Make

avagy a deklaratív fejfájás

Bodor András

Tartalomjegyzék

1. Jelölések

A dokumentum során a következő jelölésekkel élünk:

Ezzel a füllel figyelmeztetünk gyakran/könnyen elkövethető hibákra.

A fontos fülek vagy olyan fontosnak ítélt tanulságokat tartalmaznak, amelyek megértése kulcsfontosságú, vagy felettébb szükséges lépéseket emelnek ki.

Ilyen fülek egyszerű észrevételeket vagy nem olyan fontos tartalmi kiemeléseket tartalmaznak.

Ezek a fülek tanácsokat tartalmaznak, melyeket érdemes betartani magunk és mások mentális épségének megtartásásért.

Ezek a fülek trükkös, és/vagy nem nyilvánvaló viselkedésre hívják fel a figyelmet. Érdemes átgondolni és megérteni, hogy miért így viselkedik make.

1.1. UNIX kézikönyvek

Hivatkozásokkor van, hogy a UNIX-féle man-page formátumként hivatkozunk egy-egy programra, például `make(1)`. Ez az 1-es kötetből származó `make` névvel rendelkező kézikönyvre (manual) utal. Aki UNIX vagy ahhoz eléggé hasonló operációs rendszerrel dolgozik, ki tudja próbálni ezek olvasását a `man(1)` paranccsal:

Listázás 1. UNIX man parancs

```
$ man man ①
$ man printf ②
$ man 3 printf ③
```

① A `man(1)` hez tartózó kézikönyvet keresi fel. Érdemes elolvasni olyanoknak, akik sokat dolgoznak UNIX vagy hasonló rendszereken.

② Felkeresi az első `printf` névvel rendelkező kézikönyvet, amit az első kötetben talál meg: ez a parancssori printf dokumentációja.

③ A felhasználó által megadott kötetben keresi a `printf` kézikönyvet amit meg is talál: a C printf függvénycsalád dokumentációja.

Mivel a szekciószámok és a kézikönyv nevek rendszerfüggőek, nem biztos, hogy az itt jegyzett referenciák értelmes kézikönyvre vezetnek az olvasó rendszerén.

2. Miért make?

Make a 70-es évek válasza az egyre növekvő szoftver projektek karbantartására és kezelésére. Mégis, majdnem 50 évvel megálmodása után, nagyságrendekkel nagyobb és komplexebb projektekkel, még ma is egy elterjedt projekt menedzsment rendszer olyannyira, hogy egyes operációs rendszerek (FreeBSD) teljes csomag kezelő rendszere komplex makefileok hálózatából áll, mely bármely csomagot és annak függőségeit képes a helyi gépen elkészíteni egyetlen parancs kiadásával. Emellett a GNU projekt által használt AutoTools eszközök is make köré biztosítanak platform-függetlenséget biztosító szkripteket. [1]

Mit is tud tehát a make, ami a mai napig létjogosultságot adott neki ennyire különböző méretű és feladatkörű felhasználásához? Többek között flexibilitása, és az hogy a karbantartási feladatot úgy nevezett deklaratív oldalról közelítette meg.

2.1. Deklaratív programozás

Szemben az általában elterjedt imperatív programozási nyelvekkel, makeben nem azt kell megmondani, hogy pontosan, *hogyan* is kapjuk meg az elkészítendő fájlokból álló topologikusan rendezett gráfot, csak a fájlok közötti kapcsolatokat ismertetjük make-el, az pedig szépen megcsinálja az összes lépést, amire szükség van, hogy eljussunk az igényelt fájlig. Ezt az erejét az úgy nevezett deklaratív programozás paradigmájának köszönheti. Bár ez a paradigma egy teljes programozástani ágazat—mint az objektum orientált vagy funkcionális programozás—így mély tárgyalása ebben a make használatára egy gyakorlatias rálátást biztosító dokumentumban nem fér a keretekbe, mégis egy rövid leírással nagy vonalakban ismertetjük.

Ha gondolunk egy C programra mit látunk: utasításokat, melyek a C absztrakt gépen végrehajtódnak sorról sorra. (A szálaktól most eltekintünk, bár nem zavarják meg a modellt.) Az egyik lépésben leírtak a másik lépésben tovább fel lesznek használva és így, lépésről lépésre, jutunk el a program végén a kívánt célig. Ez az úgy nevezett imperatív programozási paradigma: mindent parancsok sorozataként értelmezünk. A végrehajtó gép pedig gondolkodás nélkül hajtja végre ezeket a parancsokat melyeket az általa érthető egyszerű nyelvre mi programozók lefordítottunk.

Ezzel állítható szembe a deklaratív programozás. Ebben az esetben nem azt próbáljuk megmondani a gépnek, hogy *hogy* csináljon dolgokat, amikből mi megkapjuk amit szeretnénk, hanem egyenesen azt mondjuk neki, hogy *miből* szeretnék és *mit* kapni, a többit megoldja ő.

Példa 1. HTML

A legelterjedtebb példa, bár programozási nyelv volta megkérdőjelezhető, a HTML. A HTML azt írja le, hogy *mi* jelenjen meg a weboldalon, viszont, hogy a kirajzoláshoz mely algoritmusok szükségesek, vagy milyen adatszerkezetekre van szükség teljesen kikerültek a nyelvből—érthető okok miatt. Ha valaki egy weboldalt szeretne megszerkeszteni, jobban érdekli a weboldal tartalma, mint a különböző GPU gyorsított kirajzolási megoldások sora, amiket egy böngésző, a deklaratív nyelv fogyasztója, implementál.

Példa 2. SQL

Másik gyakori példa az SQL. Egy SQL lekérdezés nem mondja meg az adatbázisnak milyen sorrendben, vagy hogyan keressen a sorok között: bináris keresés valamilyen rendezett indexben, egy nagy hash-table, bináris fák, stb., csak azt mondja meg neki, hogy *milyen* feltételeknek megfelelő sorokat szeretne megkapni, és hogy *hogy* fogja azt az adatbázis-kezelő odaadni, azt teljesen rá bízza.

DSL, avagy domain specifikus nyelvek

A példákból látható, hogy a deklaratív programozás domain specifikus nyelvekben (DSL) nagy népszerűségnek örvend. Ezek a nyelvek pontosan arra törekednek, hogy az implementációt egy erősen absztrakt réteg mögé rejtsék: csakis a domain feladatait lássák el és az implementációs részletekkel nem foglalkoznak—jeleníts meg egy weboldalt, mindegy, hogy hogyan, de így nézzen ki a végeredmény. Make is tekinthető egy DSL-nek, projektek kezelésére.

A deklaratív programozás tehát egy magas szintű absztrakciót biztosít egy-egy probléma megoldására, ami nem a végrehajtandó elemi lépésekben gondolkodik, hanem kezdeti bemenetekben és az azokból kiszámolt eredmények kapcsolatainak leírásával.

A deklaratív programozás, mint gyűjtőfogalom

A deklaratív programozás egy nagy gyűjtőfogalomként is értelmezhető. Sok más elterjedt programozási paradigma tekinthető a deklaratív paradigma egy alfajának: például a funkcionális, amely a függvényhívások kompozíciójából épít programokat, de például a függvényhívások szekvenciájáról már nem sokat szól, ellenben az imperatív procedurális programozással, ahol szinte minden kontextusban pontosan értelmezhető a függvények sorrendje.

A másik talán ismertebb alfaj a logikai alapú programozás: Prolog. Ezek a nyelvek a logikai igazság bizonyításával foglalkoznak: a programozó felsorol igaz állításokat, majd azokból komplex lekérdezéseket állít össze. Az értelmező megállapítja, milyen bemenetre lenne igaz az állítás, vagy hogy egyáltalán létezik-e megoldás.

2.2. Make és a hibrid megoldás

Bár látszott az előző fejezetből, hogy a deklaratív programozás egy kedvező helyzettel kecsegtet a programozási színtéren, mégis van egy apró probléma, ami a mai napig megakadályozta az általános nyelvekben elterjedését, míg az imperatív nyelvek családja folyamatosan újabb és újabb tagokkal bővül. Minél “deklaratívabb” egy nyelv, annál komplikáltabb értelmezőre van szüksége. Ha belegondolunk, ez logikusan következik a deklaratív programozás tulajdonságaiból: ha valamit csak a bemenetek és kimenetek kapcsolatából építünk fel, akkor az adott értelmezőnek tudnia kell az általános kapcsolatot kezelni a bemenetek típusai és kimenetek típusai között. Egyszerű gondolatnak tűnik a megoldás, hogy akkor csak megengedjük a programozónak, hogy definiálja az értelmező számára ezt a kapcsolatot. Viszont így eljutunk egy olyan helyzetbe, hogy, haladva lefele az absztrakciókban, valahol meg kell törni az absztrakciót teljesen, mert a deklaratív környezet magas szintű tiszta absztrakciója nem képes biztosítani akkora flexibilitást, hogy a való világ minden tökéletlenségét tudja kezelni. Ha a funkcionális nyelveket nézzük, melyek tiszta (pure) függvényekben gondolkoznak, akár meg is tiltják a mellékhatásokat (side-effect). Mégis szükségük

van egy pontra ahol kötelező megtörni a matematikailag tökéletes világukat: kommunikálni kell a külvilággal. Akármit csinálhat egy matematikai normáknak teljesen megfelelő program, ha nem tud bemenetet fogadni, vagy az eredményét velünk közölni, ekvivalens egy üres programmal, ami semmit sem csinál, vagy ér.

Ebbe a problémába ütközünk, ha egy projekt felépítő rendszert (ezentúl build system, egy olyan program, ami más programokat épít fel annak forrásfájlainak kontextusban helyes kezelésével) szeretnénk megtervezni. Akár milyen mélyre húzzuk az absztrakciót, mindig adódni fog egy pont ahol ezt el kell dobni. A make egy nagyon egyszerű válasszal ált elő erre a problémára: az absztrakciója csakis a fájlok közötti kapcsolatokra terjed ki, bármi amit csinál az ember a fájlokkal, azt már nem fedi le.

Ezt nevezik egy hibrid programozási nyelvnek; egy részből deklaratív, mert a fájlok relációit csak deklaratívan írjuk le, viszont magukat az utasításokat, hogy mit csinálunk a fájlokkal, már imperatívan mondjuk el makenek. Pontosan ebből a megoldásból ered makenek az előbbiekben már említett flexibilitása. Bár hivatalosan C (és—ugyanazon build modellel rendelkezvén—C++) projektek kezelésére lett kitalálva, azzal hogy tetszőleges lépéseket hajthatunk végre egy-egy fájl reláció során közel minden szituációban használható. Ennek a PDF-nek elkészítését is make[2] kezeli, illetve egy nagyon kezdetleges, makere erősen hajazó `Makey` nevű implementáció Perlben. (Ennek miértjére kitérünk a későbbiekben.)

Ugyan ez teszi lehetővé, hogy az első fejezetben említett FreeBSD forrásalapú csomagkezelő rendszerét is tisztán makeben implementálják. Még akkor is, ha egy-egy forrás csomagot nem make-el lehet létrehozni, make teljesen alkalmas más rendszerek meghívására. Lehet a projekt natív build systemje CMake, Ninja, Maven, stb., a make alapú rendszer tudja kezelni.

Most, hogy ismerjük miért van használatban a mai napig minimális változtatásokkal egy közel 50 éves program, kezdjünk is bele, hogy mi, mint a XXI. század C és C++ programozói, miként alkalmazhatjuk a múlt eszköztárát.

[1] A GNU AutoTools és egyéb konfigurációs rendszerekről a WIP fejezetben lesz szó.

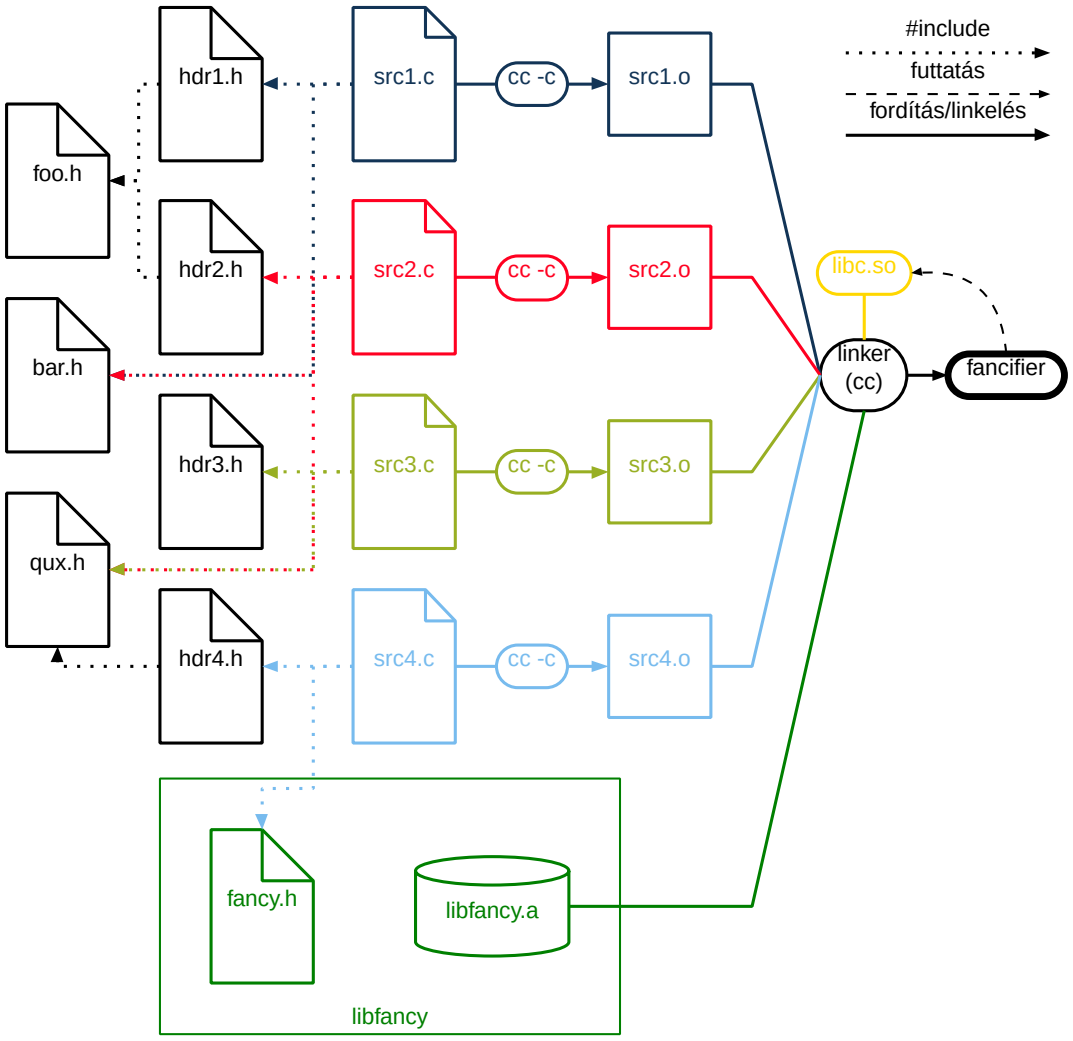
[2] A mindenkori legfrissebb Makefile az érdeklődőknek megtekinthető itt, bár az igazi logika főleg a build.pl szkriptben van.

3. A minimális makefile

A következő fejezetek felételeznek ismereteket a C és C++ build modellről. Bár ezek részletes tárgyalását, erős platform és fordító függése miatt, a dokumentum elhagyja, az A C build modell függelék az értelmezéshez szükséges információkat tartalmazza.

A következő fejezetekben egy egyszerű C11 projektet fogunk elkészíteni, a `fancifier` programot. Azt most feltesszük, hogy a C fájlok helyesek és a program, amit leírnak jól működik. A környezet egy UNIX szerű rendszer, a fordító feltételezve pedig vagy GCC, vagy Clang; aki Windowson szeretne követni a saját projektjével annak a MinGW rendszert lesz szükséges használni, a hozzá tartozó parancssorral.[3] A fájlok amiket le szeretnénk fordítani a `src1.c`-től kezdve `src4.c`-ig terjednek, mindegyikhez tartozik egy `hdrN.h` fájl, és közösen használják még a `foo.h`, `bar.h` és `qux.h` fájlokat. Mindemellett egy külső libfancy könyvtárat, ami a `ext/include` mappan tárolja a fejlécfájljait és a `libfancy.a` fájl pedig az `ext/lib` mappában található.

Az alábbi A példa projekt buildje diagram vizuálisan mutatja, hogy milyen lépésekből fordítható a projekt.



Ábra 1. A példa projekt buildje

A `make(1)` program a `makefile`, vagy `Makefile` nevű fájlokat olvassa be a jelenlegi futtatási mappából, és az azokban leírtakat hajtja végre. A különbség a két fájl között, hogy ha a kisbetűs létezik, akkor azt

olvassa be make, és a nagybetűset meg se nézi. Ennek ellenére, általánosan a nagybetűs írásmód terjedt el.

Készítsünk tehát egy `Makefile` nevű fájlt.

3.1. Egy forrás fordítása

Első lépésként egy fájlt szeretnénk lefordítani. Ehhez make deklaratív tulajdonságainak alapegysége nyújt segítséget, a szabály (rule). Mint deklaratív nyelv, elsődlegesen azt kell megmondani makenek, hogy milyen fájlokból lesznek más fájlok. Ezt tehetjük meg egy szabály definiálásával.

Listázás 2. Általános make szabály

```
cél: előfeltételek...
    recept
```

Minden szabályhoz tartozik egy cél (target) fájl—ez az amit a szabály előállít—és nulla vagy több előfeltétel (prerequisite), ezeknek a fájloknak kell léteznie, hogy a megadott recept (recipe) el tudja készíteni a célt.

Könnyen eltéveszthető, de make történelmi okokból csak tabulátor karaktert fogad el a recept indentálására. Aki GNU maket használ az találkozhat a rendkívül beszédes `Makefile:N: *** missing separator. Stop.` hibával. Ilyenkor az *N*. sorban tabulátor helyett valami mást talált, például valamennyi space karaktert. Érdemes az fejlesztőkörnyezetünket úgy beállítani, hogy Makefileok esetén ne végezzen tabulátor kibontást. CLion ezt automatikusan tudja, vim-nél az `autocmd FileType make set noexpandtab` parancs oldja meg a problémát. Egyéb szövegszerkesztőknél keresd a megfelelő dokumentációt.

Make okosan el tudja dönteni, mikor van szükség egy cél elkészítésére: ha legalább egy előfeltétele későbbi módosítási dátummal rendelkezik a célnál, vagy olyan cél amit most hajtottunk végre, akkor végrehajtja a receptet. Egy szabály akkor kerül végrehajtásra, ha, vagy a felhasználó kérte, hogy az a cél készüljön el, vagy annak a célnak, amit a felhasználó kért, valahányadik függősége az előfeltételláncban.

Fontos, hogy lássuk, hogy make fájlokban gondolkozik. Minden amit megadunk célként vagy előfeltételként, az vagy már maga is egy létező fájl, vagy egy másik szabály célja, így egy szintén egy fájl, csak még nem létezik. Ez alól létezik egy kivétel. Az úgy nevezett hamis (phony) szabályok céljai nem fájlok, hanem sokkal inkább csak parancsok makenek. Ugyan úgy kezeli őket make, mint egy rendes szabályt, viszont elvárja, hogy ne készítsék el céljukat. Így tehát parancssorból hívott make esetén lehet hamis szabályra hivatkozni, hiszen létezik és célja is van, viszont sosem fog elkészülni a cél, és ezt make is tudja. Hamis szabályt úgy készíthetünk, hogy a `.PHONY` speciális cél előfeltételeként listázzuk a célját.

Attól, hogy make feltételezi, hogy hamis szabályok nem készítik el céljukat, lehetséges hogy ez mégis megtörténik. Ilyenkor a cél késznek fog számítani, függetlenül, hogy hamis szabály, és nem fog lefutni.

Példa 3. Hamis szabály

A két leggyakrabban található cél az `all` (esetleg `build`) és a `clean`. Ők a nevükhöz illő funkciót látnak el: `all` minden szabályt elkészít—ha azok nem frissek, természetesen; `clean` pedig az összes olyan fájlt törli, melyek szabályokból jöttek létre. Nyilván ezeket manuálisan kell implementálni, ha szeretnénk őket a projektünkbe.

```
.PHONY: all clean ①

all: célok amiket szeretnél elkészíteni
    echo "All built :)" ②

# ...

clean:
    rm -f takarításra ítélt fájlok sora
```

① Egy `.PHONY` deklarációval lehet az összes hamis szabályt ismertetni make-el, de lehet őket egyesével is, minden szabálynál.

② Mivel az előfeltételek kerülnek hamarabb elkészítésre, mint a szabály, aminek előfeltételei—ahogy a névből is adódik—így az üzenet a build legvégén kerül végrehajtásra.

Olyan szabályoknál ahol nincs előfeltétel, a célok frissessége nem ellenőrizhető, így azok mindig újra el lesznek készítve, ha szükség van rájuk.

Ezt felhasználva írhatjuk fel a `src1.c` fordításához szükséges szabályt. Először meg kell alkotnunk a parancsot amivel szeretnénk fordítani: A fordításhoz szeretnénk az általános warning flageket (`-Wall -Wextra -Wpedantic`) használni, illetve a szükség van még C11 szabványra (`-std=c11`) és a libfancy könyvtár fejlécfájljaira (`-Iext/include`). Az átlagos UNIX szerű fordítókon a `cc(1)` C fordító fordításra a `-c` flaggel használható, a kimenetet pedig a `-o` kapcsoló állítja be. Ezeket az információkat felhasználva a `src1.c` fájl lefordítására a parancs a következő lesz: `cc -c -o src1.o -Wall -Wextra -Wpedantic -Iext/include -std=c11 src1.c`.

Miért nem használjuk a -Werror flaget

A `-Werror` flag (MSVC-n `/WX`) az a kapcsoló, ami utasítja a fordítót, hogy minden figyelmeztetést (warning), amit generál azt számítsa hibának és ne fordítsa le a forrásfájlt, amiben ilyet talál. Vannak emberek, akik azt vallják, hogy ezt minden esetben tartalmaznia kell a fordító parancsnak. Személyes vélemény szerint a következőt javasoljuk: a következőkben leírtak majd engedélyezni fogják a helyi fölülírását a fordítási kapcsolóknak, vagy annak kiegészítését. Ezt felhasználva mindenki a saját felhasználása közben új hívja meg maket, hogy a fordítás tartalmazza a `-Werror` flaget, de ez ne legyen belekódolva a makefileba. Ezzel magunknak elérjük azt, hogy biztosan nem fog a fordítónk figyelmeztetést generálni—hiszen le se fordulna a programunk ha ilyet csinálna—viszont másoknál, egy másik fordító, vagy akár csak egy másik verziója ugyanazon fordítónak, ami lehet generál egy olyan figyelmeztetést, amit a mienk nem, még akkor is le fogja fordítani a programunkat anélkül, hogy nekik bele kelljen piszkálnia a makefileba, vagy a forrásokba.

Most, hogy megvan a fordítási parancs, már össze tudjuk állítani a szabályt amivel a `src1.c` fájl le tudjuk fordítani a `src1.o` objektumfájlba. Megfontolandó még, a `src1.c` fájl által #includált fejlécek is előfeltételként jegyezzük, így azok módosítása is szükségessé teszi a `src1.o` fájl újrafordítását, erre később nézünk 1) szabványos de nem túl flexibilis, és 2) egy automatikus, bár nem szabványosított megoldást, egyelőre elhagyjuk.

Listázás 3. Egy teljes szabály a src1.c fájl lefordítása

```
src1.o: src1.c
    cc -c -o src1.o -Wall -Wextra -Wpendatic -Iext/include -std=c11 src1.c
```

Amint ezzel megvagyunk, már is szólíthatjuk maket, hogy fordítsa le nekünk a `src1.c` fájlt. Ha maket csak úgy, paraméter nélkül hívjuk meg, akkor az első szabályt futtatja le amit a makefileban talál.

Ezzel a tudásunkkal már szorgosan nekiállhatunk felsorolni az összes forrásfájlt amit le szeretnénk fordítani. Triviálisan látszik, hogy ez nem jó ötlet, főleg nem manuálisan.

3.1.1. Feladatok

Pár apró feladat, amivel lehet gyakorolni a fejezetben tárgyaltakat. Lehetséges megoldások a Megoldások függelékben találhatóak.

1. Készítsünk egy makefilet, ami a `greet.txt` fájlba kiírja, hogy “Hello World”. Miért nem szabványos kimenetre írunk?
2. Módosítsuk az első feladatban írt makefilet, hogy a `greet` parancsra elkészítse a `greet.txt` fájlt.
3. Miért nem listáztuk az összes szabályt előfeltételként a `clean` szabálynak a Hamis szabály példában? Egészítsük ki a makefileunkat egy `clean` céllal, ami kitörli a `greet.txt` fájlt.
4. Írjunk egy makefilet, ami egy (makefileba kódolt) C fájlról és a belőle készült objektumfájlból egy tar archívumot készít. (Modern Windows is tartalmaz egy tar.exe-t, így ott is fog működni.) A `tar(1)` parancs paraméterezése sokaknak sötét mágia, így nekik itt egy parancs, amit használhatnak: `tar cf tarfile.tar bemenetek⋯`.

3.2. Makrók

Első probléma, hogy ha jelenlegi állapotban felsoroljuk az összes fájlt, akkor gyakorlatilag belekódoltuk minden szabályba a jelenleg használatos fordítási kapcsolókat. Ha a jövőben szeretnénk ezeket módosítani, mondjuk bekapcsolni az optimalizációt (`-O2`), akkor azt minden fájlnál nekünk kell manuálisan odaírni—egy elég könnyen elrontható művelet. Erre a problémára a make által nyújtott megoldás a makrók, vagy változók[4]. Ezek létrehozása egyszerű:

Listázás 4. Makró definíció

```
CFLAGS = -Wall -Wextra -Wpedantic -Iext/include -std=c11
```

A `CFLAGS` az általánosan használatos makró a C fájlok fordítási flagjainek tárolására. C++ megfelelője a `CXXFLAGS`, de van, hogy tiszta C++ projekteknél szimplán `CFLAGS`-t használnak erre a célra is.

Az újonnan definiált `CFLAGS` makrónkat felhasználva egy jóval kevésbé repetitív módon felírhatjuk az előző szabályunkat.

Listázás 5. Makró használat

```
src1.o: src1.c
    cc -c -o src1.o $(CFLAGS) src1.c
```

Ebből látszik a szintaxisa a makrók használatának: `$`-el kell hivatkozni rájuk. Ilyenkor a `$` úgy működik, mintha egy dereferáló operátor volna, és lekérdezi a makróban található értéket. Viszont magától csak egy betűt dereferál: `$ALMA` csak az `A` makrót próbálja beilleszteni, amiben, ha mondjuk `pite` van akkor az előbbi kifejezésből `piteLMA` eredményt kapjuk. Ha több karakterből álló makrót szeretnénk kibontani (macro *expansion*), akkor zárójelekbe, vagy kapcsos zárójelekbe kell őket rakni: `$(ALMA)` immáron már az `ALMA` makró tartalmát fogja használni.

Lusta makró kibontás

A makrók kibontása csakis használatkor történik. Tekinthető a makrók kibontási módszere egyfajta lusta kiértékelésnek (lazy evaluation). Egy makró—és emiatt az értékbe ágyazott egyéb makrók—csak akkor kerülnek kibontásra, ha egy makró értékadás bal oldalán állnak, vagy ha egy szabály valamely részében szerepelnek. Ez nem feltétlen meglepő azoknak, akik dolgoztak nyelvekkel, amelyek vagy teljesen lusta kiértékeléssel dolgoznak, vagy egyáltalán támogatják ezt a lehetőséget. Ebben a dokumentumban viszont nem feltétlen ők a célközönség, illetve make legfőképpen a C és C++ világban elterjedt, ahol nem gyakran találkozunk ezzel a tulajdonsággal. Emiatt könnyen meglepő lehet a működés azoknak, akik most ismerkednek make-el.

```
.PHONY: print1 print2 printa
VAL = a ①
VALUE_WRAPPER = val: $(VAL), undefined: $(UNDEFINED) ②

UNDEFINED = no ③

print1:
    echo $(VALUE_WRAPPER)

printa:
    echo $(a) ⑤
$(VAL) = actually a ⑤

VAL = b ④
print2:
    echo $(VALUE_WRAPPER)
```

① Beállítjuk a `VAL` makró értékét az `a` sztringre. Semmi különleges.

② Felhasználjuk a `VAL` makró értékét, viszont hivatkozunk a még definiálatlan `UNDEFINED` makróra. A nem deklaratív gondolkodáshoz szokottaknak ettől vagy valami hibát várnának el, mint C-ben, vagy valamilyen érvénytelen érték beillesztését, mint mondjuk `undef` interpolálását, ahogy `no strict` Perlben történne. Mégis egyik se fog bekövetkezni. Ami itt fontos, hogy hiába vannak sorrendben a makefile utsításai, a forrásbeli helyük csupán más értékadások bal oldalát befolyásolják. Egyszerűen deklaratívan megállapítottuk, hogy mi `VALUE_WRAPPER` értéke. Úgy érdemes gondolni rá, hogy a deklarációnk azt mondja, hogy `VALUE_WRAPPER` tartalmaz két másik makrót, így tehát mindig is érvényben marad ez az állítás, függetlenül, hogy a külső makrók értéke változik.

③ Értéket adunk az `UNDEFINED` makrónak.

④ Módosítjuk a `VAL` makró értékét. Utóbbi két módosítás, indirekten ugyan, de befolyásolták `VALUE_WRAPPER` érékét is, hiszen az úgy van definiálva, hogy ezekre a makókra referál.

⑤ Az utasítások sorrendje, makrók érékadásakor, az eddigiekkel ellentétben, jelentős: ha egy makró értékét használjuk értékadás bal oldalán, akkor makenek szüksége van a makró kibontásra, így azt el kell neki azonnal végeznie, így a később felülírt `VAL` makró értéke addigra már ki lett bontva és létrejött az előző értékből az `a` nevű makró. Az látható viszont, hogy a szabályból ilyen esetben is tudunk hivatkozni, olyan makóra, ami “még nem létezik”.

Ezt értve, a makfile-t alkalmazva nem lepődünk meg, amikor a következő kimenetet kapjuk:

```
$ make print1 print2 printa
val: b , undefined: no
val: b , undefined: no
actually a
```

Mindig zárójelezzük be a makrókibontásokat, egy kivétellel, ahol a beépített egy karaketers makrókat használjuk: Beépített automatikus makrók. Ezen kívül ha definiálunk egybetűs makrókat is zárójelezve hivatkozzunk rájuk.

Általában elterjedt jelölés a zárójelek használata, a kapcsos zárójeleket egyedül a (Free)BSD kódolási stílus előírások említik[5]. Ennek megfelelően ez a dokumentum a zárójeles dereferálást használja, de ha BSD központú szoftveren dolgozunk érdemes a kapcsos zárójelet megfontolni.

3.2.1. Beépített makrók

Make rendelkezik beépített makrókkal. Ezek összessége egy adott rendszeren a `make -p` paranccsal tekinthető meg, több mással egyetemben, így általában ez egy elég hosszú lista. Ami jelenleg nekünk fontos az a `CC` makró. Ez tárolja a rendszeren jelenleg található C fordítót. Általában ez a `c99`, ellentétben azzal, ahogy eddig ezt feltételeztük, de ez nem feltétlen igaz, illetve ha egy rendszeren található több fordító is, akkor a make hívója a `CC` beállításával beállíthatja: `make CC=my-random-cc`. Utóbbi lehetőség miatt érdemes a fordítót makróval megadni, hiszen például ha GCC-vel és Clang-el is szeretnénk tesztelni programunkat, akkor jelentősen egyszerűbb csak a meghívásnál beállítani `CC` -t a megfelelő fordítóra. Ezen kívül elérhetőek még az egybetűs, *automatikus makrók*, amik receptenként mindig a jelenleg futó szabályról tartalmaznak információkat.

Táblázat 1. Beépített automatikus makrók

`@`	A jelenleg elkészítendő szabály célja
`?`	Azon előfeltételek, amelyek jelenleg nem számítanak frissnek
`<`	Az előfeltétel amiből jelenleg építkezünk. ld. Utótag szabályok
`*`	Előtagja (prefix) a fájlnak, utótagok (kiterjesztések) és elérési útvonal nélkül.
`%`	Az archívum (archive) tag neve. ld. Könyvtár szabályok
`!`	A archívum neve. ld. Könyvtár szabályok
`>`	(BSD make) az összes előfeltétele a célnak
`^`	(GNU make) az összes előfeltétele a célnak

A táblázatban szereplő `@`, `<`, és `*` makrónak létezik két-két speciális változata is: ezeket a makró nevének és `D` vagy `F` betűkkel való összefűzésével kaphatjuk. Ilyenkor a `D` betűs verziók a makró eredeti értékének elérési útvonalára, az `F` betűsek pedig pusztán a fájlnévre bontódnak ki.

Míg az egy betűből álló automatikus makrók használhatóak zárójelek nélkül, az elérési útvonalra és fájlnévre specializált verzióik már nem egy betűből állnak. Minden esetben figyeljünk a helyes kibontási szintaxisra. Például `$(@D)`.

A táblázat utolsó két sora két implementációból származó bővítmény. Szintaxisuk eltér, de pontosan ugyan azt a célt látják el: a jelenlegi cél összes előfeltételét tartalmazzák. Mivel nem szabványosak és a két jelentős implementációban nem átvihető a szintaxisuk, így nem tárgyajuk őket, de jó tudni, hogy mit jelentenek, mert sokan vannak, akik GNU make-en kívül más make támogatásra nem is gonolnak, és erőszeretettel használják a GNU `^` makrót.

Példa 4. A módosított beépített makrók

Vegyük a következő makefilet, és egy (akár üres) `test.in` fájlt ugyanabban a mappában.

Listázás 6. Makefile

```
.SUFFIXES: .in .out
.in.out: ①
    echo "@: $@ - $(@F) - $(@D)" > $@
    echo "*: $* - $(*F) - $(*D)" >> $@
    echo "<: $< - $(<F) - $(<D)" >> $@

_build/output.te.xt: test.out
    mkdir -p _build
    cat test.out > $@
    echo "@2: $@ - $(@F) - $(@D)" >> $@
    echo "*2: $* - $(*F) - $(*D)" >> $@
# nincs $< nem utótag szabályokban
```

① Ezek úgy nevezett utótag szabályok (suffix rules). Jelenleg működésük nem fontos, az Utótag szabályok fejezet tárgyalja őket részletesen.

Ezt lefuttatva megtekintetjük a különböző makrók kimeneteit a `_build/output.te.xt` fájlban. Ez az OpenBSD 7.0 gépemen a következő kimenetet készítette:

Listázás 7. _build/output.te.xt fájl tartalma

```
@: test.out - test.out - .
*: test - test - .
<: test.in - test.in - .
@2: _build/output.te.xt - output.te.xt - _build
*2: _build/output.te.xt - output.te.xt - _build ①
```

① A POSIX szabvány erre nem specifikál működést, sőt még azt sem írja elő, hogy kötelező léteznie *-nak, nem utótag szabályokban. Emiatt gyakorlatban ne használjuk, amúgy is, ahogy látható, ahol létezik sincs túl sok haszna.

Ezekből talán a legtöbbet használtak a `@` és `<` makrók, így a `@` tárgyalásával fogjuk kezdeni. Utóbbiról később, de rövidesen lesz szó. A `@` szimbólumot viszont már most beépíthetjük a receptünkbe, hiszen nagyon egyszerű a feladata: mindig a jelenleg fordítás alatt álló cél fájlt tartalmazza. Ezzel elérhetjük, hogy gyakorlatilag csak a forrásfájltól függ már csak a receptünk, minden mást automatikusan kitölt make. Jelenleg így néz ki a teljes makefile:

Listázás 8. Jelenleg makefile egy C fájlra

```
CFLAGS = -Wall -Wextra -Wpedantic -Iext/include -std=c11

src1.o: src1.c
    $(CC) -c -o $@ $(CFLAGS) src1.c
```

3.2.2. Makrókibontási módosító

Most, hogy már van egy nagyon jól működő parancsunk, ami egy teljes egész fájlt le tud fordítani, elkezdhetünk gondolkodni, hogy tudnánk a többi fájlt is feldolgozni, lehetőleg anélkül, hogy fel kellene sorolni minden fájlnak a szabályát. Erre a problémára a make által felajánlott megoldás a következő fejezet témája, de előbb a makróknak még egy tulajdonságát tárgyaljuk. Mégpedig azért, mert szükségünk lesz az összes objektumfájlra és ezek elkészítéséhez a forrásfájlokra.

Elsőként hozzunk létre egy makrót, amiben tároljuk a lefordítandó forrásokat. Ezt már könnyen meg tudjuk tenni eddigi ismereteinkkel:

Listázás 9. Forrásfájlok listázása

```
SRCS = src1.c src2.c src3.c src4.c
```

Ne `glob(7)`-oljunk

BSD és GNU make és támogat bővítményeket, amivel egy mappából az összes C fájlt össze tudjuk gyűjteni. GNU make függvényei között szerepel a `wildcard` függvény, és támogatja BSD make `!=` makró definiáló operátorát. Az utóbbi használatát mutatjuk be a `find(1)` parancs használatával.

Listázás 10. Automatikusan keresett források

```
SRCS != find . -type f -name '*.c' -maxdepth 1
```

Ez megkeresi a jelenlegi mappában az összes C fájlt, de nyilván minden hasonló implementációra igaz, csak ha egyszerűen valami glob műveletet hajtunk végre, mint egy `perl -E 'say <"*.c">'`, akkor is ugyan ez a helyzet adódik. Mi a probléma ezzel? Szimpla makenél nagyon egyszerűen a sebessége. Minden make meghíváskor kell egy mappabejárást csinálni, ami lehetséges, hogy annyira nem olcsó művelet, főleg, ha sok elemből álló projektet szeretnénk éppen elkészíteni. Még rosszabb a helyzet, ha valamiért rekurzív bejárást kell végezni—minden buildnél. Ha pedig valamilyen konfigurátort vagy generátort használunk, mint AutoTools vagy CMake, akkor nem make a felelős a fordítandó fájlokért, hanem a fölé épített rendszer. Ha ennek a rendszernek minden fordítás alkalmával végre kell hajtania, legalább a részleges konfigurációt, csak hogy biztosra mehessen, hogy az utoljára legenerált makefile a friss glob adatokkal dolgozik az elképesztően sok időveszteség. Ha még erre vesszük, hogy esetleg Windowson dolgozunk, ahol nem olyan olcsó a folyamatok indítása, akkor egy jelentős hátrányba vezettük saját magunkat, csupán azzal, hogy meg akartuk spórolni a gépelést.[6]

Az így már ismert fájljainkat szeretnénk párosítani az objektumfájlokhoz. Első gondolatunk lenne, hogy akkor manuálisan legépeljük minden .c fájlhoz tartozó .o fájlokat, de szerencsére, erre nincs szükség. Make lehetővé teszi, hogy a makrókat, használatukkor egy kicsit módosított alakban kérdezzük le: az utótagokat tudjuk kicserélni egy lekérdezésen belül minden "szónál". POSIX úgy definiálja a szavakat, hogy egy makrón belül; új sorral, sortöréssel vagy szóközzel elválasztott sztring. Ez pontosan megfelel a formátumnak, ahogy megadtuk a fájlneveket a `SRCS` makrónkban. A lehetőséget kihasználva pedig így tudjuk előállítani az objektumfájlokat tartalmazó `OBJS` makrót:

Listázás 11. Az `OBJS` makró előállítása

```
OBJS = $(SRCS:.c=.o)
```

Ez egyben be is mutatta a szintaxist: a makró neve után egy kettőspont (`:`), majd amit le szeretnénk cserélni a szavak végén (`.c`), egyenlőség jel (`=`), majd amire szeretnénk lecserélni a szavak végét (`.o`). Ebben a szintaxisban is látszik a make deklaratív jellege; nem azt kellett megmondani neki, hogy *hogy* kell lecserélni a szó végi karaktereket, csak, hogy *mit* szeretnénk *mire* cserélni.

Az általánosított csere

Minden általunk ismert make támogat egy ennél kicsit flexibilisebb kicserélési algoritmust. Az egyenlőség bal oldalára egy olyan kifejezést kell írni, ami tartalmaz egy százalék jelet `%` és opcionálisan egyéb karaktereket, szintúgy az egyenlőség jobb oldalára. Ha a konkrétan megadott karakterek illeszkednek a bal oldali kifejezés elejére és/vagy végére, akkor a százalékjel által lefedett, nem konkrét karakter értékek behelyettesítésre kerülnek a a jobb oldali százalékjel helyére. Például az `abCDEFgh` szóra a `:ab%gh=xy%yx` helyettesítés a `xyCDEFyx` szavat adja.

Ennél a formátumnál még jelentősen több alternatívát biztosítanak a BSD make implementációk. Ezeket a kiegészítéseket a Makrókibontási bővítmények részben bővebben tárgyaljuk, illetve `make(1)` egy BSD rendszeren teljes részletességgel leírja őket, és használatukat.

3.3. Utótag szabályok

Az utótag szabályokat (suffix rules) a POSIX szabvány inference rules-nak, azaz következtetett szabályoknak hívja. Mi a GNU make dokumentációja szerint a suffix rule elnevezést használjuk, mert GNU make elterjedtsége miatt (ld. GNU/Linux) általánosan elterjedtebbnek tűnik a kifejezés. Emellett sokkal jobban lehet magyarra fordítani.

Jelenleg rendelkezünk az összes forrás és objektumfájlal, amire szeretnénk őket fordítani. Erre makenek a megoldása az utótag szabályok. A fájlok utótagjai (suffix, POSIX kifejezés a fájlkiterjesztésre) között lehetséges összeköttetéseket létrehozni, ami alapján make az összes cél kiterjesztéssel rendelkező fájlt elő tudja állítani az azonos nevű, de a kezdeti kiterjesztéssel rendelkező fájlból. Például megmondhatjuk neki, hogy az összes `.in` végű fájlból elkészítheti a `.out` fájlt, egyszerűen csak egy másolással. Ilyenkor, ha létezik a `file.in`, akkor make el fogja tudni készíteni a `file.out` még akkor is ha nem szerepel explicit a `file.out: file.in` szabály.

Hol találkozunk ilyen problémával? A C fájlok feldolgozásánál: minden `.c` fájlból `.o` fájlt szeretnénk

készíteni, nyilván fordítással. Most, hogy tudjuk mire fogjuk tudni hasznosítani őket, nézzük meg a szintaxisukat. Mint makeben sok mindenre, erre is egy egyszerű, kompakt szintaxis létezik:

Listázás 12. Az utótagszabályok szintaxisa

```
.be.ki: előfeltételek...
    recept
```

Látható, hogy jelentős hasonlóságok vannak a "normál" szabályok szintaxisához. A jelentős eltérés a cél, amely `.be.ki` formában szerepel. Ilyen esetekben make nem egy szabályt csinál, hanem egy átlagos formátumot a szabálynak: amikor egy másik szabály vagy a felhasználó hivatkozik egy `.ki` utótagú fájlra, mondjuk `file.ki`, akkor make úgy tekinti, hogy létezik egy `file.ki: file.be` szabály, aminek előfeltételei—a `file.be` fájl mellett—az előbbi listázásban szereplő `előfeltételek`, receptje pedig ugyaninnen a `recept`.

Kérdezhetné viszont valaki, hogy honnan tudja make, hogy egy utótag szabályt szeretnénk létrehozni, és nem egy fájlt aminek az a neve, hogy `.be.ki`. Nyilván ez is egy valós fájlnév, mégis mi különbözteti meg a többi szabály céljaitól? Make fenntart egy speciális listát, amelyben tárolja azokat a sztringeket, amelyeket utótagnak fog tekinteni. Ha találkozik egy `.s1.s2` formátumú szabállyal ellenőrzi, hogy `.s1` és `.s2` részei-e ennek a listának, és ha igen, akkor egy utótag szabályként értelmezi, különben csak elkészíti szabványos módon `.s1.s2` nevű fájlra a szabályt. Ezt a listát a makefileból a `.SUFFIXES` speciális céllal tudjuk módosítani.

Listázás 13. A .SUFFIXES speciális cél

```
.SUFFIXES: ①
.SUFFIXES: .be .ki ②
```

① Törli a jelenleg utótagnak nyilvánított szövegeket. Ilyenkor egyáltalán nem működnek az utótag szabályok.

② Hozzáadja a jelenlegi listához a `.be` és `.ki` sztringeket.

Beépített utótagok

Mivel make rendelkezik beépített utótagokkal, és ezekhez szabályokkal, így érdemes a `.SUFFIXES` listát mindig kiüríteni használat előtt. A make beépített utótagok mögött az ötlet az, hogy ne kelljen külön minden makefilenak külön implementálnia ugyan azt a parancsot, amit mi az előző fejezetekben tettünk. Mégis, tapasztalat szerint, sokkal többen vannak, akik nem ismerik maket annyira, hogy tudjanak a beépített szabályokról, akár milyen könnyen is hozzáférhető ez az információ, így nem érdemes erre alapozni a fájlainkat, ha szeretnénk, hogy más is értse őket. Mellesleg a teljes lista sok projekt számára fölösleges elemeket is tartalmaz: ki mikor látott utoljára egy fortran forrást? És ha ez még nem lenne elég ok kerülésükre, a make implementációk nem igazán ragaszkodnak a POSIX szabvány előírásaihoz: GNU make például nem implementálja a `.c.a` szabályt. Ezt személyes tapasztalat útján nem volt kellemes felfedezni, úgyhogy ajánlott teljesen elkerülni az összes beépített szabályt.

Itt, az utótag szabályoknál, jön be a szerepe a már említett `<` makrónak. Ha megnézzük, hogy hogy néz ki és működik egy utótag szabály akkor látjuk, hogy van egy olyan előfeltétel, amit nem írtunk ki.

Egy *implicit* előfeltétel, mégpedig a megegyező nevű forrásfájl. Erre a mindenkori előfeltételre tudunk hivatkozni a `<` makróval egy utótag szabály receptjében. Ezzel már tudunk írni egy olyan utótag szabályt, ami minden forrást lefordít. Az előző forrásból kiindulva, ha kicseréljük a konkrét fájlneveket egy utótag szabályra, és beillesztjük a `.SUFFIXES` szabályt, a következőt kapjuk. Emellett beillesztettük még a forrásfájlok listáját és az azokból képzett objektumfájlokat, ahogy az előző fejezet végén definiáltuk őket.

Listázás 14. A fordító makefile

```
.SUFFIXES:
.SUFFIXES: .c .o

CFLAGS = -Wall -Wextra -Wpedantic -Iext/include -std=c11

SRCS = src1.c src2.c src3.c src4.c
OBJS = $(SRCS:.c=.o)

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

3.3.1. Szimpla utótag szabályok

Az előbbiekkel ellentétben, lehetséges az utótag szabályokat egy darab utótaggal létrehozni. Ezek (általános használat szerint) futtatható fájlokat hoznak létre egy darab megadott fájlból, de akármilyen kiterjesztés nélküli fájl létrehozására is használható. Például egy `ls.c` fájl, ami mondjuk az `ls` parancs forrása, egy ilyen szabállyal egyszerűen fordítható. Ennek szintaxisa egyszerűen csak a cél fájl utótagjának kihagyásával képezhető a normális utótagszabályokból:

Listázás 15. Az szimpla utótagszabályok szintaxisa

```
.be: előfeltételek...
    recept
```

Minden más tekintetben pontosan ugyan úgy működnek, mint az általános utótagszabályok, úgyhogy nem tárgyaljuk őket ennél részletesebben.

Ha valamilyen platformfüggetlenségre szeretnénk legalább minimálisan törekedni, próbáljuk elkerülni az ilyen formátumú utótagszabályokat: UNIX és származékai általában nem használnak futtatható fájlokhoz specifikus utótagot, de például Windowson a `.exe` kiterjesztés nem elhagyható a fájl nevéből. Ilyen esetekben definiálhatunk például egy `EXE_SFX` makrót, amit parancssorból beállíthat mindenki a platformnak megfelelően. Kényelmesebb megoldásról később lesz szó.

3.3.2. Feladatok

1. A fejezetben szó volt a `.in` fájlok `.out` fájlba másolásáról, mint lehetséges utótag szabály. Implementáljuk ezt.

3.4. Linkelés

Már csak az utólsó lépés hiányzik a minimális makefileunkból: A linkelés folyamata, amiben a meglévő objektumfájlokat összecsomagoljuk egy—jelen esetben—futtatható fájlba. A feladat leírásnak megfelelően, szükségünk van a `libfancy` könyvtárra is, hogy el tudjuk készíteni a programunkat. Ehhez két flaget kell bevezetni: a `-L` kapcsoló, ami a linkernek mondja meg, hogy mely elérési útvonalakon keressen még az alapértelmezett mellett könyvtárakat és a `-l` opció, ami pedig magát a könyvtár használatát írja elő.

Érdemes, mint fordításnál a `CFLAGS` makróban, egy makróba eltárolni a linkelési flageket. Bár lehetséges az összes szükséges kapcsolót egy makróban tárolni, két külön makróra bontjuk a flageket. Erre a fő indítékot a A GNU Linker és a könyvtárparaméterek szekció tartalmazza. Két makrónk lesz tehát: az egyik az általános flageket fogja tartalmazni, ez lesz `CFLAGS` névvel analóg módon `LDFLAGS`, a másik pedig külön a linkelendő könyvtárakat, ennek `LIBS` a neve. Konstruáljuk is meg őket:

Listázás 16. A linkelési kapcsolók

```
LDFLAGS = -Lext/lib ①
LIBS    = -lfancy ②
```

① A feladat szerint itt található a libfancy könyvtárfájl, ezt el kell mondai a linkernek a `-L` flaggel.

② A libfancy könyvtárat kell linkelni.

Fontos, hogy a következőkben nem direkt hívjuk meg a linkert, hanem a fordítón keresztül. Ennek nagyon egyszerűen az az oka, hogy a fordító tudja pontosan, milyen könyvtárakat kell linkelni az adott nyelvhez. Például a C fordító tudja, hogy, melyik C könyvárat és rendszerkönyvtárakat kell használni, hogy működjön a program. C++ esetén ez még jelentősebb, mert ott szükség van—általában—a C könyvtárra, a C++ könyvtárra, esetlegesen a fordító által biztosított kivételkezeléshez kapcsolatos könyvtárra. Ezt mind kitalálhatjuk egy adott (verziójú) operációs rendszer/fordító kombinációra, de ha legalább minimálisan érdekel a platformfüggetlenség, nem ezzel fogjuk tölteni az időnket, hanem azzal, hogy a fordítót úgy próbáljuk befogni, hogy a mi célunkat szolgálja.

A következő lépésben még egy makrót hozunk létre, mégpedig a fordítandó program nevére. Ez a lépés nem feltétlen fontos, de érdemes külön makróba tárolni, hogy egyértelmű legyen, amikor a fordítandó programra hivatkozunk. Igazából egy szimbólikus konstans, ami a fájl értelmezését teszi könnyebbé, mint C++-ben egy `constexpr` változót hoznánk létre, vagy akár egy C makrót: az olvashatóságot javítja, de külön jelentőségge makró voltának nincs. Legyen a neve `EXE`, ezzel is utalunk az elékszítendő fájl típusára, hiszen egy futtatható fájlt készítünk. Érékte nyilván `fancifier`, a program amit el szeretnénk készíteni.

Komplexebb fájlokban, amelyek több kimeneti fájlt is készítenek, érdemes kiírni a fájl nevét is. Például, ha mi építenénk a `libfancy.a` fájlt is, akkor például lehetne a két makró `FANCY_LIB` és `FANCIFIER_EXE`.

A GNU Linker és a könyvtárparaméterek

Az általános POSIX működéssel ellentétben a GNU linker (`ld(1)` GNU rendszereken) kapcsolók működése… érdekes. Általában vagy legelőre kell elhelyezni a kapcsolókat, ahogy POSIX írja elő, vagy teljesen mindegy a sorrendjük és a többi pozicionális paraméter között bárhol elhelyezkedhetnek. A linker `-l` kapcsolói azonban a parancs legvégére **kell**, hogy kerüljenek, különben hibákkal fogunk találkozni. Ennek oka egyszerűen az, ahogy a linker kezeli a paramétereit: sorban halad a paramétereken és összeszedi az összes szimbólumot (függvények, globális változók, stb.), amire hivatkoznak és azokat, amikkel találkozik. Ha egy könyvtárral találkozik azonban, akkor nem tárolja el, hogy mi az összes szimbólum, amit az a könyvtár tartalmaz, hanem csak beilleszti a jelenleg készülő fájlba az összes olyan szimbólumot, amire már valaki hivatkozott a könyvtárból, majd megy tovább a következő paraméterre. Így, ha egy a könyvtár után található objektumfájl hivatkozik a könyvtár egy olyan szimbólumára, amit más előtte nem használt, akkor azt a szimbólumot soha nem fogja “megtalálni” a linkelés pedig hibával fog kilépni.

A következő példa mutatja be ezt a jelenséget.

Listázás 17. A main.c fájl

```
int
a(void); /* liba.c függvénye */

int
main() {
    return a();
}
```

Listázás 18. A liba.c fájl

```
int
a(void) {
    return 0;
}
```

Ha a `liba.c` fájlt lefordítjuk egy statikus könyvtárrá, majd amikor megpróbáljuk a linkelést, a kövezkezőt tapasztaljuk, mint egy segítőkész kolléga GNU/Linux gépén natív GNU eszközökkel próbálkozva:

```
$ cc -c -o liba.o liba.c
$ cc -c -o main.o main.c
$ ar csr liba.a liba.o
$ cc -o exe -L. -la main.o
/usr/bin/ld: main.o: in function `main':
main.c:(.text+0x9): undefined reference to `a'
$ cc -o exe -L. main.o -la
$ ./exe
```

Ezt a makrót felhasználva hozzuk is létre a szabályt. Ha végiggondoljuk, hogy milyen előfeltételeket kell felsorolnunk, akkor az összes objektumfájlt kapjuk. Szerencsére létrehozunk már egy makrót a kibontási módosító bemutatására, ami pont ezeket a fájlokat tárolja. Ezt felhasználva már könnyedén létre tudjuk hozni a szabályt, igazából csak a már meglévő makrók kombinálása.

Listázás 19. A linkelési szabály

```
$(EXE): $(OBJS)
    $(CC) -o $@ $(LDFLAGS) $(OBJS) $(LIBS) ①
```

① Felhasználjuk a A GNU Linker és a könyvtárparaméterek szekcióban leírt kompatibilitási megfontolást és a `LIBS` makrót a parancs végén bontjuk ki.

A felhasznált parancsot gyorsan áttekintjük: a `CC` makróval meghívjuk a C fordítót, a `-o` kapcsolóval megmondjuk neki, hogy hova produkálja a kimenetet (ugye a jelenlegi cél nevére a `@` makró értékéből), majd az `LDFLAGS` makróból az általános linkelési opciókat, az előfeltételként felsorolt objektumfájlok és végül a `LIBS` makróban tárolt használandó könyvtárakat.

Érdemes még egy kényelmi funkciót implementálni. A felhasználó lehet, hogy szeretne egyéb flageket a fordítónak, például az optimalizációt szeretné bekapcsolni, vagy esetleg a platformjának szüksége van valamilyen saját könyvtárra, amit be kell állítani a linkernél. Ehhez minden makróhoz létrehozunk egy `PRE_*` verziót, amit parancssorból könnyedén lehet állítani és platformra szabni make működését:

Listázás 20. Állítható fordítási flagek

```
CFLAGS  = $(PRE_CFLAGS) -Wall -Wextra -Wpedantic -Iext/include -std=c11
LDFLAGS = $(PRE_LDFLAGS) -Lext/lib
LIBS    = $(PRE_LIBS) -lfancy
```

Ha még beillesztjük a hamis szabályok bevezetésénél tárgyalt `clean` és `all` szabályokat, megkapunk egy működőképes makefilet, ami már le tudja fordítani a teljes projektünket.

Egy apró probléma fennáll még azonban. A fejléc fájlok módosítása nem rendel el újrafordítást a forrásokon, amelyek #includálták őket. Erre a problémára sajnos nem tudunk automatikus és szabványos megoldást biztosítani, csak különféle trükköket, amely a fordító és/vagy a make implementáció saját bővítményeire épít. A szabványos megoldás azonban kis projekteknél még működhet, ezért ezt fogjuk most bemutatni. A GCC `-MD` flagjének használatát a Fejléc függőségi gráf fejezet tárgyalja GNU make kapcsán, BSD make `mkdep(1)` programjának működése pedig a mkdep(1) fejezet témája.

A szabványos megoldást kétféle módon lehetne megközelíteni, viszont egyik kényelmetlenebb, mint a másik. A egyik megoldás, hogy minden forrásnak minden fejlécet beállítjuk függőségként. Ez nagyon gyorsan implementálható, de csak nagyon kis méretű projekteknél használható, hiszen egy fejléc fájl módosítása a teljes projekt újrafordítását eredményezi. Érezhető, hogy ez nem túl jó, főleg ha mondjuk egy kernelt szeretnénk fordítani több száz forrásból. Kis projekteknél azonban működőképes megoldás kb. egy-két tucat forrásig (főleg gyorsan forduló C sorrások esetén, de még talán C++-ban is egy tucat fájl még tűrhető) utána már problémát okozhat. Sajnos szabványosan ezt nem tudjuk megkerülni; a másik elképzelhető megoldás ugyan nem rendelkezik ezzel a hibával, de egyáltalán nem alkalmazható nagy projektekre. Ez csupán annyit rejt, hogy manuálisan kilistázzuk

az összes fejléc függőséget minden forrásnál, és manuálisan tartjuk ezt a listát karban.

Az előbb tárgyalt fejléc függőség implementációja elég egyszerű: felsoroljuk a fejléceket majd informáljuk maket, hogy minden forráshoz tartozó objektumfájl függ minden fejléctől. A legegyszerűbben ezt az utótagszabályunkban tehetjük meg. Ezt implementálva kapjuk meg a végső makefileunkat.

Listázás 21. A teljes makefile

```
.SUFFIXES:
.SUFFIXES: .c .o
.PHONY: all clean

CFLAGS  = $(PRE_CFLAGS) -Wall -Wextra -Wpedantic -Iext/include -std=c11
LDFLAGS = $(PRE_LDFLAGS) -Lext/lib
LIBS    = $(PRE_LIBS) -lfancy

SRCS = src1.c src2.c src3.c src4.c
HDRS = $(SRCS:.c=.h) foo.h bar.h qux.h
OBJS = $(SRCS:.c=.o)
EXE  = fancifier

all: $(EXE)

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

$(EXE): $(OBJS)
    $(CC) -o $@ $(LDFLAGS) $(OBJS) $(LIBS)

clean:
    rm -f $(OBJS) $(EXE)
```

A .POSIX szabály

A POSIX szabvány azt írja elő, hogy minden makefilenak tartalmaznia kell a `.POSIX` szabályt előfeltételek és recept nélkül az első nem-komment sorban, különben a make implementáció működése meghatározatlan (unspecified). A gyakorlatban ez egy szinte teljesen fölösleges megkötés, hiszen a POSIX szabvány a meglévő make implementációk köré lett specifikálva (a `RATIONALE` szekció a make leírásában tartalmaz is utalásokat különféle makeekre), így azok amúgy is úgy működnek, ahogy az specifikálva van. (Ahol meg nem egyezett meg a szabványosított és a meglévő működés, általában a meglévő maradt, függetlenül a `.POSIX` szabálytól.) Ha valaki mégis nagyon hű szeretne lenni a POSIX szabványhoz (egyedül lenne vele, de adjuk meg neki az esélyt) akkor csak helyezze minden makefileja elejére, hogy `.POSIX:` .

[3] A Windowsos MSVC eszközök a Visual Studio fejlesztő környezetből való használatra vannak optimalizálva. Parancssoros használatuk lehetséges, de annyira különbözik a UNIX szerű rendszerek fordítóitól, hogy érdemes teljesen külön kezelni őket makefileok szintjén. Egyéb megoldás a problémára valamilyen generátor használata: CMake, Meson, stb..

[4] Hivatalosan makróknak hívja őket a POSIX szabvány, de több helyen, például a GNU make dokumentációja változókként referál

rájuk.

[5] FreeBSD-n `style.Makefile(5)`

[6] Ezt akkor lehetne csak elkerülni, ha make—vagy valami más szoftver—biztosra tudna menni, hogy az utoljára végzett glob művelet eredménye még mindig aktuális, anélkül, hogy végrehajtaná magát a glob műveletet.

4. Komplex make trükkök

4.1. Könyvtár szabályok

WIP

Függelék A: A C build modell

Ez a függelék egy rövid kitekintő alkalmával a C build modellt írja le. Ugyan ezen leírás érvényes a C++ build modelljére is, mely C++20-ig teljesen megegyezik a C build modelljével. Ez a dokumentum nem tárgyalja a C++20 build modellt, mert jelentősen nem tér el az eddigitől, és nem terjedtek el eléggé a module-ok, hogy általánosan elfogadott és jó megoldásokat mutassunk be használatukra. Utóbbi a jövőben változhat.

A legelső jellemzője a C build modellnek, hogy nagyon öreg. Nem gondolnánk bele, hogy ez mennyire fontos, de mégis a C, és ezen keresztül, C++ nyelvek működését jelentősen befolyásolja koruk; a build modell szempontjából pedig azért jelentős, mert olyan gépekre lett tervezve, amik nem voltak képesek több fájl memóriában tartására, hiszen a 16-bites processzorok 64 KiB-es limitációja nem túl sok[7]. Még kisebb projekteknél 2-3 kisebb fáljt meg tudtak volna etetni egy fordítóval, de ugyan úgy ahogy a mai gépeink sem lennének képesek egy fordító hívással lefordítani a NetBSD kernelt, az akkori gépek se tudták lefordítani a UNIX kernelt.

Ennek megfelelően a C fordítás 2+1 lépésből áll.[8]

- Az első lépés az úgy nevezett előfeldolgozás (preprocessing, `cpp(1)`). Ilyenkor az előfeldolgozó, történelmileg a `cpp` program, ma már inkább a fordítóba integrált lépésként történik, értelmezi az `#include`, `#define` és egyéb neki szóló direktívákat. Többé kevésbé, az így kapott fájlok tekinthetőek a C és C++ szabványokban definiált fordítási egységeknek (translation unit).

  > Manapság a fájlok alacsony méreten tartását akadályozzák inkább a fejlécfájlok és az előfeldolgozó. A következő C++ forrásfájl előfeldolgozás után a Windows gépemen Visual Studio 2022-t használva 1,4 MiB lesz. Más implementációkkal is hasonló eredményre jutunk.

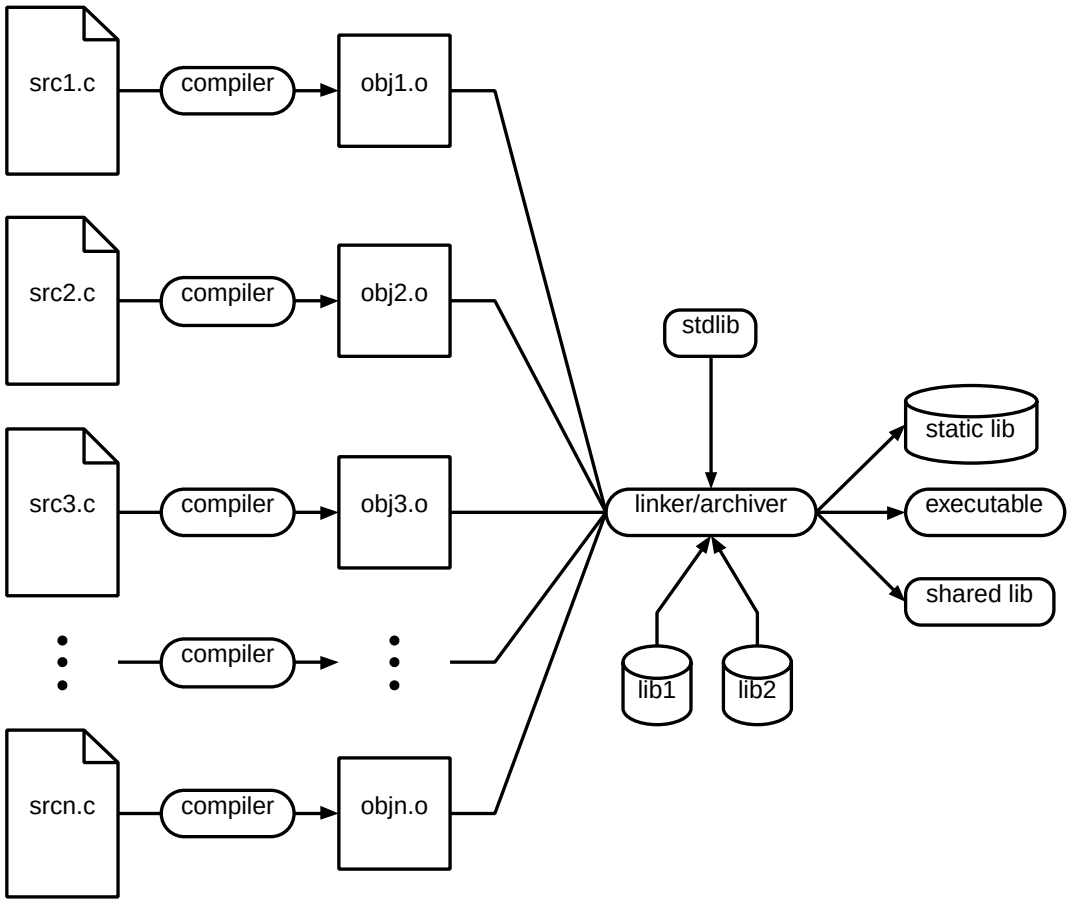
  ```
  #include <iostream>

  int
  main() {
      std::cout << "Hello world!\n";
  }
  ```

- Ez után következik a fordítás. Ilyenkor a fordítóprogram (compiler, `cc(1)` (tradícionális UNIX) vagy `c99(1)` (POSIX)) fogja az előfeldolgozott forrásfájlt, majd a megcélzott processzornak megfelelő “gépi kód” fájlt készít. Ezeket a fájlokat nevezzük objektumfájlnak (object file). Bár gépi kódként referáltunk rájuk az előbb, ezek a fájlok nem futtathatóak magukban, hiszen hiányzik több minden mellett, például az operációs rendszernek megfelelő, és nagyon különböző indítási kódrészlet. Amit tudnak viszont, hogy a C fájlban definiált függvények a lefordított utasításaikkal már szerepelnek benne.
- A harmadik lépés kicsit többrétűbb az eddigieknél, de mindenképpen a sok különálló objektumfájl valamilyen módon való összeillesztéséről van szó. Ez a lépés nagyon platformfüggő, de teljes általánosságban linkelésnek (linking) lehet nevezni.
  - A legegyszerűbb dolog amikor az archiváló (archiver, `ar(1)`) összefogja az összes objektumfájlt és egy nagy archívumot csinál belőle, egy plusz listával ami tartalmazza az

összes tárolt fájlban található függvényeket. Ezt utána más programok/könyvtárak linkelésekor használható.

- Utána érkezik a megosztott könyvtár ((dynamic) shared object/dynamically loadable library) és a futtatható fájlok. Bár működés szempontjából a kettő jelentősen különbözhet, elkészítésük általában nagyon hasonló, így itt egybe jegyezzük őket. Ilyenkor a linker (`ld(1)`) a bemeneti fájlokat és a megfelelő operációs rendszer fájlokat összeköti, jegyzi az esetlegesen futásidőben automatikusan betöltendő könyvtárakat és generál egy kimeneti fájlt, ami futtatható ha futtatható fájlt generáltunk.



Ábra 2. A C Build Modell

Az előfeldolgozást leszámítva az A C Build Modell ábra mutatja be az előbb leírtakat vizuálisan.

[7] Még ha léteztek is megoldások ezt a limitációt megkerülni, nem számítottak feltétlen a legjobbaknak. De még ha ez is megoldható lett volna, abban az időben csupán a memória ára is szükségessé tette, hogy (fölösleges) használatát minimalizálják.

[8] Történelmileg egyértelműen megkülönböztethető a 3 lépés, de mára már az előfeldolgozás szinte teljesen beleolvadt a fordításba egy folyamaton belül.

Függelék B: GNU make bővítmények

Ebben a függelékben a talán legelterjedtem make implementáció által biztosított bővítményeket ismertetjük. Ez a GNU make, és míg natív eszköze a GNU felhasználói világot használó (userland) rendszereknek, a legtöbb platformon és operációs rendszeren, így a különböző BSD variánsokon és Windowson, is elérhető. Legújabb verziója 4.3.

A többi GNU eszközhöz hasonlóan (ld. GCC) a GNU make is bővítmények tágas tárházával rendelkezik. Nem tárgyaljuk az összeset, viszont a fontosabb és gyakran használtakat bemutatjuk, hogy érthető legyen a vadonban található GNU makefileok többsége. Minden információ megtalálható a GNU make online dokumentációjában vagy egy GNU rendszeren a megfelelő `info(5)` pageben.[9]

B.1. Minta szabályok

A minta szabályok (pattern rule) a GNU make által bevezetett új szabálytípus, aminek célja az utótagszabályok általánosítása. Míg utótagszabályokban—nevükhöz híven—a fájlok utótagjaira lehet hivatkozni, minta szabályok esetén általánosabb mintákat lehet illeszteni a fájlokra. A minták egyszerűen egy százalékjelet tartalmaznak, mely helyén bármely karaktersorozat tartózkodhat, és így minden olyan fájl ami illik erre a példára hivatkozható. A százelékjel által eltakart karaktereket az elsődleges előfeltételben (az első előfeltétel) lehet felhasználni, szintén a százelékjel felhasználásával: itt kicserélődik az összes általa lefedett karakterre.

A következő példa mutatja, hogy felhasználhatóságuk teljesen lefedi az utótag szabályokét, így minden esetben használhatóak helyettük.

Példa 5. Minta szabály, mint általánosított utótagszabály

```
%.o: %.c
    cc -c $(CFLAGS) -o $@ $<

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

Ez a két szabály ekvivalens: bármely hivatkozott objektumfájlt előállítja a megfelelő nevű C forrásfájlból.

Emelett viszont használható sokkal általánosabb módon is. Erre mutatnak példát a következők.

Példa 6. Általános minta szabály

A következő példa a `bison(1)` parser generátor által generált fájlokhoz kapcsolódik. Ez az eszköz egy bemeneti `.y` fájlból egy `.tab.c` és egy `.tab.h` fájlt készít, melyek a C implementációját tartalmazzák a bemeneti fájlban leírtaknak.

Látható, hogy ezt nem igazán lehet leírni a szabványos eszközökkel, minta szabályokkal azonban egyszerűen elkészíthető a cél és az elsődleges előfeltétel.

```
%.tab.c %.tab.y: %.y
    bison -d $<
```

Ebben ez esetben mind két fájl elkészíthető egy darab parancs futtatásával. Általában ezt két külön szabállyal lehetne megoldani, ami két külön parancsot futtatna le.

POSIX `yacc(1)`, bison szabványosított megfelelője, integrálása POSIX makeben úgy történt, hogy a fejléc fájl generálása el lett hagyva. (A `-d` flag az előző bison parancsban.) Ez azonban jóval kényelmetlenebben kezelhető és integrálható parser kódot generál, hiszen a bemeneti tokenek makró definicói nem elérhetőek automatikusan.

Látható, hogy ez az általánosítás azonos a Az általánosított csere megjegyzésben tárgyalt általánosított csere algorithmussal. Szintén analógia, hogy a szabvány szerinti működés csak a kiterjesztéseket tudja kezelni, a bővítmény által általánosabb részletek cserélhetőek ki.

B.1.1. Statikus minta szabályok

B.2. Függőség nélküli előfeltétel

B.3. Függvények

B.4. Utasításirányítás

B.5. Fejléc függőségi gráf

[9] Egyes rendszereken lehetséges, hogy a dokumentáció külön csomagban található, így azt külön kell telepíteni. Például Debian és azon alapuló rendszereken a `make-doc` csomagra van szükség.

Függelék C: BSD make bővítmények

C.1. Makrókibontási bővítmények

C.2. mkdep(1)

C.3. Utasításirányítás

Függelék D: Megoldások

D.1. Egy forrás fordítása fejezet:

1. feladat

```
greet.txt:
    echo "Hello world" > greet.txt
```

2. feladat

```
.PHONY: greet
greet: greet.txt

greet.txt:
    echo "Hello world" > greet.txt
```

3. feladat

Azért nincsenek előfeltételei clean-nek, mert akkor elkészítené a jelenleg nem létező, vagy nem friss célokat, csak azért, hogy utána kitörölje. Ezért fogalmazzuk úgy meg a receptet, hogy sikeresen lefusson, akkor is, ha nem léteznek fájlok amiket törölne.

```
.PHONY: greet clean

greet: greet.txt

greet.txt: input.txt
    cat input.txt > greet.txt

clean:
    rm -f greet.txt
```

4. feladat

```
source.o: source.c
    cc -c -o source.o -Wall -Wextra -Wpedantic source.c

archiv.tar: source.o source.c
    tar cf archiv.tar source.o source.c
```

D.2. Utótag szabályok fejezet:

1. feladat

```
.SUFFIXES: .in .out
.in.out:
    cp $@ $<
```