debian-policy Az éppen aktuális disztribúció szabályzata. Korábban (a potato idején) ezek az infók a **packaging-manual** csomagban voltak. A **debian-policy** részei a *woody* [13] idején:

Debian Policy Manual A debian csomagokkal kapcsolatos szabályok

FHS Filesystem Hierarchy Standard, a Debianban már a korai idők óta[14] ez a szabványgyújtemény írja le, hogy milyen típusú fájlnak hol kell elhelyezkednie. Pl. A konfigurációs fájlok a `/etc`-be, a rendszer elindításához szüks. felhasználói binárisok a `/bin`-be, sít.

Virtuális csomagok listája A teljes lista a disztribúcióban szereplő virtuális csomagnevekről.

libc6 migráció Egy dokumentum, ami arról szól, hogy hogyan oldhatjuk meg, hogy a régi libc5-re írt programjaink a libc6-ot (AKA glibc2) is támogassák.

Debconf specifikáció A debconf csomagkonfigurációs rendszer használatáról szóló dokumentáció

Debian Java Policy Egy tervezett policy a Debianba csomagolt Java csomagok kezelési módjáról.

Debian Mime Policy Ez a policy leírja, hogy mit kell tenni, hogy a MIME rendszert használjuk a **Debian GNU/Linux** disztribúciónkban, valamint a MIME adatbázisba való beregisztrálás szabályait.

Debian Menu Policy Ez a kézikönyv leírja, hogy a **Debian GNU/Linux** disztrónk Debian menüjébe hogyan tehetünk be elemeket, valamint a menü fejezeteinek hiearchikus struktúráját.

Debian Perl Policy Ez a kézikönyv leírja, hogy a Perl rendszernek milyen igényei vannak a **Debian GNU/Linux** disztrónkban, leírva a fordítását és telepítését az olyan csomagoknak, amelyek adnak illetve használnak, Perlt és Perl modulokat.

Debian Policy Process Description A Debian Policy fejlesztés menetét leíró dokumentum.

[13] A debian disztribúciók a Toy Story című rajzfilm szereplői után kapják a nevüket:
bo → a pásztorlányka, buzz → buzz lightyear, az űrhalyós, rex → a dínó, hamm → a malac, slink(y) → a kutyus, potato → krumpliuraság, woody → a serif, sarge → a katona

[14] Már a Linux Filesystem Standard előtt létezett

doc-base Ez a dokumentum leírja mi az a doc-base és hogyan használhatjuk a Debian rendszerünk online kézikönyveinek karbantartására.

developers-reference A debian fejlesztők kézikönyve a debian szervezeti dolgairól, úgy mint: hogyan lehetünk maintainerek; a debian levelező listái, szerverei, és egyéb gépei; a debian (csomag)archívum; csomagok feltöltése; NMU-k; portolási infók; a csomagok élete; hibák kezelése és követése; a jövendőbeli fejlesztőkkel való kommunikáció; a debian karbantartó eszközeinek áttekintése

2.5.2. Előkészületek a csomagoláshoz

Ahhoz, hogy csomagokat építsünk a **build-essential** csomagra lesz szükségünk, ami jóformán egy metacsomag[15]. Ezen függőségekből néhány fontosabb csomag, ill. néhány további olyan csomag, ami hasznunkra válik, de a **build-essential** nem hozza magával:

binutils Ez a csomag szükséges hogy asszembláljunk, és összeszerkesszünk tárgykódú fájlokat. (A **gcc** csomag, ami build-essential függ a **gcc-2.95**től, ami pedig a **binutils**tól.)

cpp A C előfeldolgozója. Bizonyos, akár nem C-ben írt programok is használhatják. (A **binutils**hoz hasonló úton ez is jön a **build-essential** függőségein keresztül.)

cpio Egy a `tar`-hoz hasonló archiváló eszköz.

file Ez egy olyan program, amely belenéz fájlokba, és bizonyos jellemzők alapján megpróbálja kitalálni, hogy a fájl belsejében milyen adat/program[16] van.

gcc A GNU C fordító

libc6-dev A C könyvtár, a hozzászerkesztéshez szükséges archív, valamint a hozzátartozó fejlécfájlok.

make A GNU Make.

patch Fájlok módosítására használatos eszköz. A diff-el tudunk készíteni szöveges fájlok különbségéről szóló szöveges fájlt, amely leírja a különbséget. A patch pedig képes arra, hogy egy ilyen fájl alapján módosítsa az eredeti fájlt egy új/javított verzióra.

perl A perl interpreter. Nem hozza magával a build-essential, ellenben nagyon nehéz lenne nélküle.

Ezek eddig olyan csomagok voltak, amelyek akkor is hasznosak, vagy szükségesek lehetnek, ha nem debian csomagot szeretnénk készíteni, hanem egy átlagos programot lefordítani. Nézzünk néhány olyan csomagot, ami debian csomag fejlesztésében lesz a segítségünkre[17]:

[15] olyan csomag, amelynek a tartalma szinte üres, viszont függ más csomagoktól. Így ha a csomagot feltelepítjük, akkor az apt hozza magával a csomag függőségeit. Pl. ha valaki egy alap X grafikus rendszert szeretne telepíteni, akkor elég az x-window-system-core csomagot feltennie, és az hozza magával az X szervert, alapfontokat, és néhány nélkülözhetetlen alapprogramot

[16] Program esetén, hogy dinamikusan, vagy statikusan linkelt-e, illetve, milyen architektúrára való, strip-elve van-e, stb.

[17] Ne felejtsük, hogy a csomagkészítés végén a changes és dsc fájlokat hitelesítenünk kell, ezért a gpg is javallott.

dh-make Ha már van egy programcsomagunk, amiből debian csomagot szeretnénk készíteni, akkor ez segítségünkre lesz, hogy ne nulláról kelljen indulnunk. Előregyártott templatek alapján debianizálja a programcsomagunkat, amihez már alig kell hozzányúlnunk. Ha csomagot szeretnénk készíteni, ez nem szükséges, de nagyban megkönnyíti evvel a mechanizmussal a kezdeti munkánkat.

debhelper Ha a **dh-make**t használjuk, akkor erre is mindenképp szükségünk lesz, mert a **dh-make** által debianizált debian csomag használja a **debhelper**ben levő egyszerűsítéseket. Ebben a csomagban olyan mechanizmusok vannak elkészítve, amelyek bizonyos feladatokat automatizálnak, ill. megkönnyítik a policy-k betartását. Példa: `dh_compress`: a 4kbyte-nál nagyobb doksikat `gzip -9`el tömöríti, `dh_strip`: a csomagban található bináris futtathatókat strip-eli, a policynek megfelelően.

devscripts A csomagok fejlesztőinek munkáját megkönnyíti. Pl. a `dch` segít a parancssorból **ChangeLog** bejegyzések elkészítésében, vagy az `uupdate` és `uscan` segíti a csomagunk *upstream*jének követését.

fakeroot A csomagkészítés bizonyos fázisainál rootnak kell lennünk. Pl. a rendszerbe feltelepülő bináris programok a root tulajdonába kell kerüljenek a .deb archívban, hogy ne módosíthassák a felhasználók a rendszer alapköveit pl. trójaiakra. Mivel azonban egy fordítás általában nem igényel root jogosultságokat csak a telepítés, a fakeroot lehetővé teszi, hogy az archívkezelő programok azt higgyék a fájlokról, hogy root tulajdonában vannak, egy előtöltött az eredeti C könyvtár hívásait lefedő fakeroot könyvtáron (library) keresztül. Így lehetőség van arra, hogy a csomagkészítés egyik fázisában se legyen szükség root jogosultságokra.

lintian A policy betartásának ellenőrzésére tartalmaz bizonyos automatizmusokat. Mielőtt egy csomagot a hivatalos debian archívba feltöltenénk, előtte mindenképp ellenőrizzük a lintiannal.

Ha minden szükséges fejlesztőeszköz fenn van, akkor az alábbiakat ne felejtsük el ellenőrizni:

- Ellenőrizzük, hogy nincs-e az általunk becsomagolandó csomag már eleve benne a debianban.
- Nézzünk utána a WNPP[18]-ben, illetve a debian-devel levelezőlista archívumában, hogy nem próbálkozik-e más is a mi programunkat becsomagolni. (Természetesen, ha saját fejlesztésű csomagról van szó, akkor ez az első két lépés kihagyható.)
- A programnak **kell** legyen licensze. Lehetőleg feleljen meg a DFSG[19]nek. Ha nem így lenne, akkor még mindig kerülhet a *contrib* vagy a *non-free* szekciókba. Ha a programnak valami egyedi licensze lenne, ami felett nem tudunk itélkezni, akkor a debian-legal levelezőlistán érdeklődhetünk, hogy nálunk okosabbaknak mi a licenszről a véleményük.
- Ellenőrizzük, hogy a program ne legyen feleslegesen setuid root-os, a legjobb ha sem setuid sem setgid se semmi nem kell neki.

[18]Work Needing for Prospective Packages, `http://www.debian.org/devel/wnpp/`
[19]Debian Free Software Guidelines, `http://www.debian.org/social_contract#guidelines`

- A program nem egy démon, vagy valami, ami feleslegesen valamelyik `/sbin` könyvtárba akar kerülni...
- A programnak legyen egy végrehajtható bináris alakja a fordítás végén
- Lehetőleg legyen jól dokumentált. A policy kiköti, hogy minden végrehajtható bittel ellátott programhoz, ami vmilyen `bin` könyvtárba kerül, kapcsolódjon egy kézikönyv oldal.
- Próbáljuk felvenni a kapcsolatot a program készítőjével, hogy egyetért-e avval, hogy mi becsomagoljuk a debianba a programját[20]. Ez azért is fontos, mert nem árt ha a program *upstream*jével jó a kapcsolatunk. Könnyen lehet, hogy egy az upstreamben levő (pl. programozási) hibát a BTS-en keresztül nekünk jelentenek.
- Lehetőleg ismerjük azt a programot, amit becsomagolunk, és legyen vele némi saját tapasztalatunk.

2.5.3. A programcsomag előkészítése

Miután letöltöttük a programcsomagot, győződjünk meg róla, hogy az a mi rendszerünkön rendesen lefordul, elkészül, és használható program lesz belőle. Pl. gyakorlásképp letölthetjük a szokásos állatorvosi lovunkat az fdist-et. Lépjünk be a programunk könyvtárába, és olvassunk el minden idevonatkozó dokumentációt: `README*`, `INSTALL*`, `*.lsm`, `*.html`. Ezekben meg kell találjunk minden szükséges útmutatást, hogy a programot le tudjuk fordítani, és fel tudjuk telepíteni. Ez a rész programról programra változhat, de a legtöbb modern programnak van egy `./configure` szkriptje, ami a forráskódot konfigurálja és meggyőződik róla, hogy a fordításához minden szükséges előfeltétel a rendelkezésre áll. Miután konfiguráltuk a programot azt általában a `make` parancsal tudjuk lefordítani. Néhány támogatja a `make check` parancsot, amely néhány önellenőrzést hajt végre. A célkönyvtárba telepítést a legtöbb esetben a `make install` parancs végzi.

Most probáljuk meg lefordítani és futtatni a programot, hogy meggyőződjünk róla, hogy minden rendben működik, és nem ront el semmi mást a telepítés vagy futtatás hatására.

Bizonyos esetekben a `make uninstall`t is használhatjuk a telepített fájlok eltávolítására, és a `make clean`t (vagy még jobb a `make distclean`) a készítő könyvtár kitisztítására.

A `dh_make` előtt

A csomagolást lehetőleg egy teljesen tiszta forráskönyvtáron végezzük, de a legjobb, ha frissen kibontott forráskönyvtárral kezdünk.

A csomag helyes elkészültéhez a program nevét csupa kis betűvel kell venni, és a forráskönyvtárnak a *csomagnév-verzió* formát kell követnie.

Ha a program neve több mint egy szó, írjuk egybe, vagy használjunk egy rövidítést. Például, ha a csomagunk neve „Jancsi kicsi editora Xre”, akkor hívhatjuk **jancsikex**nek, vagy **jle4x**nek, vagy amit csak akarunk. Lehetőleg valami elfogadható hosszkorlátot válasszunk, pl. max. 20 karaktert.

Ellenőrizzük a program pontos verzióját is (ami a csomag verzióban is szerepelni fog). Ha a programunkat nem **X.Y.Z** módon verziószámozzák, hanem pl. dátummal, akkor tegyünk a

[20]Ha megfelelő licenszet tett rá, akkor nem igazán áll módjában ellenkezni, de azért jobb a békesség.

verziószám elé egy „0.0”-t (csak azért, hogy ha az upstream egyszer úgy dönt, hogy kiad egy aranyos verziószámot, pl. 1.0-t, akkor fel legyünk rá készülve). Így ha a kiadásunk vagy pillanatképünk (snapshot) dátuma 2003. Április 23., akkor legyen a verziósztringünk **0.0.20030423**. Bizonyos programokat egyáltalán nem számoznak. Ebben az esetben vegyük fel a kapcsolatot az upstream karbantartójával, hogy kiderítsük milyen verziókövetési módszert használ.

A `dh_make` futtatása

Álljunk a program forráskönyvtárába, majd adjuk ki a megfelelően felparaméterezett `dh_make` parancsot. Pl. valahogy így:

```
[pasztor@clyde ~/fdist-0.2]$ dh_make -e pasztor@fsn.hu ../fdist-0.2.tar.gz

Type of package: single binary, multiple binary, or library? [s/m/l] s

Maintainer name : unknown
Email-Address   : pasztor@fsn.hu
Date            : Wed, 23 Apr 2003 02:10:58 +0200
Package Name    : fdist
Version         : 0.2
Type of Package : Single
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. fdist
uses a configure script, so you probably don't have to edit the Makefiles.
[pasztor@clyde ~/fdist-0.2]$
```

A karbantartó nevét, és e-mailcímét automatikusan megpróbálja a program kitalálni. A nevet a `/etc/passwd` alapján (ill. NIS, vagy LDAP adatbázis alapján). Az e-mailcímet pedig a **felhasználónév@gépnév** módon, ahol a gépnevet a `/etc/mailname`-ből veszi. Viszont könnyen előfordulhat, hogy nem, vagy nem csak olyan gépen készítjük a csomagjainkat, ahol ezek jól be vannak állítva, vagy az következik belőle, amit mi szeretnénk a csomagban viszontlátni, ezért javasolt a névre a `DEBFULLNAME` környezeti változót beállítani[21], valamint az e-mailcímre a `DEBEMAIL` környezeti változót, akár a `~/.bashrc` ill. `~/.bash_profile` fájljainkban (feltéve, hogy bash-t használunk, de természetesen mindenki értse ide, a neki valót). Így az előzőt „eltakarítandó” vmi. ilyet teszek:

```
[pasztor@clyde ~/fdist-0.2]$ cd..
[pasztor@clyde ~]$ rm -r fdist-0.2
[pasztor@clyde ~]$ export DEBFULLNAME="PASZTOR Gyorgy"
[pasztor@clyde ~]$ export DEBEMAIL="pasztor@fsn.hu"
[pasztor@clyde ~]$ tar xzf fdist-0.2.tar.gz
[pasztor@clyde ~]$ cd fdist-0.2
[pasztor@clyde ~/fdist-0.2]$ dh_make -s -n -c gpl -f ../fdist-0.2.tar.gz
```

[21] Név gyanánt lehetőleg olyat adjunk meg, ami csak az angol abc betűit tartalmazza. Hasznos dolog a vezetéknevünket csupa nagybetűvel írni, mert nem mindenütt tudják, hogy mi magyarok a vezetéknevünket írjuk előre, és esetleg megsértődünk, ha egy e-mailben, amikor valaki kapcsolatot akar felvenni velünk „Hi Gyorgy” helyett, „Hi Pasztor”-t ír

```
Maintainer name : PASZTOR Gyorgy
Email-Address   : pasztor@fsn.hu
Date            : Wed, 23 Apr 2003 02:28:57 +0200
Package Name    : fdist
Version         : 0.2
Type of Package : Single
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. fdist
uses a configure script, so you probably don't have to edit the Makefiles.
[pasztor@clyde ~/fdist-0.2]$
```

Némi magyarázat az újonnan hozzáadott paraméterekkel kapcsolatban:

-c gpl A debian/copyright fájlt rendesen kitölti, feltételezve, hogy a program licensze gpl. A `dh_make` **gpl**, **lgpl**, **artistic** és **bsd** licenszekhez rendelkezik előgyártott template-tel.

-n Natív debian csomag készüljön: A natív debian csomagoknál a program írója (upstream author), és a debian csomag karbantartója ugyanaz a személy, és csak egy natív (kötőjelet nem tartalmazó) verziószám kapcsolódik a csomaghoz. A nem natív csomagoknál ez a két személy különböző, és sok egyéb adminisztratív eltérés van, pl. a csomag verziószáma az **eredetiverzió-debianalverzió** módon áll össze.

-s Egyszerű bináris csomag készüljön. Így nem kérdez, hogy egyszerű bináris, többes bináris, vagy könyvtár jellegű csomagot építsen.

Fontos, hogy ha egy csomagot egyszer már debianizáltunk, akkor többször már ne adjuk ki a csomag forrásában a `dh_make` parancsot.

2.5.4. A forráskód módosítása

Általában a programok a `/usr/local` könyvtárba telepítik magukat, ellenben a debian csomagok nem használhatják azt a könyvtárat[22], mert az a rendszeradminisztrátor saját használatára van fenntartva. Így a programunk build-rendszerébe bele kell nyúlnunk a Makefileoknál kezdve[23]. Ha egy autoconfot és automake-et használó programmal van dolgunk, akkor ne felejtsük el, hogy a `Makefile`-t hiába piszkáljuk, mert azt a `./configure` futtatása generálja.

A legtisztább dolog, ha felvesszük az upstreammel a kapcsolatot, és egy olyan `Makefile`ja van a programnak, ami támogatja a `DESTDIR` használatát.

2.5.5. A `debian/` alkönyvtár

A `control` fájl: Ez a fájl tartalmaz infót a forráscsomagról: Név, kategória, prioritás, karbantartó, szabványkövetési verzió, stb. Valamint a forráscsomagból készülő bináris csomagokról: csomagnév, architektúra, függőségek, leírás, stb. Az fdist control fájlját vhogy így készítettem el:

[22] Lásd: FHS [9]

[23] Az **fdist**re, és általában az autoconf+automake-s programokra ez nem igaz.

```
Source: fdist
Section: net
Priority: optional
Maintainer: PASZTOR Gyorgy <pasztor@fsn.hu>
Build-Depends: debhelper (>> 3.0.0)
Standards-Version: 3.5.2

Package: fdist
Architecture: any
Depends: ${shlibs:Depends}
Description: A very little program to broadcast files on a network
 A very little program, which can be server and client decided an its
 paramters. The server wait for some client, and then broadcast a file for
 them. All parameters are get from a konfig file. The server is able to
 reread its config file at a sighup, etc.
```

A `copyright` fájl: Ez a fájl a program licenszeléséről szóló infót tartalmazza. Minden a forráscsomagból készült bináris csomagba bekerül `/usr/share/doc/`*`csomagnév`*`/copyright` néven. A mi esetünkben ezt a `dh_make` kitöltötte, mert közöltük vele, hogy a program GPL-es.

A `changelog` fájl: A debian csomag élettörténete. Ha pl. egy a BTS-ben jelentett hibát kijavítunk és ide írjuk, hogy `Closes: #nnnnnn`, akkor a csomag feltöltése után a hibajegy lezáródik a BTS-ben.

A `rules` fájl: Egy makefile, amelyben bizonyos nevű targeteknek kötelezően lennie kell. Ez írja le, hogyan készüljön el a csomag.

2.5.6. Egyéb fájlok a `debian/` alkönyvtárban

A `README.Debian` fájl: Hasznos rövid információkat lehet idetenni a debian csomaggal kapcsolatban. Pl. pár szóban leírhatjuk, ha valami plusz teendő van a csomag használatához, pl. mailman esetén ide pár sorban leírhatjuk, hogy az adminisztrációs felület működéséhez milyen alias-t kell az apache konfigurációjába beírni.

A `conffiles` fájl: Azon fájlok felsorolása, amelyek a csomagban konfigurációs fájlok. A konfigurációs fájlokról is készül MD5 lenyomat, és a frissítésnél ez összehasonlítódik a fájlrendszeren levővel és ha úgy tűnik változott, akkor rákérdez a rendszer, hogy az újat tegye-e fel, vagy megtartsa a régit, vagy mutassa a különbséget, stb. Illetve ha a csomagot letöröljük, akkor a konfigurációs fájlokat nem törli a rendszer.

A `dirs` fájl: Azon létrehozandó könyvtárak neve, amelyet a make install nem hozna létre, ellenben hiányukban panaszkodna. Jelentősége csak akkor van, ha a `dh_installdirs` debhelpert használjuk a csomagépítés megfelelő fázisában.

A `manpage.1.ex` fájl: Egy rövid, üres példa kézikönyv oldal, aminek mentén el lehet indulni a csomaghoz tartozó programok dokumentálásában, és nem from scratch kell kézikönyvoldalt írni hozzá.

A `menu.ex` fájl: Példa menübejegyzés, hogy ne from scratch kelljen megírni, ha használni akarjuk a `dh_installmenu` debhelpert.

A `watch.ex` fájl: Az upstream programverzió elérhetőségét ha egy watch fájlban leírjuk,

akkor az `uscan` és `uupdate` parancsokkal könnyebben követni tudjuk az upstream programverziót.

A `ex.doc-base` fájl: Ha doc-base néven csinálunk ilyen fájlt, akkor a doc-base rendszerbe beépül a programunk dokumentációja. A `dh_installdoc` debhelpert kell használjuk hozzá, viszont a `dh_installdoc` debhelper sok más feladatot is elvégez.

A `postinst.ex`, `preinst.ex`, `postrm.ex` és `prerm.ex` fájlok: A `dh_installdeb` debhelper fogja a helyére tenni ezeket. Példák arra nézve, hogy mit tegyen a csomag a település/eltávolítás előtt/után.

2.5.7. A végkifejlet

A csomag megépítése

```
[pasztor@clyde ~/fdist-0.2]$ dpkg-buildpackage -rfakeroot -k69A2F424
dpkg-buildpackage: source package is fdist
dpkg-buildpackage: source version is 0.2
dpkg-buildpackage: source maintainer is PASZTOR Gyorgy <pasztor@fsn.hu>
dpkg-buildpackage: host architecture is i386
 fakeroot debian/rules clean
dh_testdir
dh_testroot
rm -f build-stamp
...
...
...
dh_md5sums
dh_builddeb
dpkg-deb: building package ‘fdist’ in ‘../fdist_0.2_i386.deb’.
 signfile fdist_0.2.dsc

You need a passphrase to unlock the secret key for
user: "PASZTOR Gyorgy (fsn) <pasztor@fsn.hu>"
1024-bit DSA key, ID 69A2F424, created 2001-12-28


 dpkg-genchanges
dpkg-genchanges: including full source code in upload
 signfile fdist_0.2_i386.changes

You need a passphrase to unlock the secret key for
user: "PASZTOR Gyorgy (fsn) <pasztor@fsn.hu>"
1024-bit DSA key, ID 69A2F424, created 2001-12-28


dpkg-buildpackage: full upload; Debian-native package (full source is included)
[pasztor@clyde ~/fdist-0.2]$ cd ..
```

```
[pasztor@clyde ~]$ lintian -i fdist_0.2.dsc
[pasztor@clyde ~]$ lintian -i fdist_0.2_i386.changes
E: fdist: binary-without-manpage fdist
N:
N:   Each binary in /usr/bin, /usr/sbin, /bin, /sbin, or /usr/games, must
N:   have a manual page.
N:
N:   Note, that though the ‘man’ program has the capability to check for
N:   several program names in the NAMES section, each of these programs
N:   must have its own manual page (a symbolic link to the appropriate
N:   manual page is sufficient) because other manual page viewers such as
N:   xman or tkman don’t support this.
N:
N:   Refer to Policy Manual, section 13.1 for details.
N:
```

2.6. debhelperek használata

Mint azt láttuk a `dh_make` által „inicializált” csomag debhelper-függő lesz. Nem véletlen, hogy az újabb fejlesztőeszközökben igyekeznek az egyre újabb technikákat beszivárogtatni a fejlesztők kezébe, ugyanis a debhelperek, valóban hasznos segítséget nyújtanak, és sokkal átláthatóbbá teszik a `debian/rules` fájl elkészítését is. Ezért tekintsük át a főbb debhelperek múködését:

dh_testdir Ellenőrzi a (forrás)könyvtárat a csomag építése előtt. Meggyőződik arról, hogy egy szabályos debian forráscsomagban vagyunk, létezik `debian/control`, stb. Valamint argumentumban további ellenőrzendő fájlokat is megadhatunk. Ha nem lenne valami jó, akkor hibakóddal lép ki, és mivel a `debian/rules` egy *Makefile*, ezért nem fog továbbmenni, hanem itt megáll.

dh_testroot Meggyőződik arról, hogy a csomagot valóban root-ként kezdtük fordítani. (Természetesen ha a **fakeroot**al indítottuk, az is megfelel.) Ha nem, akkor hibakóddal kilép.

dh_clean Ez a debhelper felelős azért, hogy egy csomag építése után kitisztuljon a munkakönyvtár. Eltávolítja a csomagok build-könyvtárát, és más fájlokat is letöröl pl. `debian/substvars`, `debian/files`, `*~`, `*.orig`, stb. Részletekért lásd a kézikönyv oldalát.

Azonban a `-k` opcióját kiemelném, mert ha ezt megadjuk neki akkor nem törli a `debian/files`-t, aminek akkor lehet jelentősége, ha több mint egy bináris csomag épül a forráscsomagunkból.

dh_installdirs Ez a debhelper felelős azért, hogy az alkönyvtárakat elkészítse a csomag build-könyvtáraiban. Minden a paraméterében megadott könyvtárnevet a control rekordban található első csomag build-könyvtárában hoz létre, illetve ha használjuk a `-p`, `-i`, `-a` kapcsolók valamelyikét, akkor az annak megfelelő első csomag build-könyvtárában. A `debian/`*`csomag`*`.dirs` fájlban tudjuk megadni a létrehozandó könyvtárakat. A könyvtárneveket könnyű-szóközzel kell elválasztani[24].

dh_installdocs A csomag build-könyvtáraiba feltelepíti a dokumentációt.

A csomag `usr/share/doc/`*`csomagnév`* könyvtárába feltelepíti a dokumentációt. Automatikusan ideteszi a `debian/copyright` fájlt. Ha több bináris csomag készül, és más-más copyright fájlokat kell hozzájuk telepíteni, akkor használhatjuk a `debian/`*`csomag`*`.copyright` fájlneveket.

Hasonlóan járhatunk el a `README.Debian` fájlok illetve a `debian/TODO` fájlok esetén is. Azt azonban tudnunk kell, hogy a célhelyen a `TODO` fájlból `TODO.Debian` lesz, hogy az esetleges eredeti (upstream) szoftvercsomag `TODO` fájlját ne írja felül.

A `debian/`*`csomag`*`.docs` fájlban további feltelepítendő dokumentációt sorolhatunk fel.

A csomag generál automatikusan parancsokat is a `postinst` és `prerm` scriptekbe, hogy a `/usr/doc/`*`csomag`* szimbolikus kötés automatikusan meglegyen, illetve eltávolodjon a `/usr/share/doc/`*`csomag`* könyvtárra.

[24] Ebből következően a létrehozandó könyvtár nevében nem lehet szóköz

Ha léteznek `debian/`*`csomag`*`.doc-base` fájlok, akkor a *doc-base* vezérlő fájlok is feltelepülnek, illetve a csomag `postinst` és `prerm` scriptjeibe bekerülnek a megfelelő parancsok, hogy ezek a dokumentációk a csomag telepítése folyamán a debian *doc-base* rendszerébe bekerüljenek.

Ha egy csomag több dokumentációt is szeretne beregisztrálni a *doc-base* rendszerbe, akkor a `debian/`*`csomag`*`.doc-base.*` minta használandó a fájlok neveire.

dh_installexamples Ez a debhelper fájlokat telepít a `usr/share/doc/`*`csomag`*`/examples` könyvtárba. Minden fájlnév, amit a `debian/examples` illetve `debian/`*`csomag`*`.examples` fájlokban felsorultunk feltelepít a fent nevezett könyvtárba.

dh_installmenu Ez a debhelper program felelős azért, hogy a debian menü csomaghoz tartozó fájlok a csomag build-könyvtárába bekerüljenek. Automatikusan generál `postinst` és `postrm` scriptrészleteket, hogy a debian menü csomag interfészén keresztül a debian menübe beregisztrálja a program saját magát. A csomaghoz tartozó menüadatokat a `debian/`*`csomag`*`.menu` nevű fájlba kell elhelyezni, és a `usr/lib/menu/`*`csomag`* néven kerül be a build-könyvtárba[25]. Lásd a menufile kézikönyv oldalt a tartalom formátumával kapcsolatban, illetve a debian-menu policy-t[9].

dh_installman A csomag build-könyvtáraiba feltelepíti a man oldalakat. Megadható, hogy mely kézikönyvoldalakat kell feltelepítenie. A telepítéskor figyelembe veszi a szekciót is a fájl kiterjesztésében található szám alapján, illetve a fájlban található `.TH` sor alapján. A kézikönyv oldal nyelvének megállapítása szintén támogatott, amennyiben valamilyen `.`*`ll`*`.n` vagy `.`*`ll_LL`*`.n` kiterjesztése van a kézikönyvoldalnak.

Ha a `dh_installman` úgy tünik, hogy rosz helyre telepíti a kézikönyvoldalt, akkor az azért van, mert a fájlban hibás a `.TH`-t tartalmazó sor. Ebben az esetben javítsuk ki a kézikönyvoldalt.

dh_undocumented Ez a debhelper a dokumentálatlan man oldalakat jelzi olymódon, hogy a megjelölt néven és szekcióban létrehozza a kézikönyvlapot, pontosabban egy az undocumented kézikönyvlapra mutató szimbolikus kötést. A létrehozandó kézikönyvoldalak nevét a `debian/`*`csomagnév`*`.undocumented` fájlban kell felsorolni, illetve argumentumban is megadható a szokásos módon, és a szokásos `-p`, `-i` ill. `-a` kapcsoló felhasználásával.

Ez a módszer erősen ellenjavallott a policy szerint. A javasolt eljárás egy debian csomag készítésénél, a megadott példafájl alapján elkészíteni a program leírását.

dh_installinfo Ez a debhelper felelős azért, hogy a programunkhoz tartozó info oldalak feltelepüljenek a csomagba, valamint hogy azok a csomag telepítése során regisztrálódjanak az info adatbázisába.

A szokásos módon lehet paraméterben illetve `debian/`*`csomagnév`*`.info` módon is a tudtára adni, hogy milyen info dokumentumokat kell feltelepíteni a build-könyvtárakba. Valamint a szokásos módon lehet használni a `-p`, `-i` és `-a` kapcsolókat.

A fentiekből következik az is, hogy a `prerm` ill. `postinst` scriptekbe való regisztrációval kapcsolatos részleteket ő is beleteszi.

[25]Ha pedig a `debian/`*`csomag`*`.menu-method` nevű fájl létezik, akkor az a `etc/menu-methods/`*`csomag`* fájlba kerül a build-könyvtárba.

dh_installchangelogs Függően attól, hogy a csomagunk natív debian csomag-e feltelepíti a `debian/changelog` fájlt a build-könyvtár `usr/share/doc/`*`csomagnév`*`/changelog` név alá. Illetve ha nem egy natív debiancsomagról van szó, akkor a `usr/share/doc/`*`csomagnév`*`/changelog.Debian` fájlnév alá. Természetesen, ha létezik `debian/`*`csomagnév`*`.changelog`, akkor inkább azt telepíti.

Ha nem egy natív debiancsomagról van szó, akkor megadhatjuk paraméterben az upstream changelog fájljának a nevét, amit a build-könyvtárban `usr/share/doc/`*`csomagnév`*`/changelog` néven helyez el. Figyeljük meg, hogy ebben az esetben ez nem ütközik, az ugyanabban a könyvtárban levő `changelog.Debian`-al, hanem ott lesz mind a két changelog.

Ez a debhelper arra is figyel (a kiterjesztés alapján), hogyha az eredeti changelog egy html fájl, akkor azt a html2text-el egyszerű szövegre konvertálja előbb, és azt helyezi el `changelog` néven az említett könyvtárba.

Hatása módosítható a `-k` kapcsolóval, amikoris megtartja az eredeti changelog fájl nevét és nem nevezi át `changelog`-ra. Hasznos lehet, ha az eredeti csomag changelogjának valami szokatlan neve van, vagy a csomag hivatkozik ilyen néven a changelog fájljára.

dh_link A csomag build könyvtáraiban szimbolikus kötéseket csinál. A szokásos módon a `debian/`*`csomagnév`*`.links` néven elhelyezünk egy fájlt, amelynek minden sorában két fájlnevet sorolunk fel: Az eredeti fájlét, illetve a hozzátartozó szimbolikus kötés nevét.

Mindig a teljes nevét adjuk meg a fájloknak, a `dh_link` gondoskodik arról, hogy a szimbolikus kötés a debian policynek [9] megfelelően jöjjön létre.

Az újabb verziókban már arra is képes, hogy a nem általa létrehozott szimbolikus kötéseket is leelenőrizze, és a policy-nek nem megfelelőket kijavítsa.

dh_strip Ez a debhelper felelős azért, hogy strip-eljen minden futtatható programot, osztott és statikus könyvtárat (library), amelyek nem hibakeresésre használatosak. Feltételezi, hogy az olyan fájlok, amelyeknek a neve a `lib*_g.a`-hoz hasonló, azok debugra használatos statikus könyvtárak, és nem strip-eli őket.

Ha a `DEB_BUILD_OPTIONS` környezeti változó tartalmazza a nostrip szót, akkor a Debian policy-nek megfelelően semmit sem fog strip-elni.

dh_compress Ez a debhelper azért felelős, hogy minden olyan fájl, amelyre a policy előírja, hogy tömörítettnek kell lennie, össze legyen tömörítve, illetve hogy az ilyen fájlokra mutató szimbolikus kötések javítva legyenek az új tömörített fájlokra.

Alapértelmezésben a Debian Policy[9]-nek megfelelő fájlokat tömöríti össze. Nevezetesen minden fájlt a `usr/share/info`, `usr/share/man`, `usr/X11R6/man` könyvtárakban, valamint a `usr/share/doc` alatt levő minden 4 kilobájtnál nagyobb fájlt (kivéve a `copyright` fájlt, `.html` fájlokat, valamint azokat amelyek a nevük alapján már tömörítettnek tűnnek), és minden `changelog` fájlt.

A `debian/`*`csomagnév`*`.compress` fájl ezesetben feltételezett, hogy egy shellscript, amelynek az outputja szolgáltatja az összetömörítendő csomagokat. A shellscriptet a csomag build-könyvtárában futattja le a rendszer. Ha bizonyos fájlok összetömörítését el akarjuk kerülni, akkor inkább a `-X` kapcsolót használjuk, mint ezt a lehetőséget.

dh_fixperms Kijavítja a fájlok és könyvtárak jogosultságait egy a Debian Policynek megfelelő változatra. Így a `usr/share/doc` alatt levő fájlokat 644 módúra javítja, kivéve az examples könyvtárban levőkét. Kijavítja a man oldalakat adó fájlok módját is 644-re. Minden fájlt a root tulajdonába vesz és a csoporttól illetve az egyéb felhasználóktól elvesz minden írási jogosultságot. Az osztott könyvtárakról leveszi a futtathatási jogot, ha valamelyikre véletlenül be lett volna állítva. A `bin/` és a `etc/init.d` könyvtárakban levő fájlokra beállítja a futtathatósági bitet. Végül eltávolít minden setuid illetve setgid bitet a csomagban levő összes fájlról[26].

dh_installdeb A csomag `DEBIAN` könyvtárába való fájlokat előkészíti, és a helyükre teszi, a megfelelő jogosultságokkal.

Az alábbi fájlokat teszi a `debian/` könyvtárból a `DEBIAN` könyvtárba:

- *csomagnév*.postinst
- *csomagnév*.preinst
- *csomagnév*.postrm
- *csomagnév*.prerm
- *csomagnév*.shlibs
- *csomagnév*.conffiles

Az újabb verziók esetében már nem kell külön feljegyezni a `etc/` könyvtárban levő fájlokat, azok automatikusan a `conffiles`ba számítanak.

dh_gencontrol Ez a debhelper generálja és helyezi el a `DEBIAN` könyvtárakba a `control` fájlokat.

dh_md5sums Legenerálja a `DEBIAN/md5sums` fájlt (illetve fájlokat több bináris csomag esetén), amelyek a csomagban levő fájlok md5 kivonatai. Természetesen minden a `DEBIAN` könyvtárban levő fájl kimarad a listázásból. Az md5sum fájl, a megfelelő tulajdonossal, és jogosultságokkal jön létre.

Alapértelmezésben a konfigurációs fájlok md5-kivonatai kimaradnak ebből a listázásból, de ha a `-x` opciót megadjuk, akkor azok is belekerülnek. Megjegyzendő, hogy ez az információ redundáns, mert ezek már máshol is eltárolódnak egy debiancsomagban.

dh_builddeb Egyszerűen meghívja a `dpkg`-t, hogy készítse el a debian csomagot, vagy csomagokat.

A sokat emlegetett általános kapcsolóknál általában argumentumban megadhatók a telepítendő/regisztrálandó menüelemek/manoldalak/infóoldalak, stb. Viszont könnyen előfordulhat, hogy a forráscsomagból több bináris csomag generálódik, és más más csomagokba másnak kell kerülnie, illetve jó lenne az átfedést elkerülni. Ezek szabályozására a `-a` kapcsoló hatásával azt érhetjük el, hogy az összes architektúrafüggő csomagunkra hatni fog a szóban forgó debhelper. A `-i` kapcsolóval az architektúrafüggetlen csomagukra tudunk hatni, illetve a `-p`*csomagnév* kapcsolóval a megadott csomagra tudunk hatni. Az általános debhelper paraméterezések a **debhelper(1)** kézikönyvoldalban találhatók.

[26] Vagyis ha valaki mégis setuid-es vagy setgid-es fájlt akar egy csomagban elhelyezni, akkor azt a `dh_fixperms` után kézileg be kell a megfelelő fájlokra állítani.

2.7. Mire figyeljünk a csomagolásnál, a csomag elkészítése, felépítése, ellenőrzése (lintian)

Ábrák jegyzéke

Példák jegyzéke

Irodalomjegyzék

[1] Vi IMproved tutorial

[2] CVSBook, http://cvsbook.red-bean.com/

[3] GCC kézikönyv

[4] GNU MAKE info oldalak

[5] GNU M4 info oldalak

[6] GNU AUTOCONF info oldalak

[7] GNU AUTOMAKE info oldalak

[8] debian **maint-guide** csomag, Josip Rodin

[9] debian **debian-policy** csomag, Julian Gilbey, Manoj Srivastava, továbbá: Ian Jackson, Christian Schwarz, David A. Morris

[10] debian **doc-base** csomag, Christian Schwarz, Adam Di Carlo

[11] debian **developers-reference** csomag, Adam Di Carlo, Christian Schwarz valamint Ian Jackson